Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ФГБОУ ВО «АмГУ»)

Факультет математики и информатики
Кафедра математического анализа и моделирования
Направление подготовки 01.04.02 Прикладная математика и информатика
Направленность (профиль) образовательной программы Математическое и программное обеспечение вычислительных систем

ДОПУСТИТЬ К ЗАЩИТЕ
И.о. зав. кафедрой
_____ Н.Н. Максимова
«15» 06 2018 г.

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему: Алгоритм и программная реализация метода декомпозиции расчетных сеток для параллельной вычислительной системы

Исполнитель
студент группы 652 ом _____ 10.06.2018 (подпись, дата) А.А. Казанцев

Руководитель
доцент, канд. техн. наук _____ 13.06.2018 (подпись, дата) А.В. Рыженко

Руководитель научного
содержания программы
магистратуры _____ 14.06.2018 (подпись, дата) А.Г. Масловская

Нормоконтроль
доцент, канд. техн. наук _____ 11.06.2018 (подпись, дата) А.В. Рыженко

Рецензент
доцент, канд. техн. наук _____ 15.06.2018 (подпись, дата) Н.П. Семичевская

Благовещенск 2018

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ФГБОУ ВО «АмГУ»)

Факультет математики и информатики
Кафедра математического анализа и моделирования

УТВЕРЖДАЮ
И.о. зав. кафедрой
_____ Н.Н. Максимова
«26» 02 2018 г.

З А Д А Н И Е

К магистерской диссертации студента Казанцева Андрея Александровича.

1. Тема магистерской диссертации: Алгоритм и программная реализация метода декомпозиции расчётных сеток для параллельной вычислительной системы (утверждена приказом от 13.02.2018 № 319-уч).
2. Срок сдачи студентом законченной работы: 14.06.2018 г.
3. Исходные данные к магистерской диссертации: отчет по преддипломной практике, средства автоматизации вычислений – ППП Matlab, интегрированная среда разработки–Microsoft Visual Studio.
4. Содержание магистерской диссертации (перечень подлежащих разработке вопросов): исследование задачи сортировки и декомпозиции на параллельной вычислительной системе, описание и реализация метода декомпозиции с применением чётно-нечётного алгоритма сортировки слиянием.
5. Перечень материалов приложения: листинг программной реализация метода рекурсивной координатной бисекции для декомпозиции расчетных сеток со слиянием Бэтчера; листинг кода корректного вывода результатов работы программы; листинг программы расчёта экспоненты с применением технологии MPI.
6. Консультанты по магистерской диссертации: рецензент – Семичевская Н.П., канд. техн. наук, доцент; нормоконтроль – Рыженко А.В., канд. техн. наук, доцент.
7. Дата выдачи задания:26.02.2018 г.

Руководитель магистерской диссертации: Рыженко Андрей Викторович доцент, канд. техн. наук

Задание принял к исполнению (26.02.2018): _____ Казанцев А.А.

РЕФЕРАТ

Магистерская диссертация содержит 107 с., 15 рисунков, 4 таблицы, 3 приложения, 30 источников.

РАСЧЁТНАЯ СЕТКА, ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ, MPI, ДЕКОМПОЗИЦИЯ, СОРТИРОВКА СЛИЯНИЕМ БЭТЧЕРА, C++, ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ

Актуальность параллелизации методов декомпозиции расчётных сеток обуславливается тем, что при распараллеливании процесса декомпозиции существенно уменьшается нагрузка на вычислительную систему, что даёт возможности для более точных численных решений методами конечных разностей и конечных элементов на расчётной сетке.

Целью магистерской диссертации является анализ методов декомпозиции и параллельных вычислительных систем, создание параллельного алгоритма декомпозиции расчётной сетки, а так же программного кода основанного на нём.

Получены следующие новые результаты:

– создан параллельный алгоритм декомпозиции основанный на алгоритме чётно-нечётной сортировки слиянием Бэтчера на основе технологий MPI;

– создан программный код декомпозиции основанный на алгоритме чётно-нечётной сортировки слиянием Бэтчера на основе технологий MPI;

– адаптирован алгоритм чётно-нечётной сортировки слиянием Бэтчера для параллельной архитектуры вычислительной системы;

– приведены результаты численных экспериментов с различными параметрами задачи.

СОДЕРЖАНИЕ

ОБОЗНАЧЕНИЯ

БИС – большие интегральные системы;

ВС – вычислительная система;

МКМД – множественный поток команд – множественный поток данных;

МКОД – множественный поток команд – одиночный поток данных;

ММС – многомашинные вычислительные машины;

МПС – многопроцессорные системы;

ОКМД – одиночный поток команд – множественный поток данных;

ОКОД – одиночный поток команд – одиночный поток данных;

ПК – персональный компьютер;

ЭВМ – электронно-вычислительная машина;

MPI – message passing interface.

ВВЕДЕНИЕ

Для численного решения конечно-разностными методами или методом конечных элементов уравнений в частных производных используются расчётные сетки, которые описывают физическую область в дискретной форме.

От того на сколько будет точно численное решение зависит эффективность численного исследования, так же не маловажным фактором является время потраченное на вычисления. Точность численного решения в физической области зависит от ошибки интерполяции и ошибки решения в узлах сетки. Основные причины погрешности численных расчётов в узлах сетки следующие:

1) отображение физического явления математической моделью не является абсолютно точным;

2) на этапе численного приближения математической модели всегда существует, чаще всего известная, погрешность;

3) ошибка может возникать из за размеров и форм ячеек сетки;

4) ошибка, внесённая вычислением дискретных физических величин, удовлетворяющим уравнениям численного приближения;

5) ввиду неточности процесса интерполяции дискретного решения появляется погрешность. Известно, что в численном анализе физических задач в контроле третьей и пятой причины погрешностей существенную роль играют качественные и количественные свойства сетки.

Для численного решения краевых задач в многомерных областях существует два фундаментальных класса сеток: структурированные и неструктурированные.

В большинстве практических задач форма объекта сложна и с трудом поддается обработке только структурированными методами. Структурированные сетки часто имеют недостаток гибкости и устойчивости, при работе с сложными границами областей, может происходить деформация и искривление ячейки, в следствии чего будут производится неэффективные расчеты. Одним из решений проблемы построения сетки в областях сложной

формы стала концепция неструктурированной сетки.

Недостатком неструктурированных сеток является серьёзное усложнение численного алгоритма, так как требует дополнительной обработки данных, которые выражаются в нумерации узлов, граней, рёбер, что так же требует создание отдельного алгоритма, увеличение необходимой памяти для хранения большего количества информации о связях ячеек. Из этого вытекает дополнительная вычислительная работа связанная с увеличением, как числа элементов сетки, так и ее характеристик. Как результат, сетки с неструктурированной топологией сильно загружают память на один узел, а так же обладают большей трудоемкостью касаемо операции на шаг времени.

С появлением параллельных компьютеров и развитием параллельных вычислителей сократилось время получения решения на неструктурированных сетках. На данный момент, большинство программ решающих уравнения механики адаптированы под параллельные алгоритмы. Так же начиная с 2007 г. активно создаются параллельные алгоритмы построения сеток, однако алгоритмы бисекции области, необходимые для параллельной реализации, как самого вычислительного алгоритма, так и алгоритма построения расчетной сетки остались далеко позади. Предварительный процесс бисекции расчетной области остается узким местом, где вычисления выполнятся последовательно.

Сетки размером, превышающим 10^7 элементов физических объектов уже становятся обычными для промышленного моделирования в вычислительной электродинамике [16, 24] и вычислительной аэрогидродинамике [17, 18, 23, 28]. Ожидается, что в скором будущем будут требоваться сетки, превышающие $10^8 \div 10^9$ элементов [19].

С таким увеличением размера сетки процесс ее бисекции на одном последовательном компьютере становится затруднительным, а зачастую и невозможным в плане вычислительного времени и требований к памяти.

Из этого следует, что параллелизация процесса автоматической бисекции сетки позволит повлиять на проблемы построения больших сеток, а именно: нехватку памяти и длительное время вычислений. Поскольку большое

количество научных приложений испытывают нехватку памяти. Производительность современных процессоров простаивает из-за нехватки пропускной способности запоминающего устройства и нехватки памяти. Параллельная биекция позволит сократить и время построения сетки. Это крайне полезно, когда требуется частое присчитывание сетки.

Целью работы является разработка алгоритма и программная реализация метода декомпозиции расчётных ?.

Для достижения поставленной цели необходимо выполнить следующие задачи:

1) разработка параллельного алгоритма автоматической декомпозиции сеток;

2) сравнение и анализ алгоритмов параллельной декомпозиции;

3) создание комплекса программ для параллельной декомпозиции расчётных сеток.

В результате работы создан и проанализирован алгоритм декомпозиции расчетных сеток основанный на параллельной архитектуре вычислительной системы.

Алгоритм может иметь широкое применение в построении сложных расчетных сеток, как отдельная программа, так и в составе кода.

Работа апробирована в рамках двух межвузовских («Молодежь XXI века: шаг в будущее», г. Благовещенск, 2017-2018 гг.) и двух вузовских научных конференций («День науки АмГУ», г. Благовещенск, 2017-2018 гг.). Результаты магистерской диссертации обсуждались на научных семинарах выпускающей кафедры. Результаты опубликованы в двух научных работах (материалов докладов научных конференций).

Магистерская диссертация состоит из введения, трёх глав, заключения, списка литературы, трёх приложений, 11 рисунков и 5 таблиц. Объём работы 92 страницы. Библиографический список включает 28 наименований.

1 ИНСТРУМЕНТАРИЙ И ПРИНЦИП ПОСТРОЕНИЯ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

1.1 Архитектура вычислительных систем

Вычислительная система, совокупность аппаратно-программных средств, образующих единую среду, предназначенную для решения задач обработки вычислений. В начале развития универсальные вычислительные системы создавались на основе однопроцессорных ЭВМ с целью увеличения быстродействия. В первых ЭВМ процессоры сами управляли операциями ввода-вывода, но скорость работы внешнего устройства значительно меньше скорости работы процессора, из-за этого во время операций ввода-вывода процессор простаивал. Чтобы оптимизировать их работу, в конце 1950-х – начале 1960-х годов ЭВМ начали комплектовать несколькими независимыми процессорами ввода-вывода для параллельного выполнения вычислений и операций обмена данными, именно тогда и появился термин «Вычислительная система». Основными достоинствами вычислительной системы по сравнению с однопроцессорной ЭВМ являются:

1) весомое повышение производительности за счет статического или динамического распараллеливания процесса решения задачи

2) повышение эффективности использования оборудования за счёт более плотной его загрузки;

3) повышение надёжности и т.п.

1.1.1 Классификация вычислительных систем

Термин «вычислительная система» появился в середине – конце 60-х годов с момента создания ЭВМ третьего поколения. Это время знаменовалось переходом на новую элементную базу – интегральные схемы [1].

Результатом стало появление новых идей реализации, в системе применяемых вычислительных систем, а так же и в основах управления. В настоящее время существует огромное количество определений термина «вычислительная система», но общим для всех них является создание

параллельных ветвей расчетной нагрузки, что отличает вычислительную систему от классической ЭВМ.

Вычислительная система – это совокупность взаимосвязанных и взаимодействующих процессоров или ЭВМ, периферийного оборудования и программного обеспечения, предназначенную для сбора, хранения, обработки и распределения информации [1]. Главным отличием вычислительной системы по сравнению с классическим компьютером является наличие в них некоторого количества процессоров, выполняющих обработку данных параллельно. ВС создаются для уменьшению времени работы процессов обработки, что ведет к улучшению общей производительности, увеличению точности и надёжности вычислений.

Основными причинами появления и развития вычислительных систем стали экономические факторы. Анализ характеристик ЭВМ показал, на данный момент зависимость мощности вычислений от стоимости выражается квадратично, то есть в ближайшем будущем, введу быстрого роста необходимой потребности нагрузки, стоимость ЭВМ будет чрезмерно высока.

Для ЭВМ и вычислительных систем существует свой порог решения задач. Критический предел находится точкой пересечения двух приведенных зависимостей. Помимо выигрыша в стоимости, вычислительные системы позволяют решать более широкий спектр задач таких как: надежность, достоверность результатов обработки, резервирования, централизация хранения и обработки данных, децентрализации управления и т.д.

На данный момент существует большой прикладной опыт в разработке и использовании вычислительных систем. В зависимости от поставленных задач характеристика и архитектура вычислительной среды может сильно различаться. Вычислительные системы классифицируются по большому количеству признаков, несмотря на это главными из них являются отличительные способности структурной и функциональной организации ВС.

По применению вычислительные системы бывают двух видов: универсальные и специализированные. Специализированные системы

применяются для решения узконаправленных задач, в некоторых случаях одной. Универсальные системы применяются во всех остальных случаях.

По типу вычислительные системы разделяются на многомашинные и многопроцессорные. Многомашинные вычислительные системы (ММС) появились раньше. Основные отличия многомашинной вычислительной системы заключаются, в большинстве случаев, в создании связи и передаче информацией между ЭВМ комплекса. Любая из машин работает автономно и управляется собственной операционной системой. Каждая иная подключаемая ЭВМ комплекса рассматривается как внешнее специальное оборудование. Многопроцессорные системы (МПС) проектируются при объединении некоторого числа процессоров. Объединяющим для всех является оперативная память. Параллельная работа вычислителей и применение оперативной памяти обеспечивается под предводительством общей операционной системы. В отличие от ММС здесь достигается максимальная оперативность взаимодействия вычислителей-процессоров.

Несмотря на очевидные преимущества многопроцессорной системы, она имеет и свои недостатки, которые, в большинстве, выражаются из-за использования общей оперативной памяти. При большом количестве процессоров возникают ситуации, при которых некоторое количество вычислителей обращаются к одному и тому же участку оперативной памяти. Помимо процессоров к ней также подключаются вычислители ввода-вывода, таймеры измерения времени и другие компоненты. Также весомым недостатком МПС является сложность коммутации абонентов и доступа их к оперативной памяти. От качества решения этих проблем зависит эффективность многопроцессорной системы. Принятое решение должно обуславливается аппаратурно-программными средствами.

По типу ЭВМ или вычислителей, используемых для создания ВС, различают однородные и неоднородные системы. В однородных системах используются однотипные процессоры, в неоднородных процессоры и ЭВМ могут кардинально различаться. В однородных системах упрощена разработка

операционной системы. В однопроцессорных системах создается возможность стандартизации и унификации соединений и процедур сопоставления элементов системы.

По методам управления элементами вычислительной системы выделяют: централизованные, децентрализованные и ВС со смешанным управлением. Для выполнения параллельных вычислений необходимо выделять ресурсы для управления процессорами или ЭВМ. В централизованных ВС ответственность ложится на главный ЭВМ или вычислитель. Основной функцией этого узла является равномерное распределение вычислительной нагрузки, выделение и контроль состояния ресурсов, координация взаимоотношений между другими элементами. Центральный вычислитель не обязательно четко фиксирован, его задачи могут присваивать себе другие блоки, что способствует увеличению надежности системы. Операционные системы для централизованных систем просты. В децентрализованных системах операции управления разнесены между ее элементами. Каждая ЭВМ или вычислитель системы частично автономен, а необходимое взаимоотношение между элементами создается по специальным наборам сигналов.

1.2 Источники параллелизма и локальности в моделировании

Рассмотрим общие источники параллелизма и локальности в четырех основных видах моделирования реального мира. Изучив каждый вид моделирования кратко, мы рассмотрим последний вид более подробно. В частности, мы будем выводить и решать уравнения для теплового потока в квадратной пластине и для гравитационного потенциала в квадратной области. В обоих случаях вычислительное узкое место будет одинаковым.

1.2.1 Источники параллелизма и локальности

Разделим моделирование явлений реального мира на четыре широкие категории, в зависимости от того, какое основное представление они используют. Возможно, что данный физический феномен представлен более чем одним из этих способов. Каждое представление состояния будет иметь связанный набор управляющих уравнений или правил, который показывает, как

один вычисляет или манипулирует состоянием. Каждая система уравнений или правил будет обладать определенным естественным параллелизмом и локальностью и приведет к набору методов параллельного решения.

1.2.2 Моделирование дискретных систем событий

Эти системы представлены как конечный набор переменных, каждый из которых может принимать конечное число значений. Простейшими примерами являются проблемы с акулами и рыбой, где акулы и рыба перемещаются по дискретной сетке в соответствии с определенными правилами. Состояние представлено списком обитателей каждой ячейки сетки. На отдельных временных шагах каждая рыба и каждая акула могут перемещаться в соседнюю ячейку (или есть или размножаться или умирать) в соответствии с определенными правилами.

Следующий пример, который, пожалуй, наиболее близко знаком с цифровой схемой в компьютере. Самый простой способ моделирования цифровой схемы – на функциональном уровне , где состояние состоит из конечного числа 1s и 0s, представляющих все присутствующие высокие и низкие напряжения. Управляющими правилами являются набор логических уравнений, описывающих переходы состояний; вся система называется машиной конечного состояния. В такой системе время дискретно, так что нас интересует только состояние в дискретном, равномерно распределенном множестве раз, определяемом системными часами. Такая симуляция называется синхронной, поскольку изменения состояния происходят только при тактировании часов.

Немного более сложная модель дискретных событий позволяет время быть непрерывным и позволяет в любое время изменять состояние или события. Такая модель называется вызванной событием, а метод моделирования для нее асинхронен, потому что в любой данный момент времени различные части модели могут быть обновлены до разных имитируемых времен. Одной такой дискретной системой событий будет модель очередей, скажем, для пешеходного движения с использованием лифтов в новом здании. Государство состоит из числа людей, ожидающих каждого лифта, количества людей в каждом лифте и

кнопок, которые каждый человек хочет нажать. Время может быть непрерывным, а не дискретным, если мы решили моделировать время прибытия новых пассажиров в очереди как процесс Пуассона со случайным временем ожидания t между прибылями, распределенными экспоненциально: Prob ($t>x$) = $z \cdot e^{-z \cdot x}$. Руководящие правила предполагают, что пассажирские пункты выбираются случайным образом и соответственно перемещают людей через систему.

Также можно провести симуляцию, основанную на событиях, цифровой схемы, где разные компоненты схемы имеют разные задержки распространения. Это моделирование на логическом уровне является на один шаг более сложным, чем упомянутое выше симуляция функционального уровня, и используется для аппроксимации задержки, связанной с цифровой схемой.

Основным источником параллелизма в дискретном симуляции событий является разбиение системы на ее физические компоненты и отдельные процессоры имитируют каждый компонент. В случае цифровых цепей можно назначить подключенные подсхемы для разделения подпроцессоров. Если одна подсхема подключена к другой подсхемы, соответствующие процессоры должны взаимодействовать друг с другом. Есть много способов, которыми можно было бы представить назначение подсхем для процессоров (два показаны на рисунке 1.1), и нам нужно подумать о том, как выбрать хороший. В случае моделирования лифта можно представить отдельный процессор для каждого лифта или один процессор для каждого этажа. Опять же, нам нужен критерий для выбора конкретного разбиения и назначения.

Вот простая модель дискретной системы событий, которую мы можем использовать для описания разбиения. Мы моделируем систему как граф $G = (N, E)$ с N узлами и E ребрами соединяющими их. Каждый узел представляет физический компонент (например, подсхему), который должен быть назначен процессору, и каждый ребро представляет собой прямую зависимость между компонентами (например, провод). Кроме того, каждый узел n имеет вес W_n, представляющий работу, требуемую для имитации, и каждое ребро e имеет вес

W_e, представляющий, сколько информации должно быть передано вдоль него. Естественным набором целей для разбиения узлов на процессоры являются следующие факторы.

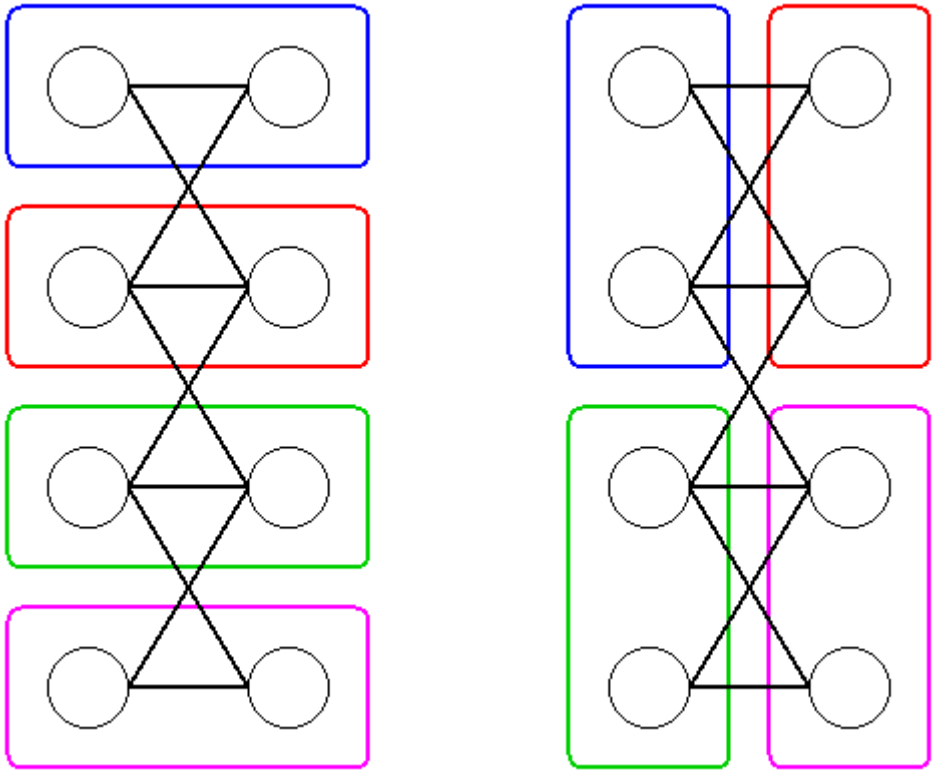

Рисунок 1.1 – Выборки графа с шестью и десятью краевыми переходами

Каждый процессор должен иметь примерно равный объем работы, т.е. Сумма весов W_n узлов, назначенных каждому процессору, должна быть примерно одинаковой для каждого процессора. Разделение работы в равной степени также называется балансировкой нагрузки.

Объем связи должен быть минимизирован, т.е. сумма весов W_e границ, пересекающих границы процессора, должна быть минимизирована.

Так, на рисунке 1.1 овалы рисуются вокруг узлов, которые должны быть назначены одному процессору. При каждом показанном разбиении каждому из четырех процессоров присваивается два узла. На схеме слева есть только 6 ребер, пересекающих границы процессора, тогда как схема справа пересекает десять, поэтому при условии, что каждое ребро имеет одинаковый вес, левое разделение требует меньше общения и, следовательно, превосходит его.

Синхронное и асинхронное моделирование различаются тем, как расписано обновление каждой подсистемы, то есть порядок, в котором процессоры могут обновлять часть своей системы. При синхронном моделировании планирование легко: все процессоры обновляют свои подсистемы на один временной шаг, а затем обмениваются информацией с другими процессорами. Это выражается в следующем простом цикле:

– рассылка;

– вычислить локально для обновления локальной подсистемы;

– барьер;

– обмен информацией с соседями;

– пока не будет выполнено условие.

Вместо барьера, который заставляет все процессоры ждать самого медленного, можно предположить, что каждый процессор будет ждать только нужные ему входы, а также отправлять свои выходы на процессоры, которые, как он знает, понадобятся им. Но если один процессор достаточно далеко опережает или отстает от других процессоров, это приведет к тому, что сообщения будут архивироваться где-нибудь в системе (проблема управления потоком), поэтому на самом деле потребуется какая-то более сложная синхронизация.

Планирование асинхронного моделирования еще сложнее, поскольку разные процессоры могут обновляться до разных времен, в зависимости от того, какие события произошли. Простейший вид асинхронного моделирования является консервативным, когда процессор может моделировать только до времени t, если все соответствующие события до времени t от всех других процессоров прибыли. В качестве примера рассмотрим моделирование комбинаторной логики, где события распространяются в одном направлении по цепи без обратной связи. В этом случае любой компонент схемы, может изменить свой выход, как только его входы изменились, потому что мы знаем, что входы могут меняться только один раз. Структура данных графика событий в Multipol библиотека распределенной библиотеки данных поддерживает такой

параллелизм (после того, как было выполнено разделение).

В качестве альтернативы ожиданию всех других событий до времени *t*, прежде чем приступить к времени *t*, процессор может смело предположить, что такое событие не поступит и все равно имитирует. Поскольку это предположение может быть неправильным, необходимо сохранить прежнюю информацию о состоянии, чтобы обеспечить резервное копирование в правильное состояние. Это предлагает второй источник параллелизма, называемый спекулятивным параллелизмом, который следует использовать, если консервативный подход приводит к слишком малому параллелизму.

Простой пример такого параллелизма, с которым большинство выпускников в конечном итоге становится знакомым, составляет квалификационный экзамен. Из-за отсутствия распределенного программного обеспечения для календаря каждый аспирант должен связаться с 4 преподавателями асинхронно и найти общее время встречи. Студент, как правило, соглашается на некоторое время с несколькими преподавателями, а позже обнаруживает, что последний член факультета не может этого сделать, требуя от других преподавателей скорректировать свои календари. Это может повториться. Другой пример возникает, когда центральная администрация пытается планировать классы на основе допущений о размерах классов.

Более интересный пример спекулятивного параллелизма происходит в некоторых суперскалярных компьютерных архитектурах, пытаясь быстрее выполнить код. Суперскалярные архитектуры пытаются автоматически выполнять инструкции одновременно, если между ними нет зависимостей, например одна инструкция, использующая регистр, вычисленный предыдущей инструкцией. Мы видели это на IBM RS6000 / 590, который может выполнять до 6 команд одновременно. Если условная ветвь встречается в программе, часто ее нельзя выполнить за эту точку, так как нельзя всегда предсказать, как она будет идти. Некоторые новые архитектуры будут пытаться угадать, в какую сторону будет идти ветвь, исходя из статистики прошлой производительности, и выполнять в любом случае спекулятивно, сохраняя при этом достаточное

состояние для резервного копирования. Например, в конце цикла хорошо догадаться, что ветка вернется к началу цикла. Другие архитектуры могут даже выполнять как возможные потоки команд, которые могут следовать за веткой, а затем отбросить ее.

Резервное копирование дорого по трем причинам: оно тратит время, затрачиваемое на вычисление ошибочной последовательности состояний системы, оно освобождает пространство, необходимое для сохранения достаточно старой информации о состоянии, чтобы обеспечить резервное копирование (не так важно для планирования циклов, но тем более в другие ситуации), и это может привести к простою процессоров, ожидающих резервного копирования и повторного компрометации другого процессора. Спекуляция является эффективной схемой распараллеливания, только если редко приходится создавать резервные копии и восстанавливать их.

Пример использования спекулятивного выполнения для моделирования параллельной схемы: «Параллельное моделирование времени на распределенном многопроцессоре памяти», Международная конференция по компьютерному проектированию, Санта-Клара, Калифорния, ноябрь 1993 года. Это приложение выполняет контроль и восстановление старого состояния. Было бы интересным проектом создать общий инструмент для поддержки спекулятивного параллелизма, учитывая только последовательные подпрограммы пользователя для имитации каждого компонента; такая система, называемая Timewarp, была построена Дэвидом Джефферсоном.

1.2.3 Моделирование систем частиц

В системе частиц конечное число частиц движется в пространстве с определенными законами движения и некоторыми видами взаимодействий. Время непрерывное, а также назначение программирования на отскок шаров, являются примерами систем частиц. В обоих случаях законы движения Ньютона в 2 измерениях лежат в основе моделирования; Также возможно трехмерное моделирование. Другие примеры включают:

– движение ионов в плазме, например, при моделировании звезд или

термоядерных реакторов; силы на частицах являются электростатическими, а законы Ньютона управляют движением;

– движение нейтронов в реакторе деления; законы Ньютона управляют движением наряду со случайными столкновениями нейтронов и ядер, которые могут делить и создавать больше нейтронов;

– движение атомов в молекуле, как в вычислительной химии; силы на атомах моделируются как электростатические или, возможно, исходя из довольно жестких химических связей с определенными степенями свободы; законы Ньютона регулируют движение;

– движение звезд в космосе, изученное астрофизиками, рассматривающим валовую структуру Вселенной; законы Ньютона и гравитация определяют движение;

– движение автомобилей на автостраде, планирование перевозок; силы на автомобилях зависят от законов Ньютона, а также от более или менее сложных моделей двигателя и того, как водитель реагирует на движения близлежащих автомобилей.

Сила, которая перемещает каждую частицу, обычно может быть разбита на 3 компонента:

force = external_force + nearby_force + far_field_force

Внешняя сила в акулах и рыбах является текущей. Его можно оценивать независимо для каждой частицы без знания других частиц и, таким образом, смущающе параллельным образом. Для эффективного распараллеливания требуется, чтобы каждый процессор имел приблизительно равное количество частиц.

Nearby_force относится к силам, зависящих только от других частиц очень близко. При назначении прыгающего шара сила столкновений попадает в эту категорию. На автостраде поведение автомобиля зависит (надеется) от поведения соседних водителей, особенно с тем, что было впереди. В молекуле самые сильные силы на атоме могут быть от других атомов, к которым он имеет прямую химическую связь. Для эффективного распараллеливания требуется

локальность, то есть, вероятно, что ближайшие соседи частицы находятся на том же процессоре, что и частица.

Далеко силовое поле не зависит от всех других частиц в системе, независимо от того, как далеко. Эти силы в общем случае имеют вид

far_field_force на частице i = sum_ {j = от 1 до n, за исключением i} force (i, j),

где force(i, j) – сила на частице i из-за частицы j. Примером является электростатическое усилие. Взаимодействие вихрей в потоке жидкости является другим. На первый взгляд, эти силы, по-видимому, требуют глобальной коммуникации на каждом шагу, нанося ущерб любой попытке сохранить местность и минимизировать связь. Но для этого есть множество умных алгоритмов.

Иногда сила может быть классифицирована как ближайшая или дальняя. Например, сила Ван-дер-Ваальса между атомами является слабым притяжением, когда атомы имеют по меньшей мере несколько атомных диаметров друг от друга. Он уменьшается, как $1/r^6$, где r – расстояние между атомами. Эта функция уменьшается так быстро, как растет r, что ее можно просто игнорировать, когда r достаточно велико, что делает ее близкой_force и поэтому меньше работы для вычисления. Гравитационные и электростатические силы подчиняются закону обратного квадрата и, следовательно, уменьшаются с ростом r/r^2. Функция $1/r^2$ убывает гораздо медленнее, чем $1/r^6$, и не может быть проигнорирована ни для какого r, если это единственная сила, имитируемая.

Есть два очевидных способа попытаться разделить частицы между процессорами, чтобы вычислить внешние_force и близлежащие_force. Наилучшим способом разделения для распараллеливания вычисления внешних сил является разбиение по числу частиц или просто назначение равного количества частиц каждому процессору по числу частиц независимо от того, как их физические местоположения могут измениться. Это гарантирует идеальный баланс нагрузки, но не подходит для близких_forces, потому что ни одна местность не гарантируется. Лучшим способом разделения для близлежащих сил

является разделение физического пространства, в котором находятся частицы, причем каждый процессор отвечает за частицы, которые находятся в той части пространства, в которой он принадлежит.

При назначении прыгающего шара разбиение физического пространства требует связи только частиц вблизи границ разделов, поскольку частицы (шары) взаимодействуют только с соседними частицами. Если частицы распределены равномерно, это означает, что существует эффект объемного объема, который, как правило, делает объем связи относительно небольшим по сравнению с количеством вычислений для большого количества частиц, что дает хорошую эффективность. Например, предположим, что мы разбиваем 3D-пространство на единичные кубы, что конкретный единичный куб содержит n равномерно распределенных частиц и что только частицы на расстоянии $r << 1$ поверхности куба должны быть переданы соседним процессорам. Тогда можно ожидать связи $c = 6rn$ частиц, но локальная работа над $n >> c$ локальными частицами. Другими словами, локальные вычисления выполняются со всеми частицами в объеме куба, но общаются только с гораздо меньшими частицами вблизи поверхности.

К сожалению, если частицы распределены неравномерно, и мы используем фиксированное разбиение физического пространства, скажем, на квадраты равного размера или кубы, частицы могут перемещаться так, чтобы собирать в небольшом количестве процессоров, разрушая баланс нагрузки. Это часто бывает в вычислительной гидродинамике (CFD) с использованием вихревого метода, где турбулентность и другие движения жидкости приводят к сгущению вихрей (частиц). Он также возникает в астрофизике, где звезды склонны сжиматься в галактики и другие крупные космические структуры, а не быть однородными (действительно, именно это поведение астрофизиков хочет изучать). Это приводит нас к динамическим схемам разбиения на основе четырех деревьев в 2D и октальных деревьях в 3D. Ниже приведен пример, где мы рекурсивно разбиваем плоскость на меньшие поля, пока каждый ящик не будет содержать равное количество частиц. Это разложение должно периодически обновляться, когда частицы меняют свои позиции.

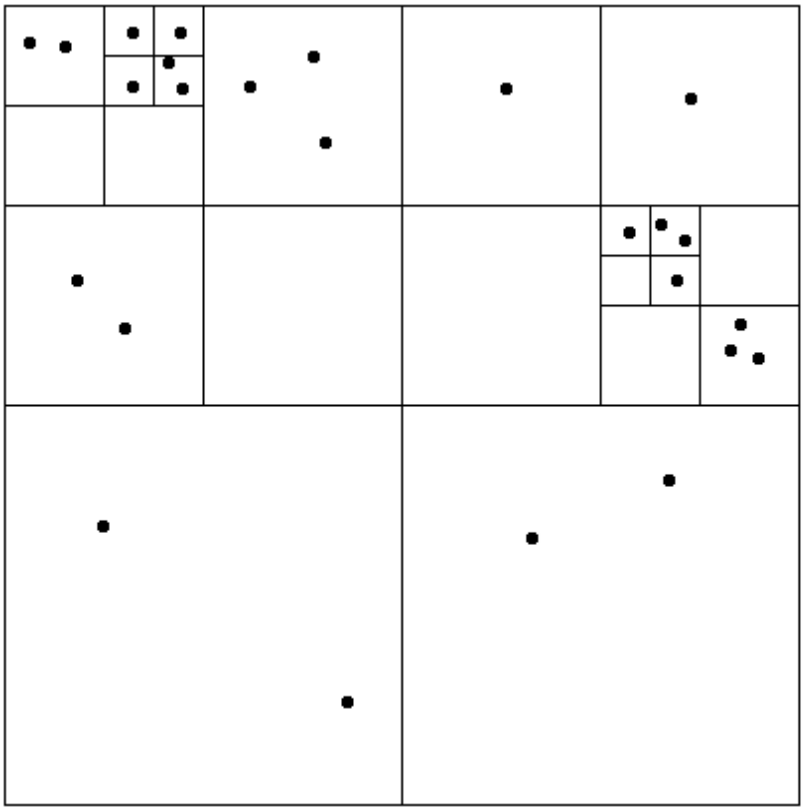

Рисунок 1.2 – Разложение квадратного дерева, где ни один квадрат не содержит более трех частиц

Теперь рассмотрим распараллеливание силы far_field. Поскольку эти силы настолько распространены и важны, для уменьшения их алгоритмической сложности был разработан ряд умных алгоритмов, а не только параллельного алгоритма, но и последовательного алгоритма. Очевидный последовательный алгоритм, описанный выше, принимает $O(n^2)$ для n частиц. Если ничего особого не известно о силовой функции силы (i, j), ничего больше не может быть сделано. Но для таких сил, как гравитация, электростатика и другие, в силовой функции существует большая структура, позволяющая уменьшить сложность от $O(n^2)$ до $O(n \log n)$ или даже $O(n)$, в зависимости от о точности приближения, которое мы желаем. Другими словами, если мы готовы торговать с точностью до скорости, мы можем значительно уменьшить сложность. На практике, можно просто сделать ошибку меньшей, чем точность машины, если хотите, поэтому симуляция может быть выполнена так же точно, как с прямым методом. Для некоторых симуляций недостаточно точность, чем полная точность машины, и

можно идти еще быстрее.

Мы рассмотрим два вида быстрых методов вычисления гравитационных (или электростатических) сил на n частицах. Первый – метод частиц–меш. Этот метод накладывает правильную сетку на множество частиц и создает аппроксимацию исходной задачи, перемещая частицы в ближайшие точки сетки, как показано ниже (более точные интерполяционные методы могут быть использованы на практике). Так как эта приблизительная задача лежит на регулярной сетке, для ее быстрого решения доступно несколько умных решений для разделения и покорения, таких как Multigrid и FFT. Точность этого приближения зависит от того, насколько тонкой сеткой мы выберем и насколько равномерно распределены частицы.

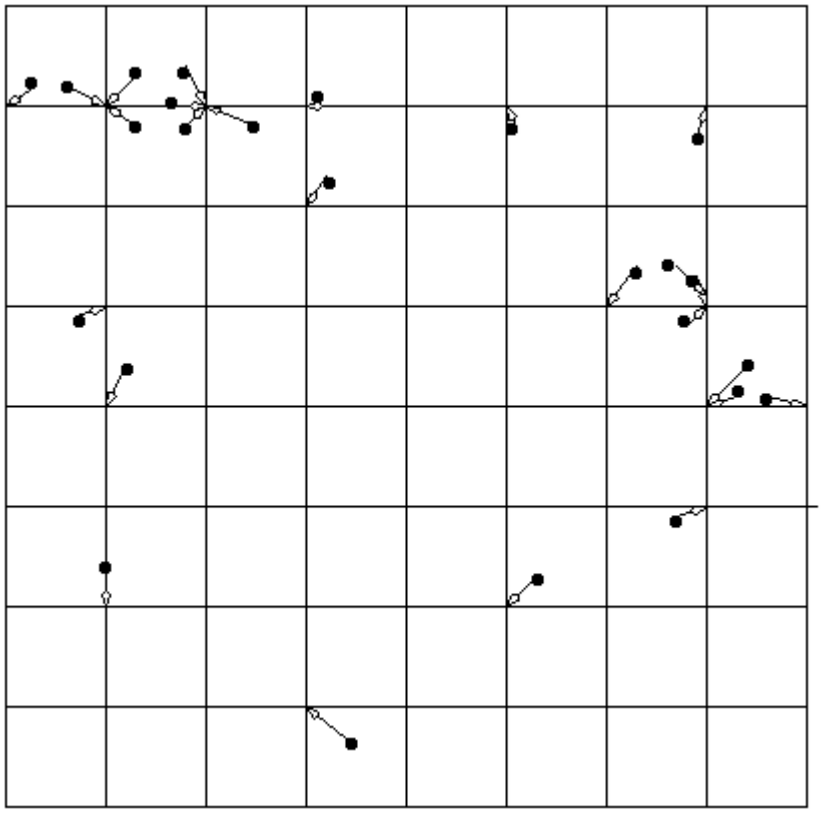

Рисунок 1.3 – Перемещение частиц в точку сетки методом частиц

Второй тип метода также использует разделение и завоевание, но на исходных частицах, без искусственной сетки. Двумя алгоритмами в этом классе являются алгоритм Барнса-Хат и быстрый мультипольный метод; они используют аналогичное разделение и разделение пространства с квадратным

деревом (в 2D) или октовым деревом (в 3D), как описано выше. Основная идея состоит в том, что гравитационная сила большого скопления частиц при условии, что она достаточно далеко, может быть аппроксимирована точечной массой в центре массы частиц. Это позволяет нам «сжимать» информацию в удаленном скоплении частиц, и они одновременно вычисляют и сообщают об этом быстрее. Квадратное дерево (или oct-tree) используется для определения того, какие частицы могут быть сгруппированы и сжаты таким образом.

Моделирование систем сосредоточенных переменных в зависимости от непрерывных параметров. Стандартным примером такого моделирования является схема. Рассмотрим схему как график с проводами как ребра (также называемые ветвями) и узлы, в которых подключены два или более провода. Каждое ребро содержит один резистор, конденсатор, индуктор или источник напряжения. Состояние системы представлено напряжениями узла $v_n(t)$, ветвями токов $i_b(t)$ и напряжениями ветвления $v_e(t)$ во всех частях схемы. Другими словами, существует конечный («сосредоточенный») набор переменных состояния $v_n(t)$, $i_b(t)$ и $v_e(t)$, непрерывно зависящих от времени параметра. (Это фактически избыточное множество переменных, так как $v_e(t)$ легко может быть выражено через $v_n(t)$, но мы сохраняем все три для удобства изложения.) Управляющие уравнения являются хорошо известным течением Кирхгофа и законы напряженности Кирхгофа, закон Ома ($v=i \cdot R$) и законы, определяющие емкость ($i=C \cdot \mathrm{dv/dt}$) и индуктивность ($v=L \cdot \mathrm{di/dt}$). Все эти уравнения могут быть записаны как одна большая система уравнений

[0 *A* 0] [*v_n*] [0] – Текущий закон Кирхгофа

[*A* '0 -*I*] * [*i_b*] = [*S*] – Закон напряжения Кирхгофа

[0 *R* -*I*] [*v_b*] [0] – Закон Ома

[0 -*IC* * *d* / *dt*] [0] – емкость

[0 *L* * d / dt *I*] [0] – индуктивность

Матрицы *A*, *R*, *C*, *L* и *S* являются разреженными матрицами, представляющими связность схемы, а в интересных схемах они также являются большими матрицами. Например, *A* имеет одну строку для каждого узла и один

столбец для каждой ветви с $A(n, b)= \pm 1$, если узел n является одной из конечных точек ветви b и нуль в противном случае (каждой ветви задается произвольная так что один конечный узел равен +1, а один – -1). Аналогично, R имеет одну строку и один столбец для каждой ветви, причем R (b, b) равен сопротивлению ветви b. C – аналогичная большая диагональная матрица емкостей и L индуктивностей. S – вектор источников напряжения.

Это система одновременных алгебраических уравнений (например, $A \cdot v_n = v\_b$) с обычными дифференциальными уравнениями (ODE) (например, $i\_b = C \cdot dv\_b$ / dt). Уравнения могут быть нелинейными, если сопротивления R, емкости C или индуктивности L зависят от значений v или i, протекающих через них (например, мы могли бы моделировать диоды). Схема имитатора обновляет значения V_n (T), i_b (m) и v_e (T) от временного шага т до времени шага T + dt с использованием этих основных уравнений.

(Наша простая схема может обрабатывать только пассивные схемы, поскольку каждый компонент – резисторы, конденсаторы и т. д. Имеет всего 2 соединительных провода. Активные схемы, которые включают транзисторы, требуют компонентов с тремя проводами (например, дренаж, источник и затвор для транзисторов)).

Другим примером является структурный анализ в области гражданского строительства. В простейшем примере переменными являются перемещения отдельных точек или узлов в здании или другой структуре из их положений равновесия. Структура может быть смоделирована путем предоставления этим точкам массы элементов конструкции (например, стен и балок) и соединения их с жесткими элементами или, возможно, жесткими пружинами, которые моделируют гибкость в конструкционных элементах. Управляющие уравнения – это законы Ньютона и закон Гука, в котором говорится, что сила, оказываемая пружиной, пропорциональна тому, насколько она растянута или сжата. В статическом моделировании, один из них оказывает внешние силы на определенные узлы и запрашивает смещение других узлов; в частности, можно спросить, превышена ли прочность на разрыв какого-либо элемента

конструкции. В динамическом моделировании некоторые узлы будут подвергать непрерывным силам (например, от симулированного землетрясения) и спросить, как структура движется во времени. Статическая задача включает в себя решение системы алгебраических уравнений, а динамическая задача – система дифференциальных уравнений. Как и в случае схем, матрицы большие и разреженные, где структура разреженности (какие записи отличны от нуля и которые равны нулю) в зависимости от того, какие структурные элементы связаны с другими членами.

Третий пример – химическая кинетика или вычисление концентраций химических веществ, подвергающихся реакции. Предположим, что у нас есть три химиката O (атомный кислород), O3 (озон) и O2 (нормальный молекулярный кислород). Очень простая модель для фотохимического смога в таких городах, как Лос-Анджелес, состоит из 4 реакций

O + O2 -> O3,

O + O3 -> 2O2,

O2 -> 2O,

O3 -> 0 + O2.

Кроме того, каждая реакция *i* имеет константу скорости, в котором говорится, как происходит «быстрая» реакция. Учитывая эту информацию, легко получить систему ОДУ для концентрации каждого химического вещества. Обозначим [O], [O2] и [O3] концентрации (в молях на литр, атомы на кубический дюйм или некоторые аналогичные единицы), которые мы хотим вычислить как функции времени. Мы предполагаем, что реакции происходят в «хорошо перемешиваемом контейнере», поэтому все химические вещества равномерно распределены. В этом случае можно сконцентрировать концентрацию [O] в O как вероятность нахождения атома O в фиксированном крошечном объеме. Для реакции O и O2 оба должны одновременно находиться в одном и том же крошечном объеме. Поскольку O и O2 хорошо перемешиваются, то есть независимо распределяются, вероятность того, что один атом или молекула каждого из них одновременно присутствует, является продуктом вероятностей

[O] * [O2]. Учитывая, что оба присутствуют, скорость реакции k1 измеряет вероятность того, что реакция действительно произойдет. Если это происходит, создается одна молекула O3, и каждая из O и O2 уничтожается. Это приводит к следующим ОДУ, которые моделируют только первую реакцию выше:

d [O3] / dt = + k1 * [O] * [O2],

d [O] / dt = -k1 * [O] * [O2],

d [O2] / dt = -k1 * [O] * [O2].

Чтобы включить вторую реакцию в ОДУ, мы рассуждаем аналогичным образом. Таким образом, мы получаем дополнительный член -k2 * [O] * [O3] для уравнений для d [O3] / dt и d [O] / dt, поскольку один атом / молекула каждого из O и O3 расходуется, и + 2 * k2 * [O] * [O3] для уравнения для d [O2] / dt, так как создаются две молекулы O2. Это дает следующие ОДУ для первых двух реакций:

d [O3] / dt = + k1 * [O] * [O2] - k2 * [O] * [O3],

d [O] / dt = -k1 * [O] * [O2] - k2 * [O] * [O3],

d [O2] / dt = -k1 * [O] * [O2] + 2 * k2 * [O] * [O3].

Обработка реакций 3 и 4 аналогичным образом дает полный набор ОДУ для всех четырех реакций:

d [O3] / dt = + k1 * [O] * [O2] - k2 * [O] * [O3] - k4 * [O3],

d [O] / dt = -k1 * [O] * [O2] - k2 * [O] * [O3] + 2 * k3 * [O2] + k4 * [O3],

d [O2] / dt = -k1 * [O] * [O2] + 2 * k2 * [O] * [O3] - k3 * [O2] + k4 * [O3].

Самая полная модель смога в Лос-Анджелесе включает 92 химических вида, участвующих в 218 реакциях. Это представлено как разреженная матрица с одной строкой и одним столбцом для каждого химического вещества и отличная от нуля запись в (i, j), если химические реактивы i и j реагируют.

Теперь мы обсудим источники параллелизма в этих проблемах. Несмотря на то, что мы решаем дифференциальные уравнения и нелинейные системы, вычислительные узкие места сводятся к линейной алгебре с разреженными матрицами. Простейшим примером является решение системы линейных алгебраических уравнений $A \cdot x = b$. Второй пример – решение нелинейной системы уравнений, которую мы пишем как $f(x)=0$. Наиболее распространенным

методом его решения является метод Ньютона, который мы пишем как

$x\,(i+1) = x\,(i) - \mathrm{inv}(\mathrm{d}f\,(x\,(i))/\mathrm{d}x))*f(x\,(i))$,

В этом уравнении $\mathrm{d}f\ (x(i))/\mathrm{d}x$ – матрица якобиана первых частных производных от f, вычисленная по $x\,(i)$. Если $x\,(i)$ является n-мерным вектором, то $\mathrm{d}f/\mathrm{d}x$ является n-матрицей, а на один шаг метода Ньютона требуется решение линейной системы уравнений

$\mathrm{d}f\,(x\,(i))/\mathrm{d}x*(x\,(i+1) - x\,(i)) = f\,(x\,(i))$

Теперь обратимся к решению ОДУ. Для простоты рассмотрим простую модельную задачу $\mathrm{d}r/\mathrm{d}t = A \cdot x(t)$, где A – матрица, которая может зависеть от времени. Для того чтобы понять вычислительные особенности, мы рассмотрим два простейших методы решения: метод Эйлера и обратную метода Эйлера. Часто используются более сложные методы, но они достаточно хороши, чтобы проиллюстрировать вычислительные узкие места. В обоих случаях мы используем фиксированный временной шаг h, другими словами, мы только пытаемся аппроксимировать решение при $t = 0, h, 2 \cdot h, \ldots, i \cdot h$. Более сложные методы будут использовать шаг с переменным временем, делая большие шаги, когда решение будет плавным и настолько простым в вычислении, и небольшие шаги, где он быстро изменяется, и поэтому его сложно вычислить.

Метод Эйлера аппроксимирует производную $\mathrm{d}x(t)/\mathrm{d}t$ разделенной разностью

$x(t+h) - x(t)\mathrm{d}x\,(t) / \mathrm{d}t$

Если мы возьмем предел, когда h обращается в нуль, правая часть приближается к производной $\mathrm{d}x\,(t)/\mathrm{d}t$, поэтому это разумное приближение, если h достаточно мало. Теперь заменим это приближение для $\mathrm{d}x(t)/\mathrm{d}t$ в ODE, чтобы получить

$x(t + h) - x\,(t) = A \cdot x(t)$.

Предполагая, что мы знаем приближенное решение для $x(t)$, мы можем решить для $x(t + h)$, чтобы получить

$x(t + h) = \mathrm{x}(t) + h * A \cdot x(t)$

Это метод Эйлера и дает простую формулу для вычисления

приближенного решения в момент времени $t+h$, учитывая решение в момент времени t. Используя это, мы можем шагнуть во времени, пока нам нравится. Вычислительным узким местом является разреженное умножение матрицы-вектора $A \cdot x$.

Иногда метод Эйлера требует очень малого шага времени h, чтобы получить разумную точность, даже когда решение является гладким. Это происходит, например, если ODE обладает свойством, называемым жесткостью, что означает, что вектор решения $x(t)$ имеет некоторые компоненты, которые быстро меняются, а другие, которые медленно меняются, и нас интересуют только медленно меняющиеся компоненты. Жесткие ОДУ обычно встречаются в химической кинетике и некоторых схемных симуляциях. Для этих проблем выбранным методом является Backward Euler:

$x(t + h) - x(t) = A \cdot x(t + h)$

Заметим, что мы изменили правую часть от $A \cdot x(t)$ до $A \cdot x(t+h)$. Решение для $x(t + h)$ дает

$(I - h * A) * x(t + h) = x(t)$

Линейная система уравнений для решения при $x(t+h)$. Это вычислительное узкое место.

Таким образом

– разреженное умножение матрицы-вектора и

– решая разреженную систему линейных уравнений

это вычислительные узкие места, которые нам нужно распараллелить. Здесь мы довольствуемся тем, что хороший параллельный алгоритм для разреженного умножения матрицы-вектора зависит от решения проблемы разбиения графа, рассмотренной выше.

Пусть задача распараллеливать, умножая $y = A \cdot x$, где

Нам нужно решить

– какие процессоры будут хранить данные y (i), $x(i)$ и $A(i, j)$ и

– какие процессоры будут выполнять вычисления $y(i) = \text{sum\_}\{i = 1\} A(i, j) \cdot x(j)$, где мы опускаем члены, где $A(i, j) = 0$,

таким образом, чтобы

– балансировать нагрузку,

– сбалансировать хранение и

– минимизировать связь.

Балансировка нагрузки и минимизация связи были поставлены ранее; средство балансировки хранения, каждое из которых хранит примерно равную долю от общего количества данных; это важно, чтобы иметь возможность масштабировать до больших проблем, которые не могут уместиться на одном процессоре. Вот один из способов преобразования этой проблемы в задачу разбиения графа взвешенного графа $G = (N, E, W)$.

Пусть узлы $N = \{1,2, ..., n\}$ графа G представляют собой строки из A, x и y. Предположим, что мы разбиваем узлы N на объединение $N1$ на Np, где p – количество процессоров. Назначение узлов в Ni процессору i будет означать, что

– процессор i сохранит $y(k)$, $x(k)$ и $A(k, :)$ для каждого k из Ni (здесь $A(k, :)$ k – я строка A, где нам нужно хранить только ненулевые); а также

– процессор i вычислит $y(k)$ для каждого k в Ni, обмениваясь данными с другими процессорами, чтобы получить необходимые записи x, не сохраненные локально.

Это пример правила вычисления владельца, где мы позволяем процессору, владеющему $y(k)$ выполнять все вычисления, необходимые для его вычисления (можно было бы представить и другие возможности).

Края E и веса W графа G определяются следующим образом. Существует (неориентированный) ребро (i, j) в E всякий раз, когда $A(i, j)$ или $A(j, i)$ отличен от нуля, а i не равен j. Вес узла i равен числу ненулевых $A(k, j)$, хранящихся в узле i. Например, на рисунке 1.4 показана конкретная 8×8 разреженная симметричная матрица и конкретное разбиение на 4 процессора (цветной синий, красный, зеленый и пурпурный):

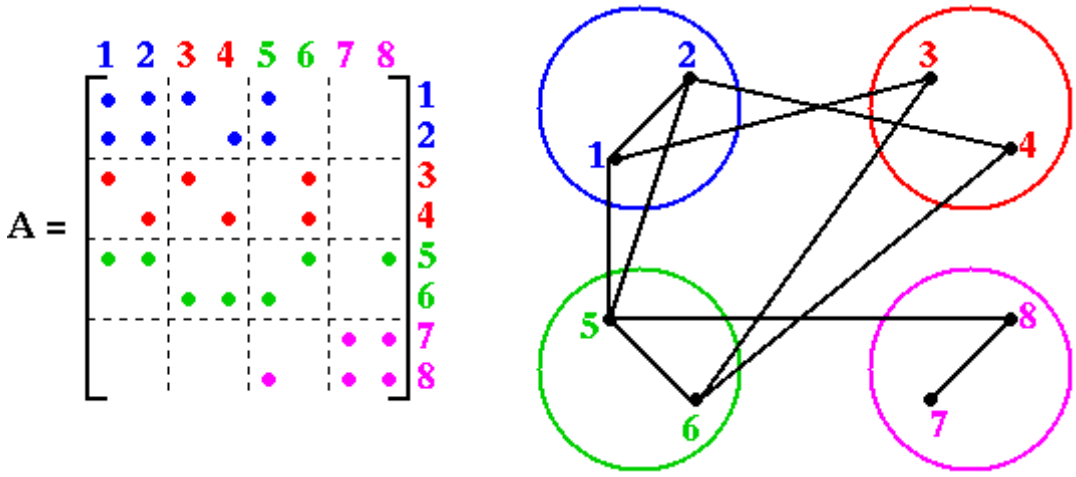

Рисунок 1.4 – Разбиение разреженной симметричной матрицы

Используя этот механизм, мы можем сформулировать задачу разбиения графа как выбор разбиения *N* на объединение *N1-Np*, такое, что

Сумма весов узлов в *Ni* равна для всех *i*. Это означает, что рабочая нагрузка сбалансирована, поскольку сумма весов узлов пропорциональна количеству операций с плавающей запятой, выполняемых на процессоре. Это также означает, что хранилище сбалансировано, потому что сумма весов узлов равна количеству ненулевых записей из *A*, сохраненных.

Количество границ, пересекающих границы процессора, минимизируется. Это минимизирует объем данных (количество удаленных хранимых *x*(*j*)).

Последней проблемой линейной алгебры, возникающей при решении ОДУ, является следующая. Предположим, что в d*x* (*t*)/d*t* = *A*·*x* (*t*) матрица *A* постоянна, не зависит от *x* или *t*. В этом простом, но важном случае мы можем «догадаться» о решении $x(t)=e^{z \cdot t} \cdot x0$, где *z* – определяемый константный скаляр, а *x*0 – ненулевой вектор константы, который будет определен.

d*x*(*t*)/d*t* = *z* * exp(*z* * *t*) * *x*0 = *A* * exp (*z* * *t*) * *x*0

Переменная exp (*z* * *t*), которая всегда отлична от нуля, дает

z·*x*0 = *A*·*x*0

которая представляет собой задачу на собственные значения, мы можем решить для собственного значения *z* и собственного вектора *x*0. Позже мы обсудим методы решения собственных проблем.

Например, мы можем использовать этот анализ для моделирования безопасности зданий в землетрясениях. В этом случае мы выбираем ОДУ для описания движения здания, как описано выше. Тогда (наименьшие) собственные значения A являются «резонансными частотами» здания. Если эти собственные значения близки к частотам, в которых вибрируют землетрясения, то существует опасность того, что здание будет резонировать с землетрясением и будет более склонным к краху.

Все описанные выше методы ODE продвигают решение $x(t)$ с временного шага t на временной шаг $t + h$. Другими словами, во времени нет параллелизма , который позволил бы нам вычислить решение в далеком будущем времени T без вычисления решения во все промежуточные времена последовательным образом. (В самом частном случае, линейного ОДУ $\mathrm{d}x/\mathrm{d}t = A(t) \cdot x(t) + b(t)$, фактически можно использовать параллельный префикс для его решения во времени $O(\log (T/h))$ вместо $O(T/h)$).

Существует совершенно другой метод, который не требует последовательного шага по времени, и поэтому предлагает возможность параллелизма во времени, а также в простой шаг. Этот метод называется релаксацией формы волны, Он начинается с предположения для решения на интересующий весь промежуток времени, например $[0, T]$, и итеративно улучшает его (используя технику, аналогичную итерации Пикара), сходящуюся на каждом временном шаге одновременно. Для некоторых специальных ODE он сходится достаточно быстро, чтобы конкурировать с такими методами, как Euler и Backward Euler. Но методы, подобные Euler и Backward Euler, используются гораздо чаще.

1.3 Последовательные алгоритмы сортировки

1.3.1 Обор алгоритмов

Selection sort (сортировка выбором) – смысл алгоритма состоит в проходе по массиву от начала до конца в поиске наименьшего элемента массива и переносе его в начало. Сложность такого алгоритма равна $O(n^2)$.

Bubble sort (сортировка пузырьком) – алгоритм меняет местами два рядом стоящих элемента, в случае если первый элемент массива больше второго. Это повторяется до тех пор, пока алгоритм не поменяет местами все неотсортированные элементы. Сложность этого алгоритма сортировки равна $O(n^2)$.

Insertion sort (сортировка вставками) – алгоритм сортирует массив по мере прохождения по его элементам. На каждой итерации каждый элемент сравнивается с каждым элементом в уже отсортированной части массива, таким образом находя «свое место», далее элемент вставляется на свою позицию. Так повторяется до тех пор, пока алгоритм не пройдет по массиву. В результате получим отсортированный массив. Сложность данного алгоритма равна $O(n^2)$.

Quick sort (быстрая сортировка) – основная идея алгоритма заключается в разделении массива на два подмассива, центральным считается элемент, который находится в середине массива данных. В результате работы алгоритма элементы, значение которых меньшие чем средний будут перемещены влево, а большие вправо. Такое же действие будет происходить рекурсивно и с подмассивами, они будут разделяться на еще два подмассива до тех пор, пока не будет чего разделать (останется один элемент). В результате получаем отсортированный массив. Сложность алгоритма зависит от исходных данных и в лучшем случае будет равняться $O(n \cdot 2\log_2 n)$. В худшем случае $O(n^2)$. Существует также среднее значение, это $O(n \cdot \log_2 n)$.

Comb sort (сортировка расческой) – смысл работы алгоритма крайне схож с сортировкой обменом, но основным отличием здесь является то, что сопоставляются не два соседних элемента, а элементы на промежутке, к примеру, в десять элементов. Это обеспечивает от избавления мелких значений в конце, тем самым дает возможность ускорить сортировку в крупных массивах. Первая итерация совершается с рассчитанным по формуле шагом (размер массива) / (фактор уменьшения), где фактор уменьшения приблизительно равен 1,247330950103979. Вторая и следующие итерации будут проходить с шагом (текущий шаг) / (фактор уменьшения) и будут происходить до тех пор, пока шаг

не будет равен единице. Практически в любом случае сложность алгоритма равняется $O(n \cdot \log_2 n)$.

1.3.2 Результаты работы алгоритмов

Для получения лучших результатов все упомянутые ранее алгоритмы будут сортировать массив из двухсот целочисленных элементов. Система, на которой тестируются алгоритмы имеет следующие характеристики: процессор Intel core i5-2300 4x2.4 GHz, оперативная память 8 GB, операционная система Windows 7 x64 build

Таблица 1.1 – Время сортировки массива целочисленных элементов

Алгоритм сортировки	10 элементов		50 элементов		200 элементов		1000 элементов	
	время	память	время	память	время	память	время	память
Selection sort	13 ms	510 K	37 ms	637 K	118 ms	854 K	550 ms	936 K
Bubble sort	11 ms	524 K	37 ms	629 K	116 ms	863 K	564 ms	932 K
Insertion sort	12 ms	512 K	38 ms	641 K	116 ms	849 K	556 ms	928 K
Quick sort	22 ms	389 K	102 ms	454 K	412 ms	612 K	732 ms	730 K
Comb sort	12 ms	505 K	37 ms	632 K	117 ms	854 K	560 ms	936 K

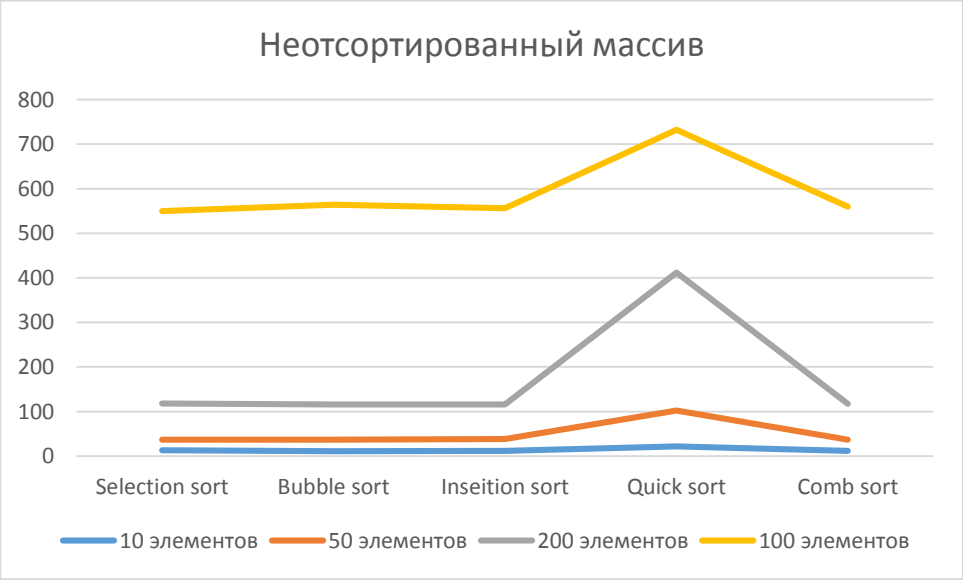

Рисунок 1.5 – Время сортировки массива целочисленных элементов

В результате исследования и полученных в результате данных, для сортировки неупорядоченного массива, самым оптимальным из перечисленных алгоритмов является быстрая сортировка, что легко видеть, если посмотреть на рисунок 1.5. Несмотря на то, что алгоритм выполняется дольше, он потребляет меньше памяти, что очень важно в крупных проектах. Другие алгоритмы, такие как сортировка, выбором, обменом, а так же вставкой могут лучше подойти для научных целей.

2 ПРОБЛЕМЫ ДЕКОМПОЗИЦИИ РАСЧЁТНЫХ СЕТОК И ЗАДАЧИ СОРТИРОВКИ

2.1 Простые алгоритмы разбиения сеток

Наиболее простыми алгоритмами разбиения сеток являются алгоритмы, учитывающие только номера вершин. К таким алгоритмам относятся линейное распределение вершин (linear method), рассеивание (scattered method) и случайное распределение (random method) [21].

При линейном распределении непрерывный диапазон номеров вершин разбивается на интервалы, вершины из одного интервала попадают в один домен. При рассеивании номер вершины, взятый по модулю числа доменов, определяет, в какой домен попадет вершина. То есть сначала в каждый домен по очереди распределяется по одной вершине, потом по две и т.д. При случайном распределении домен вершины выбирается случайным образом среди доменов, в которых вершин меньше, чем должно быть при равномерном распределении.

Достоинством данных алгоритмов является быстрота выполнения. Поэтому они используются для получения предварительного разбиения сетки, когда декомпозиция сетки выполняется на распределенной вычислительной системе, и перед вычислением самой декомпозиции сетку нужно неким образом распределить между процессорами. Однако разбиения, получаемые данными методами, не учитывают геометрию сетки, поэтому в качестве предварительного разбиения они подходят только для тех алгоритмов, результат работы которых не зависит от качества начального распределения вершин по процессорам.

2.2 Метод сдваивания

Параллельный алгоритм сортировки данных на принципе сдваивания включает в себя два этапа:

1) на каждом из процессоров с помощью алгоритма dhsort сортируется фрагмент массива A_i длинной $|A_i| = \frac{n}{p}$. При этом, для простоты анализа, положим, что $n=vp$, где v – целое неотрицательное число;

2) элементы отсортированных на первом этапе фрагментов A_i упорядочиваются с помощью слияния, причем последовательность слияний определяется методом сдваивания (рисунок 1.2).

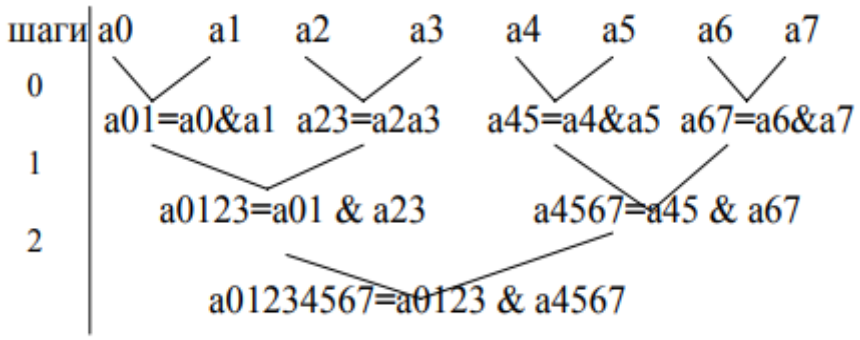

Рисунок 1.2 – Порядок выполнения слияния фрагментов массива методом сдваивания на восьми процессорах

Порядок выполнения слияния частей массива методом сдваивания на восьми процессорах.

На нулевом шаге (j=0) каждый вычислитель с номером $2i$, (i=0,…, p-1) принимает от вычислителя с номером $2i$+1 отсортированный фрагмент массива A_{2i+1} длиной n/p, что требует $k_2 \frac{n}{p}$ действий, после чего процессоры с номерами $2i$ выполняют соединение пар фрагментов, что требует от каждого процессора выполнения $2k_3 \frac{n}{p}$ действий: $A_{2i} \otimes A_{2i+1} \to A_{2i}$.

На шаге j каждый процессор с номером $2^j \cdot 2i$ принимает от процессора с номером $2^j(2i+1)$ отсортированный фрагмент массива длиной $2^j \frac{n}{p}$, что требует от каждого активного процессора $k_2 \frac{n}{p} \cdot 2^j$ действий. Затем процессор с номером $2^j \cdot 2i$ исполнение слияние двух фрагментов $A_{2^j 2i} \otimes A_{2^j(2i+1)} \to A'_{2^j 2i}$, что требует выполнения $k_3 \cdot \frac{n}{p} \cdot 2 \cdot 2^j$ итераций на каждом работающем процессоре. Общее время выполнения сортировки в таком случае:

$$T(n,\ p) = \frac{n}{p}\left(k_1 \log_2 \frac{n}{p} + k_2 (p-1) + 2k_3 (p-1) \right). \quad (1)$$

Заметим, что $k_1 = k_3$, из-за того, что оба эти коэффициента определяют количество действий выполняемых одним и тем же алгоритмом слияния упорядоченных массивов. Таким образом, полученное ускорение:

$$S(n,p) = \frac{T(n,1)}{T(n,p)} = \frac{k_1 n \log_2 n}{\frac{n}{p}\left[k_1 \left(\log_2 \frac{n}{p} + 2_3 (p-1) \right) + k_2 (p-1) \right]} =$$

$$= \frac{p}{\left(\log_2 n - \log_2 p + 2(p-1) + \frac{k_2}{k_1}(p-1) \right) \log_n 2} S(n,p) = \quad (2)$$

$$= \frac{p}{1 + \left(2(p-1) - \log_2 p + \frac{k_2}{k_1}(p-1) \right) \log_n 2}$$

Определим предполагаемое ускорение, достижимое на 4 и на 32 процессорах при $n = 10^9$.

$$S(10^9, 4) \approx \frac{4}{1{,}13 + \frac{1}{30}\frac{k_2}{k_1}} < 3{,}5;$$

$$S(10^9, 32) = \frac{32}{1 + \frac{1}{30}\left(56 + 31\frac{k_2}{k_1} \right)} \approx \frac{32}{3 + \frac{k_2}{k_1}} < 10.$$

При четырех вычислителях, которые имеют доступ к памяти (в таком случае затраты ресурсов на перенос массивов от процессора к процессору можно считать равными нулю: $k_2 = 0$), ускорение не превысит 3,5. При использовании 32 процессоров на общей памяти ускорение не превысит 10. Поскольку в системах с распределенной памятью $k_2 >> k_1$, допускаем, что ускорение будет небольшим и не будет иметь значимость, а значит, нет смысла применять системы с распределенной памятью с количеством вычислителей большим нескольких штук. Кроме того, представленный алгоритм не дает возможности обрабатывать массивы, размер которых превышает оперативную память одного

процессорного узла, так как заключительное слияние двух половин всего массива выполняется на одном процессоре. Таким образом, метод сдваивания достаточно прост в реализации, но эффективен только при небольшом числе процессоров, объединенных общей памятью [14].

2.3 Параллельные алгоритмы сортировки

Для того чтобы разбить расчетную сетку на домены, для начала нужно отсортировать массив данных. Сортировка в большинстве случаев изучается как задача упорядочивание элементов неупорядоченного набора значений:

$$S = \{a_1, a_2, ...a_n\}, \tag{3}$$

в порядке монотонного возрастания или убывания:

$$S \sim S' = \left\{ \left(a_1', a_2', ...a_n'\right) : a_1 \le a_2{}' \le ... \le a_n' \right\}. \tag{4}$$

Существует большое количество возможных способов решения этой задачи; один из наиболее полных обзоров алгоритмов сортировки содержится в работе [6]. Процедура сортировки значений является весьма трудоемкой. Для изученных на данный момент стандартных методов таких как: пузырьковая сортировка, сортировка включением и другие необходимые операции определяются квадратичной зависимостью от числа отсортированных данных

$$T_1 \sim n^2. \tag{5}$$

Для более быстрых алгоритмов сортировки: слиянием, сортировка Шелла, быстрая сортировка величина трудоёмкости определяется значением:

$$T_1 \sim n^2 \log_2 n. \tag{6}$$

Выражение (6) дает оценку наименьшего числа операций для упорядочивания набора из n значений, меньшая трудоёмкость может быть получена только в частных случаях задачи.

Обеспечить увеличение скорости сортировки можно, если задействовать несколько $(p, p > 1)$ процессоров. В этом случае набор упорядоченных значений разделяется между процессорами. Данные в ходе сортировки передаются между вычислителями и подвергается сравнению. При этом для стандартизации представленного разделения для вычислителей вносится система

последовательной индикации и в ряде случаев необходимо, чтобы при в результате сортировки данные, находящиеся на процессорах с меньшими номерами, не превышали значений процессоров с большими номерами.

2.3.1 Параллельное обобщение базовой операции сортировки

Наибольший сегмент алгоритмов упорядочивания массивов данных основан на использовании базовой операции «сравнить и переставить» (compare-exchange), включающий в себя сопоставление пар значений из сортируемого набора значений и переноса данных, в случае когда их порядок не соответствует правилам сортировки (операция «сравнить и переставить» if ($a[i] > a[j]$) {temp = $a[i]$; $a[i]$ = a[j]; $a[j]$ = temp;}).

Целенаправленное использование операции compare-exchange дает возможность упорядочить данные. То, каким образом будут выбирается пары значений для сопоставления и различает алгоритмы сортировки. Для примера, в пузырьковой сортировке [2] реализуется последовательное сравнение всех рядом стоящих элементов; в результате прогонки по упорядочиваемому набору данных в крайнем, то есть в верхнем, элементе оказывается максимальное значение («всплывание пузырька»); после этого для продолжения сортировки этот уже упорядоченный элемент отбрасывается и действия повторяются (пузырьковая сортировка for (i=1; i<n; i++) for (j=0; j<n-i; j++) <сравнить и переставить элементы ($a[j]$, $a[j+1]$)>).

В случаи параллельного обобщения основной операции сортировки исследуем начальную ситуацию, то есть ту, когда число процессоров совпадает с количеством сортируемых значений (т.е. $p = n$). Сравнение значений a_i и a_j, находящихся, например, на процессорах P_i и P_j имеет смысл организовать следующим образом:

1) использовать взаимообмен присутствующих на вычислителях P_i и P_j значений;

2) сравнить на всех вычислителях P_i и P_j полученные схожие пары значений (a_i, a_j); итоги сравнения используются для разделения значений между вычислителями – на одном процессоре (например, P_i) остается меньший элемент,

другой процессор (P_j) заносит для последующей обработки большее значение пары $a_i' = \min\left(a_i, a_j\right), a_j' = \max\left(a_i, a_j\right)$.

Изучаемое параллельное обобщение базовой операции сортировки может быть особым образом адаптировано в случаи когда $p < n$, в случае когда число вычислителей будет меньше числа отфильтрованных значений. В таком случае каждый вычислитель хранит уже не одно значение, а кусок сортируемого набора значений. Эти блоки чаще всего переставляются в самом начале сортировки на каждом вычислителе в отдельности с использованием какого-либо быстрого алгоритма что называется предварительной стадией обработки. Затем, применяя схему поэлементного сравнивания, взаимодействия пары процессоров P_i и P_j для общей сортировки содержимого блоков A_i и A_j осуществляется последующим образом:

– выполнить взаимообмен блоков между процессорами P_i и P_j;

– объединить блоки A_i и A_j на каждом процессоре в один отсортированный блок двойного размера

– разделить полученный двойной блок на две равные по размеру части и оставить одну из этих частей на процессоре P_i, а другую часть – на процессоре P_j

$$\left[A_i \cup A_j\right]_{copm} = A_i' \cup A_j' : \forall a_i' \in A_i', \forall a_j' \in A_j' \Rightarrow a_i' \le a_j'.$$

Данная процедура чаще всего именуется в литературе как операция «сравнить и разделить» (compare-split). Необходимо заметить, что созданные в результате такой процедуры блоки на вычислителях P_i и P_j совпадают по размеру с начальными блоками A_i и A_j и все данные, располагаются на процессоре P_i, являются меньшими значений на процессоре P_j.

2.3.2 Пузырьковая сортировка

Алгоритм пузырьковой сортировки [6], в стандартном виде весьма трудоемок для разнесения на несколько вычислителей – сопоставление пар данных упорядочиваемого набора значений происходит только последовательно. В результате чего для параллельного применения используется

не сам этот алгоритм, а его улучшенная версия, известная в литературе как метод четно-нечетной перестановки (odd-even transposition) [3]. Смысл модификаций состоит в том, что алгоритм сортировки дополняется двумя различными принципами выполнения итераций метода – в зависимости от того четный или нечетный номер итерации сортировки для изменения выбираются элементы с четными или нечетными индексами, сопоставление выделенных данных чаще всего применяются с их правыми соседними элементами. На всех нечетных итерациях сравниваются пары (a_1, a_2), (a_3, a_4), … (a_{n-1}, a_n) (при четном n), на четных итерациях обрабатываются элементы (a_2, a_3), (a_4, a_5), … (a_{n-2}, a_{n-1}).

После n-кратного дублирования схожих итераций сортировки изначальный набор значений оказывается упорядоченным.

Разработка параллельной модификации для метода четно-нечетной перестановки уже не предусматривает каких-либо осложнений – соотношение пар значений на итерациях сортировки каждого типа являются независимыми друг от друга и могут рассчитывается параллельно. Для описания такого параллельного способа сортировки на рисунке 2.1 показан пример упорядочения данных при $n = 8$, $p = 4$.

Слева представляется номер и тип итерации метода, перечисляются пары вычислителей, для которых параллельно выполняются операция «сравнить и разделить»; взаимодействующие пары вычислителей выделены в таблице двойной рамкой. Для каждого шага сортировки показано состояние упорядочиваемого набора данных до и после выполнения итерации.

Необходимо отметить, что в показанном примере последняя итерация сортировки является ненужной – упорядоченный набор данных был найден уже на третьей итерации алгоритма. В общем случае выполнение параллельного метода может быть прекращено, если в течение каких-либо двух последовательных итераций сортировки состояние упорядочиваемого набора данных не было изменено. В результате, общее количество итераций может быть уменьшено и для выявления таких случаев нужно введение некоторого управляющего вычислителя, который анализировал бы состояние системы после

выполнения каждой итерации сортировки. Однако сложность такой коммуникационной операции (сборка на одном процессоре сообщений от всех процессоров) может быть столь значительной, что весь эффект от возможного сокращения итераций сортировки будет несравним с затратами на реализацию операций межпроцессорной передачи данных.

Номер и тип итерации	Процессоры			
	1	2	3	4
Исходные данные	2 3	3 8	5 6	1 4
1 нечет (1,2), (3,4)	2 3	3 8	5 6	1 4
	2 3	3 8	1 4	5 6
2 чет (2, 3)	2 3	3 8	1 4	5 6
	2 3	1 3	4 8	5 6
3 нечет (1, 2), (3, 4)	2 3	1 3	4 8	5 6
	1 2	3 3	4 5	6 8
4 чет (2, 3)	1 2	3 3	4 5	6 8
	1 2	3 3	4 5	6 8

Рисунок 2.1 – Пример сортировки данных параллельным методом четно-нечетной перестановки

Оценим трудоемкость рассмотренного параллельного метода. Продолжительность операций переноса данных между процессорами полностью определяется физической топологией вычислительной сети. Если логически-соседние вычислители, принимающие участие в выполнении операций «сравнить и разделить», являются близкими фактически (например, для линейки или кольца эти процессоры имеют прямые линии связи), общая коммуникационная сложность алгоритма пропорциональна количеству упорядочиваемых значений, то есть n. Вычислительная трудоемкость алгоритма определяется выражением

$$T_p = (n/p)\log(n/p) + 2n,$$

где первая часть соотношения учитывает сложность изначальной сортировки блоков, а вторая величина задает общее количество операций для слияния блоков в ходе выполнения операций «сравнить и разделить» (слияние двух блоков требует $2(n/p)$ операций, всего выполняется p итераций сортировки). С учетом

данной оценки показатели эффективности параллельного алгоритма имеют вид:

$$S_p = \frac{n\log n}{(n/p)\log(n/p)+2n},\ E_p = \frac{n\log n}{p\left[(n/p)\log(n/p)+2n\right]}$$

Анализ выражений показывает, что если количество процессоров совпадает с числом сортируемых данных (т.е. $p=n$), эффективность использования процессоров падает с ростом n; получение асимптотически ненулевого значения показателя E_p может быть обеспечено при количестве процессоров, пропорциональных величине $\log n$.

2.3.3 Сортировка Шелла

Общий смысл алгоритма Шелла [6] заключается в сопоставлении на начальных этапах сортировки пар значений, находящихся достаточно далеко друг от друга в упорядочиваемом наборе данных. Представленная модификация метода сортировки даёт возможность быстро переносить далекие неупорядоченные пары значений (сортировка таких пар обычно требует большого количества перестановок, если используется сравнение только соседних элементов).

Для алгоритма Шелла может быть представлен параллельный аналог метода, если топология вычислительной сети имеет структуру N-мерного гиперкуба (т.е. количество процессоров равно $p=2^N$). Выполнение сортировки может быть разделено на два последовательных этапа. На первом этапе (N итераций) выполняется взаимодействие процессоров, являющихся соседними в структуре гиперкуба (но эти процессоры могут оказаться далекими при линейной нумерации; для определения соответствия двух принципов нумерации процессоров может быть использован, как и ранее, код Грея). Второй шаг состоит в создании простых итераций параллельного алгоритма четно-нечетной сортировки. Итерации этого этапа выполняются до прекращения фактического изменения сортируемого набора и, таким образом, общее количество L таких итераций может быть различным – от 2 до p. Сложность параллельного варианта алгоритма Шелла определяется выражением:

$$T_p = (n/p)\log(n/p) + (2n/p)\log p + L(2n/p),$$

где вторая и третья части соотношения фиксируют вычислительную сложность первого и второго этапов сортировки соответственно. Не трудно заметить, что эффективность представленного параллельного способа сортировки оказывается гораздо лучше показателей обычного алгоритма четно-нечетной сортировки при $L < p$.

2.3.4 Быстрая сортировка

При кратком изложении алгоритм быстрой сортировки [18] опирается на поочерёдном разделении сортируемого набора значений на блоки меньшего размера таким образом, что между значениями различных блоков обеспечивается сопоставление упорядоченности (для каждой пары блоков существует блок, все значения которого будут меньше значений другого блока). На первом шаге метода осуществляется деление изначального набора значений на первые две части – для компоновки такого разделения находится некоторый основной элемент и все значения набора, меньшие ведущего элемента, передаются в первый формируемый блок, остальные значения создают второй блок набора. На следующей итерации сортировки указанные правила применяются поочерёдно для обоих сформированных блоков и т.д. После выполнения $\log n$ итераций изначальный массив данных оказывается упорядоченным.

Высокая эффективность быстрой сортировки в значительной степени определяется правильностью выбора основных элементов при создании блоков. В наихудшем случае сложность метода имеет тот же порядок сложности, что и пузырьковая сортировка (т.е. $T_1 \sim n^2$). При правильном выборе основных элементов, в случае когда разделение каждого блока производится на равные по размеру части, сложность алгоритма совпадает с быстродействием наиболее эффективных методов сортировки ($T_1 \sim n \cdot \log n$). В среднем случае количество операций, выполняемых алгоритмом быстрой сортировки, определяется выражением [6]:

$$T_1 = 1,4n\log n .$$

Параллельное обобщение алгоритма быстрой сортировки наиболее

простым способом может быть получено для вычислительной системы с топологией в виде N-мерного гиперкуба (т.е. $p=2^N$). Пусть, как и прежде, изначальный набор значений распределен между процессорами блоками одинакового размера n/p; результирующее расположение блоков должно соответствовать нумерации процессоров гиперкуба. Возможный способ выполнения первой итерации параллельного метода при таких условиях может состоять в следующем:

– выбрать каким-либо образом, главный элемент и передать его по всем вычислителям системы;

– разделить на каждом процессоре имеющийся блок данных на две части с использованием полученного ведущего элемента;

– образовать пары вычислителей, для которых битовое интерпретирование номеров отличается только в позиции N, и осуществить взаимообмен значениями между этими вычислителями; в следствии таких передач данных на вычислителях, для которых в битовом представлении номера бит позиции N равен 0, должны оказаться части блоков со значениями, меньшими основного элемента; процессоры с номерами, в которых бит N равен 1, должны сосредоточить, соответственно, все значения данных, превышающие значение основного элемента.

В результате применения такой итерации сортировки изначальный набор оказывается разделенным на две части, одна из которых (со значениями меньшими, чем значение главного элемента) находятся на вычислителях, в битовом представлении номеров которых бит N равен 0. Таких процессоров всего $p/2$ исходный N -мерный гиперкуб также оказывается разделенным на два гиперкуба размерности N-1. К этим подгиперкубам, в свою очередь, может быть параллельно применена описанная выше процедура. После N-кратного повторения схожих итераций для окончания сортировки достаточно упорядочить блоки данных, получившиеся на каждом отдельном процессоре ВС.

Для наглядности на рисунке 2.1 проиллюстрирован пример сортировки данных при n = 16, p = 4 (т.е. блок каждого процессора содержит 4 элемента). На

этом рисунке вычислители представлены в виде прямоугольников, внутри которых показано содержимое упорядочиваемых блоков данных; данные блоков приводятся в начале и при завершении каждой итерации сортировки. Взаимодействующие пары вычислителей соединены двунаправленными стрелками. Для разделения данных выбираются наилучшие значения главных элементов: на первой итерации для всех процессоров использовалось значение 0, на второй итерации для пары процессоров 0, 1 ведущий элемент равен 4, для пары процессоров 2, 3 это значение было принято равным – 5.

Эффективность параллельного метода быстрой сортировки, как и в последовательном варианте, во многом зависит от правильности выбора значений ведущих элементов. Нахождение общего правила для выбора этих значений не представляется возможным; трудоёмкость такого выбора может быть снижена, если выполнить упорядочение локальных блоков вычислителей перед началом сортировки и обеспечить однородное распределение сортируемых значений между вычислителями ВС.

Время выполняемых операций переноса значений обуславливается операцией рассылки главного элемента на каждой итерации сортировки – общее количество межпроцессорных обменов для этой операции на N-мерном гиперкубе может быть ограничено оценкой

$$\sum_{i=1}^{N} = N(N+1)/2 \sim \log^2 p,$$

и взаимообменом частей блоков между соседними парами процессоров – общее количество таких передач соответствует количеству итераций сортировки, т.е. равно $\log p$, общий объем передаваемых данных не превышает удвоенного объема процессорного блока, т.е. ограничен величиной $2n/p$.

Вычислительная сложность метода обуславливается трудоёмкостью локальной сортировки процессорных блоков, временем выбора ведущих элементов и сложностью разделения блоков, что в целом может быть выражено при помощи соотношения:

$$T_1 = (n/p)\log(n/p) + \log p + \log(n/p)\log p.$$

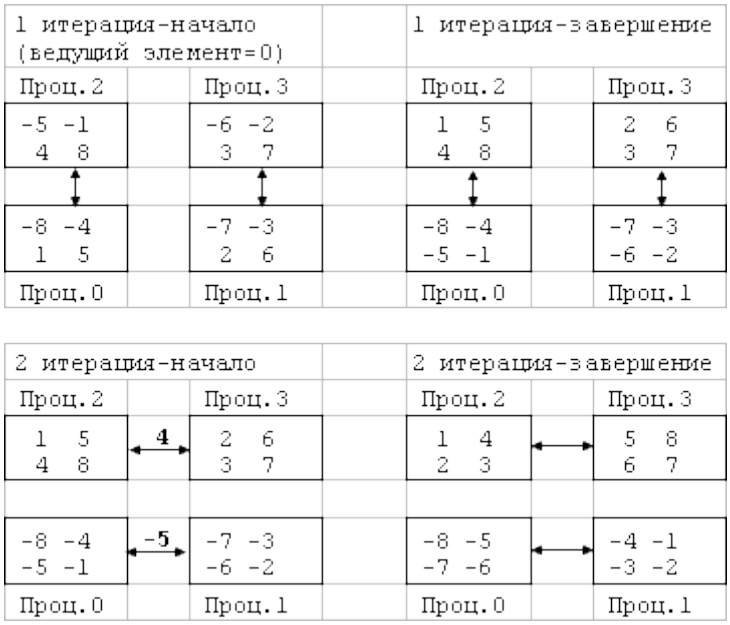

Рисунок 2.1 – Пример упорядочивания данных параллельным методом быстрой сортировки (без результатов локальной сортировки блоков)

2.3.5 Сеть обменной сортировки со слиянием Бэтчера

Параллельный алгоритм сортировки массива на основе метода «обменной сортировки со слиянием» Бэтчера состоит из двух этапов, причем первый из них совпадает с первым этапом рассмотренного выше алгоритма. На втором этапе так же выполняется ряд слияний упорядоченных фрагментов, но есть два существенных отличия от алгоритма сдваивания. Во-первых, порядок слияния определяется сетью сортировки. Во-вторых, и это самое важное, при слиянии фрагментов не происходит увеличение размера обрабатываемого на каждом из процессоров фрагмента [14]. В следствии этого не происходит увеличения, к концу процесса сортировки, объема передаваемых от процессора к процессору данных, равно как не происходит и уменьшения числа выполняющих полезную работу процессоров. В этом заключается существенное отличие от метода сдваивания, на последнем шаге которого работает только один процессор, принимающий

половину всего сортируемого массива. Итак:

– на каждом из процессоров с помощью алгоритма dhsort сортируется фрагмент массива длинной n/p. При этом будем полагать, что $n=rp$, где r – целое

число.

– отсортированные фрагменты объединяются с помощью процедуры, выполняемой модулем компаратора слияния [10], причем последовательность слияний определяется алгоритмом «обменной сортировки со слиянием» Бэтчера [6].

Процедура, выполняемая компаратором слияния заключается в преобразовании фрагментов массива, расположенных на процессорах **а** и **б**: – процессоры **а** и **б** обмениваются хранящимися на них отсортированными фрагментами, после чего на каждом из процессоров **а** и **б** оказываются два предварительно упорядоченных фрагмента. – процессор а выделяет из двух фрагментов, длины m каждый, m наименьших элементов, формируя новый отсортированный фрагмент длины m.

Одновременно с этим, процессор **б** выделяет m наибольших элементов, формируя новый отсортированный фрагмент длины m. Опишем теперь алгоритм сортировки предварительно упорядоченных фрагментов массива. Алгоритм рекурсивный и предполагает при сортировке $n+m$ фрагментов выполнение независимой сортировки n первых фрагментов и m последних фрагментов, после чего объединение двух сформированных упорядоченных массивов с помощью (n, m)-сети слияния фрагментов. Предлагаемый алгоритм практически полностью совпадает с подробно изложенным в монографии Кнута [6] методом, с той разницей, что вместо сортировки элементов выполняется сортировка фрагментов массива. (n, m)-сеть слияния фрагментов можно описать следующим образом:

– если $n = 0$ или $m = 0$, то сеть пуста.

– если $n = 1$ и $m = 1$, то сеть состоит из единственного компаратора слияния.

– если $nm > 1$, то, обозначив объединяемые последовательности через $\langle A_1, A_2, ..., A_n \rangle$ и $\langle B_1, B_2, ..., B_n \rangle$, объединим последовательности фрагментов, имеющих нечетные номера

$$\langle A_1, A_3, ..., A_{2\lceil n/2 \rceil - 1} \rangle \otimes \langle B_1, B_3, ..., B_{2\lceil m/2 \rceil - 1} \rangle \to \langle C_1, C_2, ..., C_r \rangle$$

Аналогично поступим с последовательностями фрагментов, имеющих четные номера $\langle A_2, A_4, ..., A_{2\lfloor n/2 \rfloor} \rangle \otimes \langle B_2, B_4, ..., B_{2\lfloor n/2 \rfloor - 1} \rangle \to \langle D_1, D_2, ..., D_t \rangle$, где $r = \lceil n/2 \rceil + \lceil m/2 \rceil$, $t = \lfloor n/2 \rfloor + \lfloor m/2 \rfloor$.

Сформируем окончательный результат $\langle E_1, E_2, ..., A_{n+m} \rangle$, выполнив операции компаратора слияния над парами фрагментов:

$C_1 = E_1$

$\langle D_1 \rangle \otimes \langle C_2 \rangle \to \langle E_2, E_3 \rangle,$

$\langle D_2 \rangle \otimes \langle C_3 \rangle \to \langle E_4, E_5 \rangle,$

$\vdots$

если $r = t + 1$

$\langle D_t \rangle \otimes \langle C_r \rangle \to \langle E_{n+m-1}, E_{n+m} \rangle,$

иначе, $C_r \to E_{n+m}$.

Рассмотрим в качестве примера этапы сортировки шести фрагментов массива длиной $n/6$ каждый.

– отсортировать на каждом вычислителе массивы длиной $m=n/6$.

– выполнить слияние упорядоченных фрагментов в соответствии со следующей схемой (рисунок 2.2):

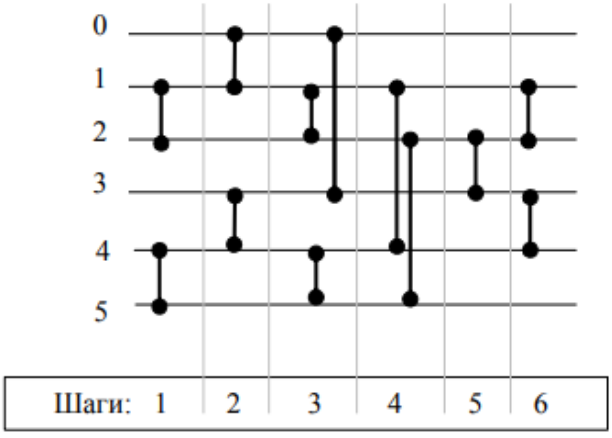

Рисунок 2.2 – Порядок выполнения слияния фрагментов массива методом сдваивания на восьми процессорах

Каждая горизонтальная черта на рисунке 2.2 отображает вычислитель, каждая вертикальная – компаратор слияния. Применяя принцип нулей и единиц

[6] можно показать, что представленный алгоритм верно сортирует случайные массивы только в том случае, когда размеры объединяемых фрагментов в точности равны друг другу. Из этого следует, что при сортировке реальных значений следует дополнять массив фиктивными элементами, с тем, чтобы общая длина массива была кратна числу процессоров.

Для подтверждения необходимости равенства размеров фрагментов достаточно убедится, что есть такой массив, неверно сортируемый сетью, в случае когда длины фрагментов не равны. Соответствующий пример приведен на рисунке 2.3, на котором представлена сортировка массива, содержащего три единицы (черные кружки) и четыре нуля (белые кружки). В следствии сортировки элементы массива, имеющие наибольшие значения должны находится на процессорах с большими номерами, однако на процессоре 6 после сортировки остался один из нулей, что является ошибкой.

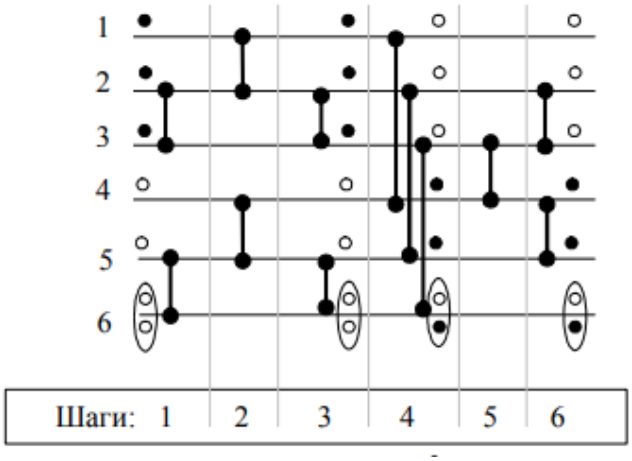

Рисунок 2.3 – Пример неправильного упорядочивания фрагментов неравной длины: на процессорах 1 … 5 размещено по одному элементу, на процессоре 6 – два элемента

Можно показать, что общий объем данных, передаваемых (принимаемых) каждым из вычислителей не превышает n, независимо от количества используемых вычислителей, что обуславливает хорошую эффективность алгоритма в целом. Помимо этого, на каждом вычислителе в какой момент

времени не требуется хранить более чем $3\frac{n}{p}$ элементов сортируемых массивов (два исходных фрагмента и один фрагмент результата). Таким образом, общий объем сортируемых данных ограничен только суммарным объемом оперативной памяти всех используемых процессорных узлов (в предположении, что все процессорные узлы равноценны с точки зрения размера доступной им оперативной памяти). Общее число сортируемых элементов не ограничивается объемом оперативной памяти каждого из процессорных узлов.

Из представленной схемы (рисунок 2.3) вытекает, что слияние фрагментов может быть выполнено за шесть шагов. К примеру, на четвертом шаге выполняется обмен значениями и процедура компаратора слияния одновременно между парами процессоров (0,3), (1,4), (2,5). На этом шаге работой обеспечены все процессоры. Однако на остальных шагах это не так, например, на пятом шаге простаивают все процессоры, кроме пары (2,3). Таким образом, описанный алгоритм обладает ограниченной степенью внутреннего параллелизма, не зависимо от типа используемой вычислительной системы.

Время сортировки массива оценивается следующим выражением,

$$T(n,p) = k\frac{n}{p}\left(\log_2\left(\frac{n}{p}\right) + b\cdot s_p\right),\ s_p \approx \frac{\lceil \log_2 p\rceil\left(\lceil \log_2 p\rceil + 1\right)}{2},$$

где sp – число шагов слияния (точные значения для некоторых p приведены в таблице 2.2), $b \geq 1$ – константа, определяющая время слияния двух фрагментов массива, включая время либо на передачу данных между процессорами, либо на синхронизацию (если используется вычислительная система с общей памятью). В случае использования общей памяти при малом числе процессоров b~1, соответственно, максимальное значение коэффициента эффективности использования вычислительной мощности дается выражением:

$$E^{\max}(n,p) = \frac{t(n,1)}{pt(n,p)} = \frac{\log_2 n}{\log_2 n + s_p - \log_2 p} \approx \frac{1}{1+\log_n p\left(\log_2 p - 1\right)/2}.$$

Таким образом, без учета накладных расходов на обмены, максимально возможная эффективность используемого алгоритма, при отсутствии накладных

расходов на обмены заведомо меньше 100 %. Для некоторых значений p точные значения величины s_p приведена в таблице 2.2 [14].

Таблица 2.2 – Произвольные числа полученные функцией rand()

Windows, Visual C 6.0		Linux, gcc 3.2.2	
RAND MAX =	32767	RAND MAX =	2147483647
	41		1804289383
	18467		846930886
	6334		1681692777
	26500		1714636915
	19169		1957747793
	15724		424238335
	11478		719885386
	29358		1649760492
	26962		596516649
	24464		1189641421

2.4 Обзор параллельных архитектур и моделей программирования

На рисунке (рисунок 2.4) показана блок-схема «общего параллельного компьютера» с p-процессорами. Здесь каждый процессор Proc_i, выполняющий какую-то программу, иногда ссылаясь на данные, хранящиеся в собственной памяти Mem_i, а иногда и на связь с другими процессорами через сеть соединений. Это изображение достаточно абстрактно, чтобы включить случай, когда весь параллельный компьютер находится в одном «процессоре» (как в случае с двумя единицами с плавающей запятой в RS6000 / 590), где параллельный компьютер находится в одной комнате (IBM SP- 2- который состоит из нескольких RS6000 - или большинства других коммерческих параллельных компьютеров) или где параллельный компьютер состоит из компьютеров, расположенных по континентам, а сеть межсоединений - это Интернет, телефонная система или какая-либо другая сеть.

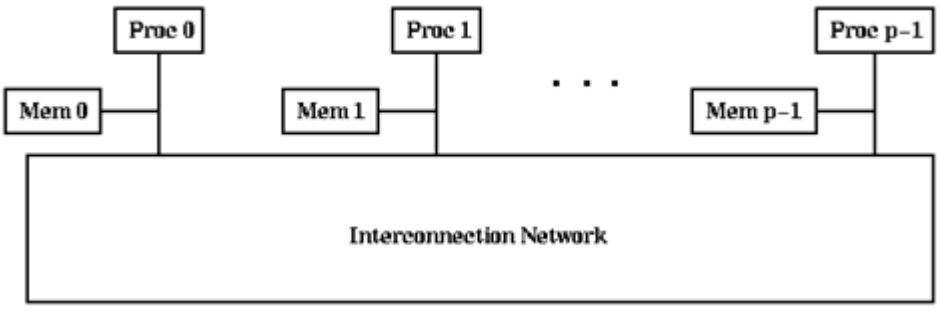

Рисунок 2.4 – Блок-схема общего параллельного компьютера

Существуют три стандартные функции, которые должна предоставлять любая параллельная архитектура. Ниже представлены конкретные реализации этих функций.

– параллелизм, или заставить процессор работать одновременно.

– межпроцессорная связь или обмен процессорами.

– синхронизация, например, получение процессорами «согласования» на значение переменной или время начала или остановки.

Модель программирования – это интерфейс, предоставляемый пользователю языком программирования, компилятором, библиотеками, системой времени выполнения или чем-либо еще, так что пользователь непосредственно «программирует». Неудивительно, что любая модель программирования должна предоставлять пользователю возможность выражать параллелизм, связь и синхронизацию в своем алгоритме.

Исторически сложилось так, что люди обычно проектировали параллельную архитектуру, а затем модель программирования для ее соответствия. Другими словами, модель программирования выражала параллелизм, связь и синхронизацию, тесно связанную с деталями архитектуры. В результате, когда машина стала устаревшей и были построены новые, все написанные для нее программы также стали устаревшими, и пользователям пришлось начинать с нуля. Это не привело к широкому распространению параллельного программирования.

В то же время люди поняли, что возможно и важно создавать модели программирования независимо от архитектур, позволяя людям создавать полезные аппараты программного обеспечения, которые выживают в поколениях машинных архитектур. Это означает, что мы будем рассматривать системы, которые мы программируем, как состоящие из слоев, с моделью программирования наверху, машиной внизу, а также компилятором, библиотеками и т.д. Предоставляя отображение из модели программирования на машину, что скрывает детали машины.

Например, Connection Machine Fortran (CMF) был первоначально разработан для компьютеров поколения CM-2 и предоставил модель программирования (называемую параллелизмом данных ниже), которая хорошо подходит для архитектуры CM-2. В частности, CM-2 по существу требовал, чтобы каждый процессор выполнял одну и ту же инструкцию одновременно, очень ограниченную форму параллелизма (называемую SIMD ниже). Следующее поколение CM5 позднее было разработано для обеспечения гораздо более гибкого параллелизма (называемого MIMD ниже), но компилятор CMF был изменен для продолжения запуска CMF-программ в новой архитектуре. Таким образом, модель программирования, предоставленная пользователю (CMF), осталась прежней, но сопоставление с машиной (способ создания кода компилятора) значительно изменилось.

Таким образом, большая польза от разделения машин от моделей программирования – это возможность писать программы, которые можно переносить на несколько машин. Потенциальным недостатком является потеря производительности, в зависимости от того, насколько хорошо модель программирования может быть сопоставлена с конкретной архитектурой. Не все модели программирования одинаково хорошо сопоставляются со всеми архитектурами, и есть много трудных исследовательских вопросов, на которые лучше всего реализовать эти сопоставления. Поэтому существующие компиляторы, библиотеки и системы времени выполнения часто являются неполными, неэффективными, неэффективными или их комбинацией. Предостережение программиста . Когда мы изучим модели программирования позже, мы потратим некоторое время на то, как работают эти сопоставления, поэтому мы можем предсказать, эффективна ли одна параллельная программа, чем другая, для решения одной и той же проблемы.

Следующие разделы организованы следующим образом. Во-первых, мы обсудим некоторые стандартные способы, с помощью которых параллельные архитектуры обеспечивают параллелизм, связь и синхронизацию. Во-вторых, мы очень кратко сделаем то же самое для наших моделей программирования, указав,

как модели программирования имеют «естественные» архитектуры, для которых они были первоначально разработаны, хотя они могут быть сопоставлены с другими. Последующие лекции будут подробно изучать эти модели программирования. В-третьих, мы представим стандартные способы измерения производительности параллельных программ.

2.4.1 Связь и синхронизация в компьютерных архитектурах

Двумя основными стилями параллелизма являются SIMD и MIMD , или данные с несколькими инструкциями и множественными данными с несколькими инструкциями. Это старая терминология, датируемая таксономией Флинна параллельных машин в 1960-х годах. SIMD означает выполнение одной и той же операции (например, одиночная инструкция) по нескольким частям данных параллельно, а MIMD означает выполнение произвольных операций (например, нескольких инструкций) на разных данных одновременно. В случае SIMD можно подумать о родовом параллельном процессоре на рисунке выше как дополненном другим управляющим процессором, который в каждом цикле отправляет общую команду каждому процессу Proc_i. Примеры параллелизма SIMD включают векторные операции на одном процессоре Cray T-90и более ранние векторные машины; Мышление машины CM-2; и единый конвейерный блок с плавающей запятой в RS6000 / 590, где сумматор и множитель должны работать в конвейере, управляемом одной инструкцией с плавным умножением. Примеры MIMD-параллелизма включают почти все коммерческие параллельные машины, где каждый процессор также может быть запрограммирован как стандартная последовательная машина, выполняющая последовательные задания. Обратите внимание, что конкретная машина может демонстрировать параллельность SIMD и MIMD на разных уровнях, например, многопроцессорный Cray T90. MIMD-параллель более гибкая и более распространенная, чем параллельность SIMD, которая теперь обычно появляется в отдельных единицах с плавающей запятой или памятью.

Мы должны сначала обсудить, как архитектура называет разные ячейки памяти, к которым должны обращаться инструкции. Ни люди, ни машины не

могут общаться без единого набора имен, на которых они согласны. Напомним, что память обычного (непараллельного) компьютера состоит из последовательности слов , каждая из которых названа по своему уникальному адресу, целым числом. Типичная компьютерная инструкция будет выглядеть так: «load r1, 37», которая говорит, чтобы загрузить слово, сохраненное по адресу 37 памяти, в регистр r1. Изучив рисунок 2.4, мы видим, что есть несколько mem mem. Вызываются два основных способа назвать ячейки памяти этих множественных воспоминаний разделяемой памяти и распределенной памяти. В общей памяти каждое слово в каждой памяти имеет уникальный адрес, на который согласны все процессоры. Поэтому, если Proc_1 и Proc_3 выполняют команду «load r1, 37», одни и те же данные из одного места в одном Mem_i будут загружены в регистр r1 Proc_1 и регистр r1 Proc_3. В распределенной памяти Proc_1 будет извлекать местоположение 37 из Mem_1, а Proc_3 будет извлекать местоположение 37 Mem_3. Поэтому для связи на компьютере с общей памятью Proc_1 и Proc_3 просто нужно загрузить и сохранить в общий адрес, например 37. На машине с распределенной памятью явные сообщения должны быть отправлены по сети связи и обрабатываться иначе, чем просто загрузок и магазинов.

На рынке можно найти как успешные компьютеры с общей памятью , так и успешные компьютеры с распределенной памятью (например, IBM SP-2, Intel Paragon, а также сети рабочих станций). Грубо говоря, машины с общей памятью предлагают более быструю связь, чем машины с распределенной памятью, и они, естественно, поддерживают модель программирования (также называемую разделяемой памятью), которая часто проще программировать, чем модель программирования, естественная для распределенных машин памяти (так называемая передача сообщений). Однако устройства с общей памятью сложнее строить (или, по крайней мере, более дорогостоящие на процессор), чем машины с распределенной памятью, для большого числа процессоров (более 32, скажем, хотя это число растет).

Важным свойством коммуникации является его стоимость, которая

необходима для понимания эффективности параллельной программы. Предположим, мы хотим отправить n слов данных с одного процессора на другой. Простейшей моделью, которую мы будем использовать в течение времени, требуемого этой операцией, является время для отправки *n* слов = латентность + *n* / полоса пропускания

Мы рассматриваем отправку n слов за раз, потому что на большинстве машин иерархия памяти диктует, что наиболее эффективно отправлять группы смежных слов (например, строки кэша) одновременно. Задержка (в секундах) измеряет время отправки «пустого» сообщения. Полоса пропускания (в единицах слова / секунда) измеряет скорость, с которой слова проходят через сеть межсетевого соединения. Форма этой формулы должна быть знакома: напомним, что время обработки n слов по конвейеру s-ступеней, каждая из которых занимает t секунд, равна

$(s-1)*t + n*t =$ латентность + *n* / полоса пропускания

Таким образом, сеть взаимодействия работает как конвейер, «накачивая» данные через него со скоростью, заданной полосой пропускания, и с задержкой, заданной задержкой для первого слова в сообщении, чтобы сделать ее по сети.

Чтобы дать ощущение порядка величины стоимости связи, давайте сможем изменить единицы измерения латентности и полосы пропускания в циклах и слова за цикл соответственно. Напомним, что цикл в базовую единицу времени, в которой компьютер выполняет одну операцию, например, add. На компьютерах с общей памятью, где связь быстрее, латентность может составлять сотни циклов. С другой стороны, машина с распределенной памятью, состоящая из сети рабочих станций с PVM через Ethernet, может быть $O(10^5)$ циклов (включая задержки аппаратного и программного обеспечения). Другими словами, в одном сообщении процессор может вместо того, чтобы делать от 10^2 до 10^5 других полезных операций. Время на одно слово (обратное полосе пропускания) обычно намного ниже, чем латентность (от *O* (10) до *O* (1000) Мбайт / с), поэтому часто гораздо эффективнее отправлять одно большое сообщение, чем многие небольшие. Мы увидим, что разница между хорошим параллельным алгоритмом

и плохим часто связана с количеством сообщений, которые они выполняют.

Некоторые компьютеры с общей памятью построены из кластеров небольших компьютеров с общей памятью. В этом случае часто быстрее обращаться к памяти, расположенной в кластере, чем к памяти в другом кластере. Такие машины называются машинами NUMA для NonUniform Memory Access и требуют (по крайней мере) двух латентностей и двух полос пропускания (для близлежащей и удаленной памяти) для точного моделирования связи.

Синхронизация относится к необходимости согласования двух или нескольких процессоров времени или значения некоторых данных. Три наиболее распространенных проявления синхронизации (или их отсутствие) - это взаимное исключение, барьеры и последовательность памяти.

Взаимное исключение означает разрешить доступ только одному процессору к определенной ячейке памяти за раз. Чтобы проиллюстрировать необходимость такого объекта, предположим, что мы хотим вычислить сумму s чисел p x_i, где Proc_i вычислил x_i. «Очевидный» алгоритм предназначен для каждого процессора для получения s от процессора, который его владеет (скажем, Proc_0), добавляет x_i в s и сохраняет s обратно в Proc_0. В зависимости от порядка, в котором каждый процессор обращается к s на Proc_0, ответ может варьироваться от истинной суммы до любой частичной суммы x_i. Например, предположим, что мы хотим только добавить x_1 и x_2. Вот две непредвиденные возможности для выполнения (ось времени вертикальна, и мы указываем время, в которое каждый процессор выполняет каждую инструкцию.

Нагрузки и запасы s включают связь с Proc_0. Конечным значением s, хранящимся в Proc_0, является x_2, так как это последнее значение, сохраненное в s. Если два магазина будут обратными во времени, конечное значение *s* будет равно x_1. Это называется условием расы, поскольку конечное значение s зависит от недетерминированного состояния того, является ли Proc_1 или Proc_2 немного впереди другого.

Взаимное исключение обеспечивает механизм, позволяющий избежать условий гонки, позволяя только одному процессору получить эксклюзивный

доступ к переменной. Это часто реализуется путем предоставления инструкции «test & set», которая задает конкретное слово для определенного значения, в то же время извлекает старое значение, без использования других процессоров. Позже, когда мы обсудим модели программирования, мы покажем, как решить проблему добавления чисел с помощью команды test & set.

Барьеры включают в себя каждый процессор, ожидающий в той же точке программы, чтобы все остальные «догоняли», прежде чем продолжить. Это необходимо, чтобы убедиться, что параллельное задание, на котором работал весь процессор, действительно завершено. Барьер может быть реализован в программном обеспечении, используя инструкцию test & set выше, но для некоторых архитектур для этого есть специальное оборудование.

Согласованность памяти относится к проблеме на компьютерах с общей памятью с кешами. Предположим, что процессоры Proc_1 и Proc_2 оба считывают местоположение 37 памяти в свои кеши. Целью кэшей является устранение необходимости доступа к медленной памяти при доступе к обычно используемым данным, например, к местоположению 37. После прочтения и использования местоположения 37 предположим, что Proc_1 и Proc_2 попытаются сохранить в нем новые, разные значения. Как будут согласовываться разные взгляды памяти, проведенные Proc_1 и Proc_2 Какое значение в конечном итоге вернет его в основную память в местоположении 37 Настаивая на том, что каждый процессор имеет одинаковое представление обо всех ячейках памяти одновременно, так что местоположение 37 имеет одно значение для всей машины, представляется очень разумным. Это называется последовательной последовательностью памяти, потому что это то же самое, что память смотрит на последовательную машину. Но это дорого реализовать, потому что все записи должны попадать в основную память и обновлять все остальные кэши на компьютере, а не просто обновлять локальный кеш. Эти расходы привели к тому, что некоторые устройства с общей памятью предложили слабые (er) модели согласованности памяти , которые не позволяют пользователю программировать таким образом, чтобы эти проблемы не

возникали. Мы обсудим это позже, когда обсуждаем модели программирования.

2.4.2 Связь и синхронизация в моделях программирования

Параллелизм данных означает применение одной и той же операции множества операций ко всем элементам структуры данных. Эта модель эволюционировала исторически из архитектур SIMD. Простейшими примерами являются операции с массивами, такие как *C=A+B*, где *A, B* – массивы, и каждая запись может быть добавлена параллельно. Параллелизм данных обобщает более сложные операции, такие как глобальные суммы, и более сложную структуру данных, например, связанные списки. Связь подразумевается, что означает, что если оператор *C=A+B* требует связи, поскольку элементы массивов *A, B* и C хранятся на разных процессорах, это делается незаметно для пользователя. Синхронизация также неявна, так как каждый оператор завершает выполнение до начала следующего. CMF очень похож на недавно появившийся стандарт High Performance Fortran (HPF), которые поддерживают многие производители. Мы будем использовать последовательный язык программирования Matlab для прототипа данных-параллельных кодов, поскольку он много похож на CMF во многих отношениях. Последний последовательный язык Fortran 90 обновляет Fortran 77, чтобы иметь аналогичные операции с массивами, а также рекурсию, структуры, ограниченные указатели и другие более современные функции языка программирования.

Передача сообщений означает запуск p независимых последовательных программ, записанных на последовательных языках, таких как C или Fortran, и передача посредством вызова подпрограмм, таких как send (data, destination_proc) и получение (data, source_proc) для отправки данных из одного процессора (source_proc) в другой (destination_proc). В дополнение к отправке и получению обычно существуют подпрограммы для вычисления глобальных сумм, барьерной синхронизации и т. Д. Поскольку неудобно поддерживать p разных программных текстов для p процессоров (так как *p* будет варьироваться от машины к машине), обычно имеется один текст программы, выполняемый на всех процессорах. Но поскольку программа может работать на основе номера

процессора (MY_PROC) процессора, на котором он выполняется,

Разные процессоры могут работать полностью независимо. Это называется программированием SPMD (однопроцессорные множественные данные). Эта модель программирования является естественной для MIMD-систем с распределенной памятью и поддерживается на CM-5 в библиотеке CMMD для передачи сообщений (на самом деле это не акроним). Две аналогичные, но более портативные библиотеки для передачи сообщений – PVM и MPI. Передача сообщений – это «программирование на языке ассемблера» параллельных вычислений, часто приводящая к длительным, сложным и подверженным ошибкам программам. Но он также самый портативный, поскольку PVM и MPI запускаются (или запускаются) на всех практически всех платформах.

Совместное программирование памяти с потоками Совместное программирование памяти с потоками является естественной моделью программирования для компьютеров с общей памятью. Существует один программный текст, который изначально запускается на одном процессоре. Эта программа может исполнять такую статуту, как «spawn (proc, 0)», что приведет к тому, что другой процессор выполнит процедуру proc с аргументом 0. Подпрограмма proc может «видеть» все переменные, которые она обычно может видеть в соответствии с правилами определения области серийный язык. Основная программа, которая породила proc, может позже подождать, пока proc завершит работу с барьерным заявлением, продолжит запуск других вызовов подпрограмм и т.д. В этом семестре мы будем использовать две такие модели программирования. Первый – это C, дополненный потоками Solaris, работающий на мультипроцессоре Sparc. Второй Sather, параллельный объектно-ориентированный язык, разработанный и внедренный локально в ICSI .

Split-C – это локально разработанный и реализованный язык, который дополняет C с помощью достаточно параллельных конструкций, чтобы выявить основной параллелизм, предоставляемый машиной. Он включает в себя функции параллелизма данных, передачи сообщений и общей памяти. Первоначально реализованный для CM-5, он был перенесен на многие машины, включая все

платформы, которые мы будем использовать в этом семестре.

2.4.3 Измерение эффективности параллельных программ

Приведен список способов использования производительности для параллельной программы. Мы будем использовать эти критерии для измерения разницы между хорошей параллельной программой и плохим.

Во всех этих примечаниях мы будем использовать p для обозначения количества доступных параллельных процессоров, а n – размер проблемы. Например, в случае умножения матрицы n может означать размерность матрицы. Было бы также разумно измерить размер проблемы на количество входных данных, которое было бы квадратом размерности матрицы для матричного умножения. Поэтому очень важно быть конкретным при определении n.

– $T(p, n)$ – это время для решения проблемы размера n на p процессорах. Иногда мы опускаем n и записываем $T(p)$.

– $T(1, n)$ или T (1) - последовательное или последовательное время с использованием наилучшего последовательного алгоритма. Лучшим последовательным алгоритмом может быть не параллельный алгоритм с p, установленным в 1, поскольку параллельный алгоритм может иметь ненужные служебные данные.

– Ускорение (p) = $T(1) / T(p)$ измеряет, насколько быстрее вы переходите на p-процессоры, чем 1 процессор. «Ускоренный график» - это ускорение (p), построенное по сравнению с p. Если вы используете плохой последовательный алгоритм, T (1) и так повышение скорости (p) будут искусственно большими, и ваш параллельный алгоритм будет выглядеть искусственно хорошим

– Эффективность (p) = Ускорение (p) / p. «График эффективности» - это эффективность (p), построенная по сравнению с p. Хороший алгоритм будет иметь эффективность около 1.

Ускорение и эффективность – наиболее распространенные способы сообщать о производительности параллельного алгоритма

Это «псевдо-теорема», что повышение скорости(Speedup) (p) <= p или

эффективность (p) <= 1, поскольку в принципе один процессор может имитировать действия p-процессоров не более чем в p раз больше времени, принимая один шаг каждой из последовательных программ p, составляющих параллельную программу, в круговом режиме. На самом деле иногда мы можем получить «сверхлинейное ускорение», то есть Speedup (p)> p, если либо

– алгоритм недетерминирован, и в параллельном алгоритме используется другой путь, чем последовательный алгоритм (это происходит в задачах поиска и символических вычислениях)

– проблема слишком велика для одного процессора, не достигнув более медленного уровня в иерархии памяти (например, подкачки или «перебивание»)

Из нашей псевдо-теоремы следует, что если мы построим прямую линию «Ускорение (p) = p» через начало координат на графике ускорения, это будет связано с истинным ускорением сверху, а качество параллельной реализации может быть измерено тем, насколько близко наш алгоритм доходит до этой «идеальной кривой ускорения».

Идеальное ускорение редко достижимо. Вместо этого мы часто пытаемся разработать «масштабируемые алгоритмы», где КПД (p) ограничена от 0 при возрастании p. Это означает, что мы гарантируем, что получим какую-то выгоду, пропорциональную размеру машины, поскольку мы используем все больше и больше процессоров.

Общей вариацией этих показателей эффективности является следующее: поскольку люди часто покупают более крупную машину для решения большей проблемы, а не только по той же самой проблеме быстрее, мы позволяем размер проблемы n (p) расти с p, чтобы измерить «масштабированное ускорение» " T (1, n (p)) / T (p, n (p)) и «масштабированная эффективность» T (1, n (p)) / (T (p, n (p)) * p). Например, мы можем позволить n (p) расти так, чтобы объем используемой памяти был постоянным на каждый процессор, независимо от того, что такое p. В случае матричного умножения $C = A * B$ это означает, что размер матрицы N для p процессоров будет удовлетворять 3 * N = n (p) = p * M, где M - объем памяти, используемый на процессор.

Закон Амдаля дает простую оценку того, сколько ускорений мы можем ожидать: предположим, что проблема тратит часть $f < 1$ своего времени на выполнение работы, чем может быть распараллелирована, а доля $s = 1-f$ выполняет серийную работу, которая не может быть распараллелена. Тогда $T(p) = T(1) * f / p + T(1) * s$, а Speedup $(p) = T(1) / T(p) = 1 / (f / p + s) <= 1 / s$, независимо от того, насколько большой p. Эффективность $(p) = 1 / (f + s * p)$ переходит в 0 при возрастании p. Другими словами, ускорение ограничено $1 / s$, и фактически увеличение p прошлое s / f не может увеличить скорость более чем в два раза.

Закон Амдаля преподает нам этот урок: нам нужно убедиться, что в кодах нет серийного узкого места (часть s), если мы надеемся иметь масштабируемый алгоритм. Например, даже если только $s = 1\%$ программы является серийным, ускорение ограничено 100, и поэтому не стоит использовать машину с более чем 100 процессорами.

3 РАЗРАБОТКА ПАРАЛЛЕЛЬНОГО АЛГОРИТМА И ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

3.1 Чётно-нечётная сортировка слиянием Бэтчера

Метод рекурсивной координатной бисекции для декомпозиции расчетных сеток

В решении задачи декомпозиции ключевую роль играет выбор метода сортировки массива. В работе используется метод четно-нечетной сортировки слиянием Бэтчера.

Алгоритм четно-нечетной сортировки слиянием предложенный Бэтчером [14] обладает несложной реализацией и относительно легко распараллеливается. Также он хорошо адаптируется под интерфейс передачи сообщений MPI (Message Passing Interface).

3.1.1 Базовые операции

Рассмотрим реализацию данного алгоритма на примере следующих трех абстрактных операций:

– операция обмена compare-exchange – если элементы идут не по порядку, меняем их местами

– операция идеального тасования perfect shuffle – переупорядочивает массив так, как может перетасовать колоду только большой специалист этого дела: колода делится точно наполовину, затем карты по одной берутся из каждой половины колоды. Первая карта всегда берётся из верхней половины колоды. Если число карт в колоде чётное, в обеих половинах содержится одинаковое их число, если число карт нечётное, то лишняя карта идёт последней в верхней половине колоды [10]. Фактически, мы расставляем элементы первой половины массива по четным позициям, а из второй половины массива – по нечетной.

– perfect unshuffle – операция, обратная предыдущей. Элементы, на четных позициях, отправляются в первую половину массива-результата, на нечетных – во вторую

3.1.2 Алгоритм чётно-нечётной сортировки слиянием Бэтчера

Предположим, что входной массив имеет размер, равный степени двойки. Тогда с помощью введенных операций формулируем алгоритм. При помощи операции unshuffle массив разбивается на две части. После нужно отсортировать каждую из половин, а затем слить обратно с помощью операции shuffle.

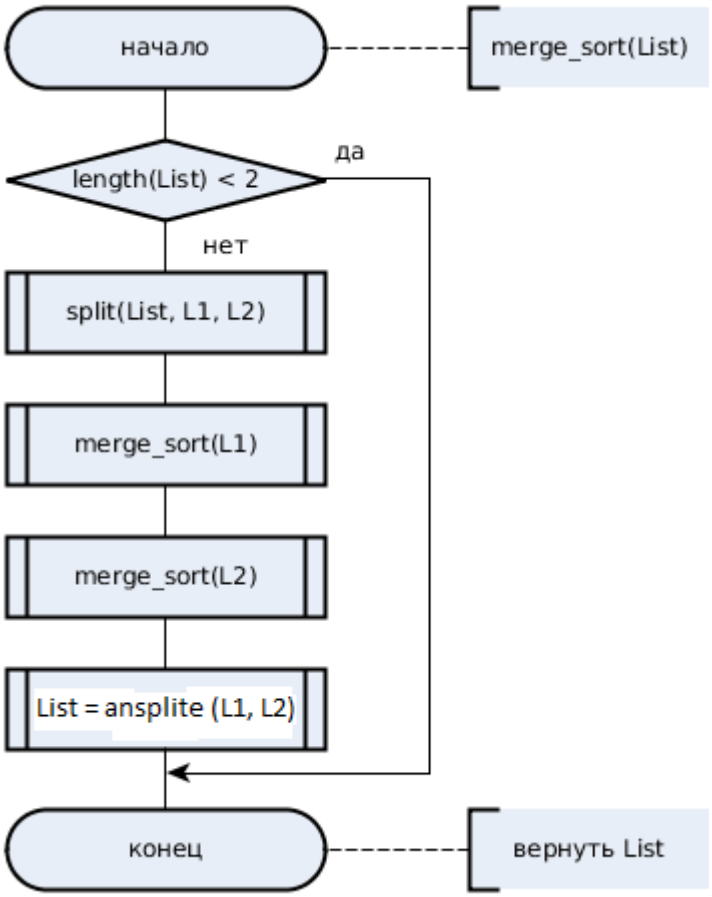

Рисунок 3.1 – Блок-схема алгоритма чётно-нечётной сортировки Бэтчера

Для массивов, длина которых не совпадает с степенью двойки существует несколько способов преобразования.

Самый простой способ – добавить необходимое число псевдоэлементов до степени двойки, которые всегда будут больше (или меньше) любого элемента в исходном массиве.

Второй подход такой. Т.к. любое число можно представить, как сумму степеней двойки, то разбить массив на такие подмассивы, отсортировать их по отдельности сортировкой Бэтчера и объединить.

Маленькие подмассивы можно сортировать другим алгоритмом, который,

например, не требует рекурсивных вызовов.

3.2 Анализ эффективности чётно-нечётной сортировки с использованием методов параллельного программирования с применением технологий MPI

Как уже было упомянуто, алгоритм чётно-нечётной сортировки Бэтчера хорошо параллелится, проанализируем параллельный алгоритм. Основная идея параллельной реализации сортировки заключается в следующем: для начала выполняется разбиение массива элементов по всем процессам. Каждый процесс параллельно сортирует свой массив последовательно. После получения нескольких отсортированных массивов задача состоит в слиянии этих массивов в один, результирующий. Слияние будет проводиться итерационно: на первом шаге процессы делятся по два, и правый массив передаёт свои данные левому. Тот процесс сортирует их, используя функцию bond. На втором шаге также объединяем процессы, но только те, которые были левыми на первом шаге. Опять, также, правый процесс передаёт свои данные левому, а тот их сортирует. Это продолжается до тех пор, пока не останется один процесс. Его массив и будет результатом параллельной сортировки.

3.2.1 Результаты вычислительных экспериментов

Для расчётов был использован процессор: Intel core i5-2300, тактовая частота 3 GHz.

Таблица 3.1 – Время выполнения сортировки в секундах

Количество элементов массива	Последовательный алгоритма	Параллельный алгоритм (2 процесса)	Параллельный алгоритм (4 процесса)
10 000	0,005135	0,002831	0.00064
50 000	0,030281	0,002719	0,004514
100 000	0,040144	0,014823	0,008167
500 000	0,165812	0,198584	0,046134
2 000 000	0,352912	0,292283	0,088729
5 000 000	1,839112	0,967221	0,732816
10 000 000	4,169191	1,923714	1,341935

3.2.2 Анализ эффективности

Трудоёмкость выполнения последовательного алгоритма сортировки

слияния равна O(n×log$_2$(n)).

Определим трудоёмкость параллельного алгоритма. После передачи элементов всем процессам они выполняют сортировку последовательным алгоритмом слияния. Значит трудоёмкость равна t1 = (n/p)×log2(n/p).

Определим трудоёмкость метода при слиянии массивов из процессов. Число итераций равно p-1, а в каждой итерации число перестановок равно отношению количества элементов в изначальном массиве n к используемому количеству процессов на данной итерации. На первом шаге их p, на втором $p/2$ и т.д. То есть трудоёмкость t_2 равна $t_2 = n/p + 2n/p + ... + n = n(2p-1)/p$

Общее время всех выполняемых в ходе сортировки операций передачи блоков можно оценить при помощи соотношения: $T_p(comm) = p(a + w(n/p)/b),$ где a – латентность, b – пропускная способность сети передачи данных, w – размер элемента упорядочиваемых данных в байтах.

Окончательная оценка алгоритма T_p равна

$$T_p = T_p(comm) + t_1 + t_2 = p(a + w(n/p)/b) + (n/p) + n(2p-1)/p$$

Общая оценка показателя ускорения равна:

$$S_p = n\log 2n / \left[p(a + w(n/p)b) + (n/p)\log 2(n/p) + n(2p-1)/p \right]$$

Оценка эффективности:

$$E_p = n\log 2n / p\left[p(a + w(n/p)/b) + (n/p)\log 2(n/p) + nm(2p-1)/p \right]$$

3.3 Декомпозиция расчётных сеток

Постановка задачи.

Для простоты будем считать, что процессоры у нас одноядерные. Вычислительная система с распределенной памятью, поэтому будем использовать технологию MPI. На практике в таких системах процессоры многоядерные, а значит для максимально эффективного использования вычислительных ресурсов следует также использовать треды.

Сетки мы ограничим лишь регулярными двумерными, т.е. такими, в которых узел с индексами (i, j)соединен с соседними существующими по i, j

узлами: (i-1, j), (i+1, j), (i, j-1), (i, j+1). Как нетрудно заметить, на верхней картинке сетка не регулярна, примеры регулярных сеток изображены ниже, рисунок 3.2.

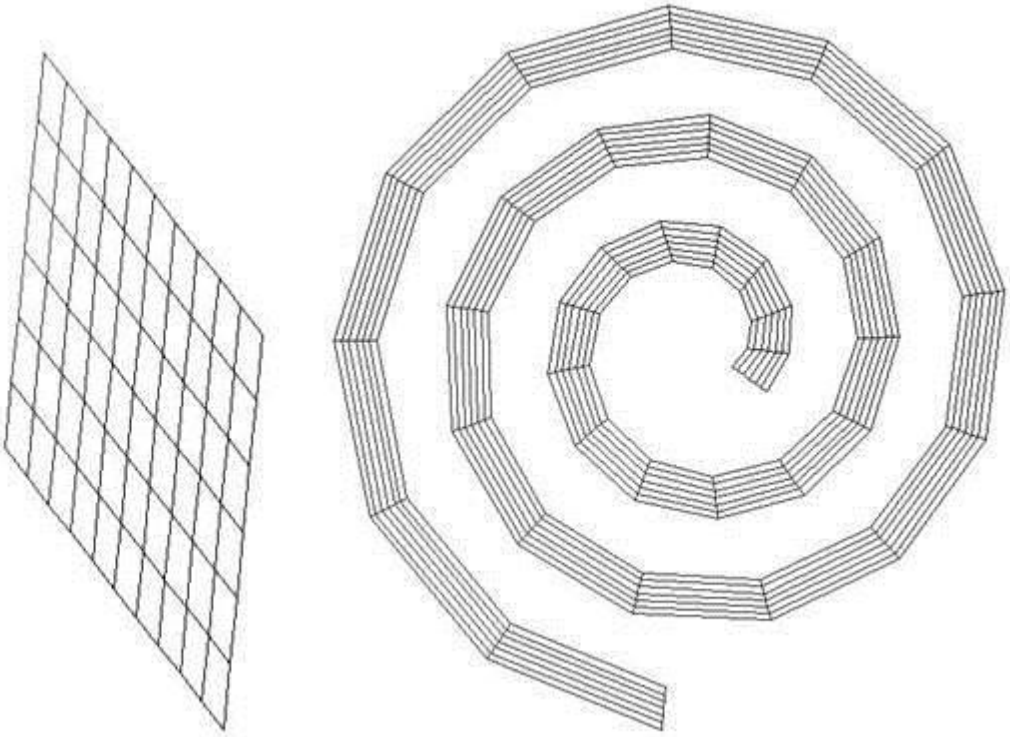

Рисунок 3.2 – Пример регулярных сеток

Благодаря этому можно очень компактно хранить топологию сетки, что позволяет существенно снизить как расход памяти, так и время работы программы (ведь нужно пересылать куда меньше данных по сети между процессорами).

На вход программе поступают 3 аргумента:

– n1, n2 – размеры двумерной сетки;

– k – число доменов (частей) на которое требуется разбить сетку.

Инициализация координат происходит внутри программы.

Один узел сетки мы будем хранить в следующей структуре:

```
struct Point{
  float coord[2]; // 0 – x, 1 – y
    int index;
};
```

Помимо координат точки здесь еще есть индекс узла, который имеет 2 назначения. Во-первых, его одного достаточно для восстановления топологии сетки. Во-вторых, с помощью него можно отличать фиктивные элементы от нормальных (например, у фиктивных элементов значение этого поля установить равным – 1). Про то, что это за фиктивные элементы и зачем они нужны, будет подробно рассказано далее.

Сама сетка хранится в массиве Point P[n1×n2], где узел с индексами (i, j) находится в P[i×n2+j]. В результате работы программы число вершин в доменах должно быть равно с точностью до одной вершины величине (n1×n2/k).

3.3.1 Алгоритм решения

Процедура рекурсивной координатной бисекции состоит из 3 шагов:

– сортировка массива узлов вдоль одной из координатных осей (в двумерном случае вдоль *x* или *y*);

– разбиение отсортированного массива на 2 части;

– рекурсивный запуск процедуры для полученных подмассивов.

Базисом рекурсии здесь является случай k = 1 (остался 1 домен). Происходит примерно следующее:

3.3.2 Деревья разбиений

В процессе параллельной геометрической декомпозиции сетки инициализируется построение глобального дерева распределения вершин по процессорам и, при необходимости, локальных деревьев разбиения вершин на домены в рамках каждого из процессоров. В двоичных деревьях разбиения каждая вершина содержит данные о медиане разбиения на две части на данном этапе рекурсии. Данные включают в себя координаты медианы, номер вершины медианы и номер координаты, по которой производится разбиение в данный момент. В глобальном дереве каждый «лист» содержит в поле номера координаты «-1», а в поле номера вершины – номер процессора, который дальше продолжит деление. Лист соответствует шагу, когда группа процессоров содержит только один процессор, и дальше соответствующая ему группа вершин будет разбиваться на домены локально. В локальном дереве разбиения каждый

лист содержит в поле номера координаты «-1», а в поле номера вершины – номер домена, в который попали данные вершины. В процессе выполнения параллельной декомпозиции сетки на каждом процессоре создается глобальное дерево разбиения вершин по процессорам и массив данных локальных деревьев разбиений вершин по доменам на каждом из процессоров. Для того чтобы определить, в какой домен попадет вершина при полученном разбиении, сначала инициализируется поиск в глобальном дереве разбиения вершин по процессорам. В каждой вершине дерева производится сравнение искомой вершины с медианой и, в зависимости от результата, дальнейший поиск производится либо в левом поддереве, либо в правом. Найденный лист позволяет определить, в локальном дереве какого процессора следует продолжить поиск. В локальном дереве поиск производится аналогично, а найденный лист дает информацию о номере домена искомой вершины. Создание деревьев было включено для случая разбиения различных сеток на одинаковые параллелепипеды, что потребовалось при параллельной интерполяции значений с одной сетки на другую. Интерполяция используется в задачах аэроакустики, когда часть расчета производится на грубой сетке, а продолжение – на измельченной сетке, что позволяет сократить время расчета.

Определение количества доменов, формируемых на процессорах.

В параллельном алгоритме геометрической декомпозиции реализованы два варианта определения количества доменов, формируемых на процессорах, и числа вершин, принадлежащих каждому из доменов. Первый вариант вызывается, когда геометрическая декомпозиция используется в качестве самостоятельного метода декомпозиции вершин на домены равного размера. Второй вариант вызывается, когда геометрическая декомпозиция используется в качестве предекомпозиции вершин по процессорам в другом алгоритме, например, в параллельном инкрементном алгоритме декомпозици процессоры с меньшими номерами будут формировать kpart / nproc + 1 домен с nc вершинами, а с большими номерами – kpart / nproc доменов. Домены с nc + 1 вершинами располагаются в обратном порядке: процессоры с большими номерами будут

создавать kpart1 / nproc+1 домен с nc+1 вершинами, а с меньшими номерами – kpart1 / nproc доменов. В соответствии с количеством доменов каждого вида, подсчитывается, сколько вершин должно быть на каждом из процессоров. При определении количества доменов во втором варианте n вершин разбивается на npart доменов (будущие процессоры) для будущего формирования на них ng микродоменов. Разбиение производится на nproc процессорах. Количество микродоменов, равное n % ng будет содержать n / ng + 1 вершину, остальные микродомены – n / ng вершин. Обозначим n / ng как nc. Число доменов, равное ng % npart будет содержать ng / npart + 1 микро– домен, а остальные – ng / npart микродоменов. Обозначим ng / npart как np. Микродомены по доменам распределяются следующим образом: в доменах с меньшими номерами будет содержаться np+1 микродомен, а с большими – np микродоменов. При этом домены с меньшими номерами сначала заполняются микродоменами, содержащими nc вершин, а после распределения всех микродоменов с nc вершинами, домены заполняются микродоменами, содержащими nc+1 вершину. В результате получатся домены четырех типов. Домены нулевого типа будут содержать np+1 микродомен с nc вершинами. Домены третьего типа будут содержать np микродоменов с nc+1 вершиной. С доменами первого и второго типа возможны варианты в зави– симости от количества микродоменов с nc вершинами. Если микродоменов с nc вершинами меньше, чем нужно для доменов, содержащих np+1 микродомен, то один домен первого типа будут содержать часть микродоменов с nc вершинами и часть с nc+1 вершиной, всего np+1 микродомен. Домены второго типа будут содержать np+1 микродомен с nc+1 вершиной. Если микродоменов с nc вершинами ровно столько, сколько нужно для доменов, содержащих np+1 микродомен, то доменов первого и второго типа не будет. А если микродоменов с nc вершинами больше, чем нужно для доменов, содержащих np+1 микродомен, то домены первого типа будут содержать np микродоменов с nc вершинами, а один домен второго типа будет содержать np микродоменов, часть из которых с nc вершинами, а часть с nc+1 вершиной. Предполагается, что алгоритмами, использующими геометрическую

предекомпозицию, поддерживается аналогичное распределение микродоменов по доменам на тот момент доменам по процессорам, при котором на процессорах с меньшими номерами формируется больше доменов, чем на процессорах с большими номерами. Домены распределяются по nproc процессорам равномерно, npart % nproc процессоров с меньшими номерами будет содержать npart / nproc+1 домен, а остальные процессоры – npart / nproc доменов. Заполнение процессоров доменами происходит, начиная с нулевого процессора. Сначала распределяются домены нулевого типа, потом первого, второго, и только в конце третьего типов. Подсчитывается, сколько вершин должно быть на каждом из процессоров. Для каждого процессора вычисляется начальный номер gr0 номеров доменов, формируемых на нем.

3.3.3 Рекурсивная координатная бисекция вершин по процессорам

Массив данных, используемый в параллельном алгоритме геометрической декомпозиции, содержит в себе наборы данных о каждой вершине, включающие координаты и номер вершины. На каждом этапе рекурсивной координатной бисекции вершин по процессорам происходит следующее:

– определяются границы наименьшего параллелепипеда, охватывающего все вершины, которые делятся на данном этапе. Вычисляются минимумы и максимумы по каждой из координат. Определяется координата j, вдоль которой параллелепипед имеет наибольшую протяженность;

– если на предыдущем шаге рекурсии вершины сортировались по другой координате, то выполняется параллельная сортировка вершин по координате *j*. В противном случае вершины только сдвигаются по процессорам для получения нужного количества вершин на каждом из процессоров. При делении группы из nproc процессоров в левой группе процессоров (процессоры с меньшими номерами, которые получат вершины меньшей координатой *j*) окажется nproc1 = nproc / 2 процессоров, остальные nproc – nproc1 процессоров попадут в правую группу. И при сортировке, и при сдвиге подсчитывается, сколько всего вершин должно быть в доменах на nproc1 процессорах. Данное количество вершин распределяется равномерно среди nproc1 процессоров. Когда nproc1 станет

равным единице, на процессор попадает итоговое количество вершин, определенное заранее. Распределение вершин среди процессоров в правой группе происходит аналогично;

– вблизи медианы выполняется сортировка вершин по всем координатам. Медиана проходит между процессорами с номерами nproc1 – 1 и nproc1. На процессоре с номером nproc1–1 выделяются вершины с наибольшей координатой j, которые расположены у правого края массива данных. На процессоре с номером nproc1 выделяются вершины с наименьшей координатой j, которые расположены у левого края массива данных. Процессоры обмениваются информацией о вершинах, и суммарные массивы отобранных вершин сортируются по всем координатам локально. Затем процессор с номером nproc1 – 1 отбирает себе нужное количество вершин с левого края отсортированного массива, а процессор с номером nproc1 – с правого края массива. При сортировке по всем координатам при сравнении данных о двух вершинах сравниваются их координаты в циклическом порядке, начиная с координаты j, до нахождения первого расхождения. Если все координаты оказались равными, сравниваются глобальные номера вершин, и вершина с меньшим номером считается меньше;

– процессором с номером nproc1 – 1 запоминаются данные о медиане этого этапа для дальнейшей сборки глобального дерева разбиения если в параллельный алгоритм геометрической декомпозиции поданы параметры, отвечающие за построение деревьев. Медиана является самой последней вершиной в массиве данных этого процессора, значение ее координаты j является наибольшим на процессоре. Запоминаются координаты медианы, номер вершины медианы и координата j, по которой происходило деление на данном этапе;

– создаются две новые группы процессоров с nproc1 и nproc – nproc1 процессорами. Процессоры с номерами от 0 до nproc1 – 1, включительно, помещаются в левую группу, остальные процессоры – в правую группу. Создаются новые коммуникаторы, и процессоры получают новые

идентификаторы в них. На следующем этапе каждая группа процессоров будет делить свой блок данных. После завершения рекурсивной координатной бисекции вершин по процессорам, если предполагается построение деревьев, на нулевом процессоре строится глобальное дерево разбиения по процессорам. Со всех процессоров собирается информация о сохраненных на них медианах. Полученная информация сортируется в порядке следования вершин двоичного дерева с помощью рекурсии. Медиана, соответствующая вершине дерева с номером i, была получена от процессора с номером ipf + nproc1 – 1, где nproc1 = nproc / 2. Она соответствует моменту, когда делилась группа из nproc процессоров с номерами, большими, либо равными, ipf. Для i, равного единице, ipf = 0. 82 Медиана вершины с номером 2 · i находится аналогично для группы из nproc1 процессоров с номерами, начинающимися с ipf. Медиана вершины с номером $2 \cdot i+1$ вычисляется для группы из nproc – nproc1 процессоров с номерами, начинающимися с ipf + nproc1. И далее по рекурсии. В момент, когда nproc1 станет равным единице, в лист дерева будет записана информация о то, что дальнейшее деление продолжит процессор с номером ipf. Для nproc – nproc1, равного единице, в лист глобального дерева будет записан процессор с номером ipf + nproc1. Запоминается максимальный номер вершины в глобальном дереве. Глобальное дерево разбиения по процессорам и максимальный номер вершины в нем, рассылаются на все процессоры.

3.3.4 Параллельный алгоритм

Алгоритм декомпозиции, представленный в работе, основан на методе рекурсивной координатной бисекции [19]. На каждом этапе область разбивается на две части. Сопоставление размеров частей основывается на количестве доменов, которые будут образованы в каждой из частей. После полученные области разбиваются подобным образом до тех пор, пока в подобластях не останется по одному домену. В алгоритме рекурсивной координатной бисекции на этапе разбиения определяется координатная ось, на протяжении которой область имеет большую протяженность. Область разбивается перпендикулярно оси. Алгоритм работает не только с координатами вершин, но и не учитывает

связи между ними, что делает его более экономичным по памяти.

Основные этапы алгоритма

– начальное распределение вершин по процессорам происходит в соответствии с порядковыми номерами;

– определение количества доменов, которые будут сформированы на каждом из процессоров, и количества процессоров;

–рекурсивная координатная бисекция вершин по процессорам. Сначала определяется координатная ось, на протяжении которой область имеет наибольшую длину. Затем блок вершин делится на две части перпендикулярно оси. Группа процессоров так же делится на две, затем каждая из групп процессоров делит свой блок вершин таким же образом. Для разделения блока вершин используется сортировка слиянием Бэтчера. После рекурсивной бисекции вершин по процессорам на процессорах оказывается то количество вершин, которое должно быть в образуемых на них доменах.

3.3.5 Рекурсивная координатная бисекция вершин

Массив данных, используемый в параллельном алгоритме декомпозиции, содержит в себе данные о каждой вершине, а именно координаты и номер вершины. На каждом этапе рекурсивной координатной бисекции вершин по процессорам происходит следующее:

– определяются границы наименьшего параллелепипеда, включающего в себя все вершины, которые делятся на данном этапе. Вычисляются минимальные и максимальные значения по каждой из координат. Определяется координата *j*, на протяжении которой параллелепипед имеет наибольшую длину;

– если на предыдущем шаге рекурсии вершины сортировались по другой координате, выполняется параллельная сортировка вершин по координате j. В противном случае вершины только сдвигаются по процессорам для получения необходимого количества вершин на каждом из процессоров. При делении группы из nproc процессоров в группе процессоров с меньшими номерами, которые получат вершины меньшей координатой j окажется nproc1 = nproc / 2 процессоров, остальные nproc – nproc1 процессоров попадут в другую группу.

При сортировке, и при сдвиге подсчитывается, необходимое количество вершин в доменах на nproc1 процессорах. Количество вершин распределяется равномерно среди nproc1 процессоров. Когда nproc1 станет равно единице, на процессор попадает итоговое количество вершин, определенное ранее;

– вблизи медианы выполняется сортировка вершин по всем координатам. Медиана проходит между процессорами с номерами nproc1 – 1 и nproc1. На процессоре с номером nproc1 – 1 определяются вершины с наибольшей координатой j, которые расположены у правого края массива. На процессоре с номером nproc1 выделяются вершины с наименьшей координатой j, которые расположены у левого края массива данных. Процессоры передают информацию о вершинах, и суммарные массивы отобранных вершин сортируются по всем координатам локально. После процессор с номером nproc1–1 отбирает себе необходимое количество вершин с левого края отсортированного массива, а процессор с номером nproc1 – с правого края. При сортировке по всем координатам при сравнении данных о двух вершинах сравниваются их координаты в порядке цикла, начиная с координаты j, до нахождения первого несоответствия. Если все координаты оказываются равными, сравниваются глобальные номера вершин, и вершина у которой номер меньше считается меньше;

– процессором с номером nproc1 – 1 сохраняет данные о медиане этого этапа для дальнейшей сборки глобального дерева разбиения. Медиана является последней вершиной в массиве данных этого процессора, значение ее координаты j является самым большим на процессоре. Сохраняются координаты медианы, номер вершины медианы и координата *j*, по которой происходило деление на данном этапе;

– формируются две новые группы процессоров с nproc1 и nproc – nproc1 процессорами. Процессоры с номерами от 0 до nproc1 – 1, включительно, помещаются в левую группу, остальные – в правую группу. Создаются новые коммуникаторы, и процессоры получают новые идентификаторы в них. На следующем шаге каждая группа процессоров будет разделять свой блок данных.

После завершения рекурсивной координатной бисекции вершин по процессорам, если необходимо построение деревьев, на нулевом процессоре строится глобальное дерево разбиения по процессорам. Со всех процессоров собирается информация о записанных на них медианах. Полученные данные сортируется в порядке следования вершин двоичного дерева с помощью рекурсии. Медиана, соответствующая вершине дерева с номером i, была переданна от процессора с номером ipf + nproc1 – 1, где nproc1 = nproc / 2. Она соответствует моменту деление группы из nproc процессоров с номерами, большими, либо равными, ipf. Для i, равного единице, ipf = 0. 82 Медиана вершины с номером 2 · i находится аналогично для группы из nproc1 процессоров с номерами, начинающимися с ipf. Медиана вершины с номером 2 · i + 1 находится для группы из nproc – nproc1 процессоров с номерами, начинающимися с ipf + nproc1. И далее по рекурсии. В момент, когда nproc1 станет равным единице, в лист дерева будет сохранена информация о то, что дальнейшее деление продолжит процессор с номером ipf. Для (nproc – nproc1), равного единице, в лист глобального дерева будет записан процессор с номером ipf + nproc1. Сохраняется максимальный номер вершины в глобальном дереве. Глобальное дерево разбиения по процессорам и максимальный номер вершины в нем, рассылаются на все процессоры.

3.4 Программная реализация

Код реализован на языке программирования C++. Версия MPI 8.0.

Для анализа эффективности алгоритма был реализован программный код, представленный в Приложении А. Программа запускалась на локальном компьютере Intel core i5-2300, тактовая частота 3.00 GHz. Программа принимает на вход количество процессоров на которые будет распараллеливается алгоритм, количество доменов биекции, а так же размер массива данных и название файла для записи декомпозированной расчётной сетки.(рисунок 3.3) Здесь «mpiexec» инициализация MPI на локальном компьютере, «-np» количество процессоров, на которые будет распараллелен алгоритм, «cb» название файла, хранящего в себе программный код. В свою очередь «cb» принимает на вход <количество

доменов биекции> < размер массива в плоскости x> <размер массива в плоскости y> <название файла с полученными значениями>.

mpiexec -np 4 cb 4 5 5 a.out

Рисунок 3.3 – Инициализация программного кода

Результатом работы программы является файл с данными декомпозированной сетки.

Так же реализован программный код для представления полученных данных в понятной форме (приложение В) и возможностью экспорта данных при необходимости.

> view \c++\cordbisect\cb\debug\a.out

Рисунок 3.4 – Инициализация программы viev

Программа «viev» принимает на вход путь к файлу, с хранящимися данными полученными в результаты компиляции программы «cb».

Результатом работы программы являются данные, показанные в следующей последовательности <координата массива i> < координата массива j> <координата x> <координата y> <номер домена биекции>

Код компилировался при помощи пробной версии visual studio. Библиотека mpi является свободно распространяемой.

```
1  3  -44494.460938  -47943.050781  0
4  3  -40221.871094  -47784.355469  0
0  4  -44894.253906  -22634.052734  0
3  0  -41192.359375  -22435.683594  0
0  0  -14909.818359  -13432.723633  0
0  3  -18507.951172  2088.993896  0
2  1  -43337.808594  2317.881592  1
1  2  -14870.142578  17775.505859  1
4  1  -15282.142578  25499.744141  1
3  2  -41778.312500  27269.816406  1
3  1  -42519.914063  30623.798828  1
2  4  -26424.451172  45800.648438  1
0  1  37441.636719  -41683.707031  2
2  0  19487.595703  -33645.128906  2
3  4  -14790.793945  -19814.142578  2
0  2  6987.213867  -18303.474609  2
4  2  -4527.419922  -5110.323242  2
1  0  39443.648438  -1753.288452  2
3  3  35830.253906  -679.036987  3
4  4  34026.609375  -141.909714  3
1  4  32900.476563  29631.949219  3
1  1  41982.789063  34658.343750  3
4  0  -8354.443359  39544.355469  3
2  3  -14830.469727  48980.683594  3
2  2  37636.953125  -1216.161255  3
```

Рисунок 3.5 – Результаты выполнения программы viev

На рисунке 3.6 визуально представлен результат декомпозиции 10^9 элементов на 10 доменов бисекции при помощи ППП MATLAB, код представлен в приложении С.

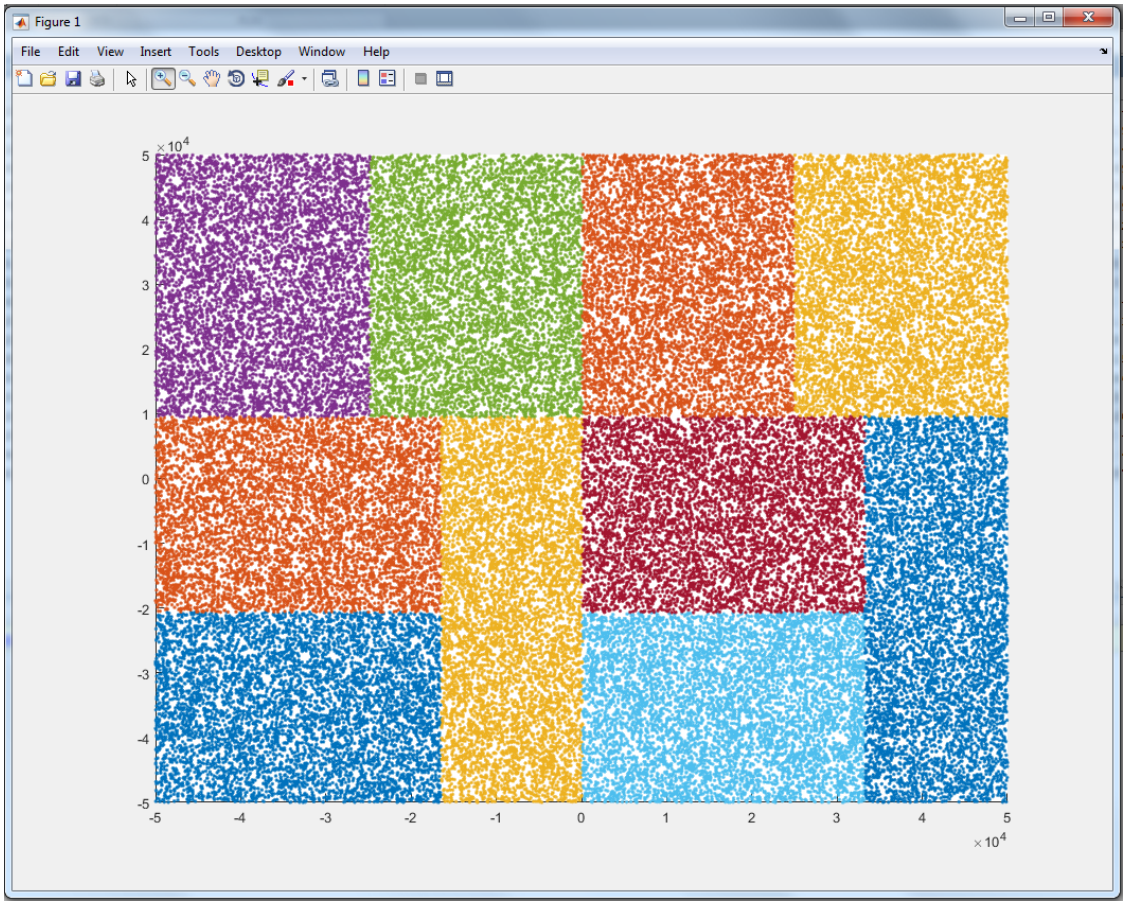

Рисунок 3.6 – Декомпозиция 10^9 элементов на 10 доменов бисекции

3.5 Анализ полученных результатов

Для расчётов был использован процессор: Intel core i5-2300, тактовая частота 3.00 GHz. Ввиду архитектуры процессора, время работы параллельного алгоритма на двух и трёх ядрах не сильно отличается. Показательными являются результаты, полученные для алгоритма распараллеленного на 4 ядра, и именно эти результаты применимы на практике, так как не имеет никакого смысла запускать расчеты, не используя всю мощность процессора. Из таблицы 3.2 видно, что выигрыш в скорости начинается при разбиении на 5 доменов и размере сетки 500×500. В свою очередь при более простых расчётах, потеря во времени не превышает одной сотой секунды (0,083 секунды при бисекции на 5

доменов сетки размером 50×50). При разбиении сетки размером 5000×5000 на 100 доменов выигрыш в скорости составляет 82 секунды.

Таблица 3.2 – Время работы программы

Кол-во доменов бисекции	Размер сетки	Последовательный алгоритм	Параллельный алгоритм 2 процессора	Параллельный алгоритм 3 процессора	Параллельный алгоритм 4 процессора
5	50×50	0,025	0,058	0,066	0,108
5	500×500	0,47	0,429	0,383	0,331
5	2500×2500	13,166	8,687	7,922	6,291
5	5000×5000	53,715	35,599	32,859	25,943
20	50×50	0,026	0,032	0,074	0,092
20	500×500	0,858	0,525	0,526	0,385
20	2500×2500	20,942	12,028	11,19	6,767
20	5000×5000	89,33	48,816	45,867	28,823
100	50×50	0,029	0,036	0,078	0,095
100	500×500	1,262	0,715	0,72	0,492
100	2500×2500	29,863	16,846	16,001	9,413
100	5000×5000	120,391	67,076	64,729	38,303

3.6 Использование результатов выпускной квалификационной работы в педагогической деятельности

В данной работе представлен широкий обзор методов сортировки, а так же алгоритмов бисекции основанных на параллельном программировании. Подробно описана технология MPI.

На основе представленной работы было составлено лекционное занятие по дисциплине параллельные вычислительные системы

Вводная часть лекции основана на главе 1.3 представленной диссертации. Основными пунктами вводной части лекции являются: преимущества модели с передачей данных, стандартизация модели передачи данных MPI, функции передачи сообщений MPI

Основные принципы и пример:

В качестве примера будет использоваться расчет экспоненты. Один из вариантов ее нахождения – ряд Тейлора:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

Представленная формула достаточно просто паралелится, так как число которое требуется найти является суммой отдельных слагаемых в последствии чего каждый отдельный вычислитель имеет возможность сфокусироваться на расчёте отдельных слагаемых.

Общее число слагаемых, которое будет вычисляется в каждом отдельно взятом вычислителе, зависит от двух факторов, а именно от общей длины интервала *n*, и от общего числа процессоров *k*, которые будут принимать участие в процессе расчётов. Так, для примера, если длина промежутка равна четырём, а в расчётах участвуют десять вычислителей, то с первого по четвертый процессоры получат по одному слагаемому, а пятый будет не задействован. В случае же если n=10, а k=5, каждому вычислителю передаётся по два слагаемых для расчётов.

Изначально, первый вычислитель при помощи функции широковещательной рассылки MPI_Bcast переносит остальным значение заданной пользователями переменной n. В общем случае функция MPI_Bcast принимает следуюущий вид: int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm), где buffer – это адрес буфера с элементом, count – количество элементов, datatype – соответствующий тип данных в MPI, root – ранг главного процессора, который занимается передачей, а comm- имя коммуникатора.

В представленном случае в роли основного процессора, как уже говорилось, будет выступать первый процессор с рангом 0.

В следствии того число n будет передано, каждый вычислитель примется рассчитывать свои слагаемые. Для этого потребуется в каждом шаге цикла к числу *i*, которое изначально равно рангу процессора, будет добавляется число, равное числу вычислителей участвующих в вычислениях. В случае когда в процессе последующих итераций число i превысит заданное изначальное число n, выполнение цикла для данного процессора завершится.

В процессе выполнения алгоритма слагаемые будут переносится в отдельную переменную и, в следствии его окончания, полученная сумма отправится в главный процессор. Для этого используется функция операции приведения MPI_Reduce. В общем виде она выглядит следующим образом: int MPI_Reduce(void *buf, void *result, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Она объединяет элементы входного буфера каждого процесса в группе, используя операцию op, и возвращает объединенное значение в выходной буфер процесса с номером root. В результате такой операции будет однозначное значение, благодаря чему функция приведения и получила свое название.

В последствии исполнения алгоритма на всех вычислителях, первый процессор получит сумму слагаемых, которая и будет являться искомым нами значение экспоненты.

Важно обратить внимание на то, что и в параллельном и последовательном методах вычисления экспоненты, для нахождения факториала используется рекурсивная функция. В процессе принятия решения по принципу распараллеливания поставленной задачи, был изучен вариант нахождения факториала также на разных процессорах, но в итоге такой вариант оказался не рациональным.

Главной задачей все же является поиск значения экспоненты и в случае когда процессоры начнут вычислять каждый факториал каждого слагаемого раздельным образом, это может привести к прямо обратно эффекту, а именно весомой потери в производительности и скорости вычисления.

Объяснить это можно тем, что в этом случае начнется весьма большая нагрузка на коммуникационную среду, которая и без того зачастую является слабым звеном в системах параллельных вычислений. Если же вычисление факториала будет происходить на каждом процессоре частным образом, нагрузка на линии коммуникаций будет минимизирована. Представленный случай можно назвать хорошим примером того, что и задача распараллеливания тоже должна порой иметь свои границы.

Алгоритм выполнения кода:

1. Из визуальной оболочки в программу заносится значение числа n, которое в следствии с помощью функции широковещательной рассылки передаётся по всем вычислителям.

2. При активации первого основного процессора, активируется таймер.

3. Каждый процессор выполняет цикл, где значением приращения является число процессоров в системе. В каждой итерации цикла расчитывается слагаемое и сумма таких слагаемых сохраняется в переменную drobSum.

4. После завершения цикла каждый процессор суммирует свое значение drobSum к переменной Result, применяя для этого функцию приведения MPI_Reduce.

5. После завершения вычислений на всех процессорах, первый процессор останавливает таймер и отправляет в поток вывода получившееся значение переменной Result.

6. В поток вывода отправляется и отмеренное нашим таймером значение времени в миллисекундах.

Листинг кода: приложение С.

Программа написана на C++, будем считать что аргументы для выполнения передаются из внешней оболочки.

Как результат мы получили простой код для расчёта экспоненты с применением сразу нескольких процессоров.

Задание для самостоятельного выполнения: установить библиотеку MPI и скомпилировать представленный в лекции алгоритм. Попытается найти слабые стороны представленного алгоритма, и предложить способы переработки.

Указание: слабым местом является хранением самого результата, введу того, что с увеличением количества разрядов вмещать значение с применением стандартных типов просто на просто не выйдет и это место требует проработки. Пожалуй, самым логичным решением является запись полученных результата в сторонний файл, хотя, в виду чисто учебной функции этого примера, особо на этом внимание можно не акцентировать.

ЗАКЛЮЧЕНИЕ

Проанализированы алгоритмы сортировки баз данных для параллельных вычислительных систем на основе технологии MPI. Найден оптимальный алгоритм сортировки для реализации декомпозиции расчётной сетки на основе параллельной вычислительной системы. Проанализирован алгоритм декомпозиции расчётных сеток, для расчётных сеток разных размеров от 50x50 до 5000x5000 доменов на разном количестве параллельно задействованных вычислителей. Получено визуальное представление результатов при помощи ППП MATLAB, показывающее правильность работы алгоритма декомпозиции, а так же справедливость разбиения области. Изучена и внедрена технология внесения псевдоэлементов массива, для повышения корректности разбиения сетки. На основе выпускной работы разработан план лекции по дисциплине параллельные вычислительные системы с примерами и индивидуальными заданиями.

На основе исследования создан программный код декомпозиции расчётных сеток на параллельной вычислительной среде, скорость работы, по сравнению с последовательным алгоритмом увеличена более чем в три раза (с 120,391 секунд до 38,303 секунд) для массива размером 5000x5000 элементов и декомпозиции на 100 доменов.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1 Бройдо, В. Вычислительные системы, сети и телекоммуникации / В. Бройдо, О. Ильина. – СПб.: Питер, 2011. – 560 с.

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

3 Гришагин, В.А. Параллельное программирование на основе MPI / В.А. Гришагин, А.Н. Свистунов. – Нижний Новгород: Изд-во ННГУ им. Н.И. Лобачевского, 2005. – 93 с.

4 Дейкстра, Э. Взаимодействие последовательных процессов [Электронный ресурс] // Национальный технический университет ХПИ : офиц. сайт. – Режим доступа : http://khpi-iip.mipk.kharkiv.edu/library/extent/ dijkstra/ewd123/ index.html. – 22.04.2018.

5 Илюшин, А.И. Построение параллельной вычислительной модели путем композиции вычислительных объектов / А.И. Илюшин, А.А. Колмаков, И.С. Меньшов // Математическое моделирование. – 2011. – Т. 23. – № 7. – С. 97 – 113.

6 Кнут, Д.Э. Искусство программирования. В 4 т. / Д.Э. Кнут. – М.: Вильямс – Т.3: Сортировка и поиск. – 832 с.

7 Лацис, А.О. Параллельная обработка данных / А.О. Лацис. – М. : Академия, 2010. – 336 с.

8 Пакет прикладных программ MARPLE3D для моделирования на высокопроизводительных ЭВМ импульсной магнитоускоренной плазмы / В.А. Гасилов [и др.] // Математическое моделирование. – 2012. – Т. 24. – № 1. – С. 55-87.

9 Паттерсон, Д. Архитектура компьютера и проектирование компьютерных систем / Д. Паттерсон, Дж. Хеннеси. – СПб. : Питер, 2012. – 784 с.

10 Седжвик, Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск / Р. Седжвик. - СПб. : ДиаСофтЮП, 2002. – 688 с.

11 Тютляева, Е.О. Интеграция алгоритма параллельной сортировки Бэтчера и активной системы хранения данных / Е.О. Тютляева // Программные системы: теория и приложения. – 2013. – № 4. – С. 127-142.

12 Эндрюс, Г.Р. Основы многопоточного, параллельного и распределенного программирования / Г.Р. Эндрюс. – М.: Вильямс, 2003. – 512 с.

13 Языки программирования / ред. Ф. Женюи. – М. : Мир, 1972. – 408 с.

14 Якобовский М. В. Параллельные алгоритмы сортировки больших объемов данных [Электронный ресурс]: офиц. сайт. – Режим доступа : http://lira.imamod.ru/FondProgramm/Sort/ParallelSort.pdf. – 24.04.2018.

15 Якобовский, М. В. Введение в параллельные методы решения задач / М.В. Якобовский. – М. : Изд-во Московского гос. ун-та, 2013. – 328 с.

16 Baum J.D., Luo H., Lohner R., Yang C.,Pelessone D., Charman C. A Coupled Fluid/Structure Modeling of Shock Interaction with a Truck // AIAA-96-0795 2006.

17 Baum J.D., Luo H., Mestreau E., L5hner R., Pelessone D., Charman C. A coupled CFD/CSD Methodology for Modeling Weapon Detonation and Fragmentation // AIAA-99-0794 1999.

18 Baum J.D., Luo H., R. Lohner The Numerical Simulation of Strongly Unsteady Flows With Hundreds of Moving Bodies // AIAA-98-0788 -1998.

19 Farhat C., Lesoinne M. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics // Int. J. Numer. Meth. Engng 2003. - V.36. P. 745-764.

20 Karypis, G. METIS – A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of 161 Sparse Matrices. Version 5.0 / G. Karypis // University of Minnesota, Minneapolis. – 2011. – August 4.

21 Karypis, G. ParMETIS – Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.1 / G. Karypis, K. Schloegel, V. Kumar // University of Minnesota, Minneapolis. – 2003.

22 Kumar, V. Introduction to Parallel Computing / V. Kumar, A. Grama, A. Gupta. - The Benjamin : Cummings Publishing Company, Inc., 2003. – 856 с.

23 Lohner R., Yang C., Cebral J., Baum J.D., Luo H., Pelessone D., Char-man C. Fluid-Structure-Thermal Interaction Using a Loose Coupling Algorithm and Adaptive Unstructured Grids // AIAA-98-2419 1008.

24 Mestreau E., Lohner R. Airbag Simulations Using Fluid/Structure Coupling // AIAA-96-0798 1996.

25 Preis, R. PARTY – A Software Library for Graph Partitioning. Advances in Computational Mechanics with Parallel and Distributed Processing / R. Preis and R. Diekmann. – CIVIL-COMP PRESS, 1997. – P. 63-71.

26 Parallel Partitioning, Load Balancing and Data-Management Services. Developer's Guide, Version 3.3 / E. Boman [and others]. – Sandia National Laboratories, Copyright©2000-2010.

27 Pellegrini, F. PT-Scotch and libPTScotch 6.0 User's Guide / F. Pellegrini // Bacchus team, INRIA Bordeaux Sud-Ouest, Universite Bordeaux 1 & LaBRI, UMR CNRS 5800. – Talence, France, 2012. – 90 p.

28 Hassan O., Bayne L.B., Morgan K., Weatherill N.P. An Adaptive Unstructured Mesh Method for Transient Flows Involving Moving Boundaries // 5th US Congress on Computational Mechanics 1999 - P. 662674.

29 Казанцев А.А. Параллельный алгоритм для реализации метода рекусивной координатной бисекции для декомпозиции расчётных сеток / А.А. Казанцев // Молодёжь XXI века: шаг в будущее материалы XVIII региональной научно-практической конференции. 2017. – С. 1022-1023.

30 Казанцев А.А. Чётно-нечётная сортировка слиянием Бэтчера параллельный алгоритм/ А.А. Казанцев // Молодёжь XXI века: шаг в будущее материалы XVIII региональной научно-практической конференции. – С. 87-88.

ПРИЛОЖЕНИЕ А

Листинг программной реализация метода рекурсивной координатной бисекции для декомпозиции расчетных сеток со слиянием Бэтчера

```
#include <stdio.h>
#include <stdlib.h>
#include <vector>
#include <cmath>
#include <mpi.h>
#include <stddef.h>
#include <string.h>
#include <float.h>
#include <time.h>

struct Point{
   float coord[2];
   int index;
};

struct PointDomain{
   int i;
   int j;
   float coord[2];
   int domain;
};

struct Comparator{
   int first;
   int second;
   Comparator(int f, int s): first(f), second(s) {}
};

int axis = 1;
std::vector<Comparator> comparators;
MPI_Datatype MPI_POINT_TYPE;
MPI_Datatype MPI_POINT_DOMAIN_TYPE;

int compare_points(const void *a, const void *b)
{
   if(((Point*)a)->coord[axis] < ((Point*)b)->coord[axis]){
      return -1;
   }else if(((Point*)a)->coord[axis] > ((Point*)b)->coord[axis]){
      return 1;
   }else{
```

```
        return 0;
    }
}

void join(int *array1, int count1, int *array2, int count2)
{
    if(count1 + count2 == 1){
        return;
    }else if(count1 + count2 == 2){
        if(count1 == 2)
            comparators.push_back(Comparator(array1[0], array1[1]));
        else if(count2 == 2)
            comparators.push_back(Comparator(array2[0], array2[1]));
        else
            comparators.push_back(Comparator(array1[0], array2[0]));
    }else{
        int oddcount1 = count1 / 2;
        int evencount1 = count1 - oddcount1;
        int *even1 = new int[evencount1];
        int *odd1 = new int[oddcount1];
        for(int i = 0, j = 0, k = 0; i < count1; i++){
            if(i % 2 == 0){
                even1[j] = array1[i];
                j++;
            }else{
                odd1[k] = array1[i];
                k++;
            }
        }
        int oddcount2 = count2 / 2;
        int evencount2 = count2 - oddcount2;
        int *even2 = new int[evencount2];
        int *odd2 = new int[oddcount2];
        for(int i = 0, j = 0, k = 0; i < count2; i++){
            if(i % 2 == 0){
                even2[j] = array2[i];
                j++;
            }else{
                odd2[k] = array2[i];
                k++;
            }
        }
```

```
        join(even1, evencount1, even2, evencount2);
        join(odd1, oddcount1, odd2, oddcount2);
        int *res = new int[count1 + count2];
        for(int i = 0; i < count1; i++)
            res[i] = array1[i];
        for(int i = 0; i < count2; i++)
            res[count1 + i] = array2[i];
        for(int i = 1; i < count1 + count2 - 1; i += 2)
            comparators.push_back(Comparator(res[i], res[i+1]));
        delete [] res;
        delete [] odd1;
        delete [] even1;
        delete [] odd2;
        delete [] even2;
    }
}

void sort(int *array, int count)
{
    int size1 = count / 2;
    int size2 = count - size1;
    if(count > 1){
        int *array1 = new int[size1];
        int *array2 = new int[size2];
        for(int i = 0; i < size1; i++)
            array1[i] = array[i];
        for(int i = 0; i < size2; i++)
            array2[i] = array[size1 + i];
        sort(array1, size1);
        sort(array2, size2);
        join(array1, size1, array2, size2);
        delete [] array1;
        delete [] array2;
    }
}

void create_shedule(int start, int count)
{
    int *array = new int[count];
    for(int i = 0; i < count; i++)
        array[i] = start + i;
    sort(array, count);
```

```
    delete [] array;
}

float x(int i, int j)
{
    return ((float)rand()/(float)RAND_MAX * 100000.0 - 50000.0);
}

float y(int i, int j)
{
    return ((float)rand()/(float)RAND_MAX * 100000.0 - 50000.0);
}

void create_point_type()
{
    const int nitems = 2;
    int blocklengths[2] = {2, 1};
    MPI_Datatype types[2] = {MPI_FLOAT, MPI_INT};
    MPI_Aint offsets[2];

    offsets[0] = offsetof(Point, coord);
    offsets[1] = offsetof(Point, index);

    MPI_Type_create_struct(nitems, blocklengths, offsets, types,
        &MPI_POINT_TYPE);
    MPI_Type_commit(&MPI_POINT_TYPE);
}

void create_point_domain_type()
{
    const int nitems = 4;
    int blocklengths[4] = {1, 1, 2, 1};
    MPI_Datatype types[4] = {MPI_INT, MPI_INT, MPI_FLOAT, MPI_INT};
    MPI_Aint offsets[4];

    offsets[0] = offsetof(PointDomain, i);
    offsets[1] = offsetof(PointDomain, j);
    offsets[2] = offsetof(PointDomain, coord);
    offsets[3] = offsetof(PointDomain, domain);

    MPI_Type_create_struct(nitems, blocklengths, offsets, types,
        &MPI_POINT_DOMAIN_TYPE);
```

```
    MPI_Type_commit(&MPI_POINT_DOMAIN_TYPE);
}

Point* init_array(int n1, int n2, int rank, int proc_count)
{
    int elem_total = n1 * n2;
    int elem_per_proc_count = ceil(elem_total / (double)proc_count);
    int proc_without_fictive_elements = elem_total % proc_count;
    int temp = elem_total / proc_count;
    int elem_non_fictive = rank < proc_without_fictive_elements ?
        temp + 1 : temp;
    int start = rank < proc_without_fictive_elements ?
        rank * elem_per_proc_count : proc_without_fictive_elements *
        elem_per_proc_count + (rank - proc_without_fictive_elements) * temp;

    Point *array = new Point[elem_per_proc_count];
    int k = 0;
    for(; k < elem_non_fictive; k++){
        int i = (k + start) / n2;
        int j = (k + start) % n2;
        array[k].index = k + start;
        array[k].coord[0] = x(i, j);
        array[k].coord[1] = y(i, j);
    }
    for(; k < elem_per_proc_count; k++){
        array[k].index = -1;
        array[k].coord[0] = FLT_MAX;
        array[k].coord[1] = FLT_MAX;
    }
    return array;
}

void write_array(PointDomain *array, int n, char *filename, int n1, int n2,
        int rank)
{
    MPI_Status s;

    MPI_File output;
    if(MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_CREATE |
            MPI_MODE_WRONLY, MPI_INFO_NULL, &output) !=
MPI_SUCCESS){
        if(rank == 0)
```

```
            printf("File '%s' can't be opened.\n", filename);
        MPI_Finalize();
        exit(1);
    }
    MPI_File_set_size(output, 0);

    MPI_File_write_ordered(output, &n1, rank == 0, MPI_INT, &s);
    MPI_File_write_ordered(output, &n2, rank == 0, MPI_INT, &s);
    MPI_File_write_ordered(output, array, n, MPI_POINT_DOMAIN_TYPE, &s);

    MPI_File_close(&output);
}

void sort_array(Point*& array, int elem_per_proc_count, MPI_Comm communicator)
{
    MPI_Status s;
    int rank;
    MPI_Comm_rank(communicator, &rank);
    Point *from_another_proc = new Point[elem_per_proc_count];
    Point *tmp = new Point[elem_per_proc_count];
    qsort(array, elem_per_proc_count, sizeof(Point), compare_points);
    for(unsigned int ui = 0; ui < comparators.size(); ui++){
        Comparator comparator = comparators[ui];
        if(rank == comparator.first){
            MPI_Send(array, elem_per_proc_count, MPI_POINT_TYPE,
                comparator.second, 0, communicator);
            MPI_Recv(from_another_proc, elem_per_proc_count, MPI_POINT_TYPE,
                comparator.second, 0, communicator, &s);

            for(int i = 0, j = 0, k = 0; i < elem_per_proc_count; i++){
                Point a = array[j];
                Point b = from_another_proc[k];
                // a < b
                if(compare_points(&a, &b) == -1){
                    tmp[i] = a;
                    j++;
                }else{
                    tmp[i] = b;
                    k++;
                }
            }
```

```
            Point *temp = array;
            array = tmp;
            tmp = temp;
        }else if(rank == comparator.second){
            MPI_Recv(from_another_proc, elem_per_proc_count, MPI_POINT_TYPE,
                comparator.first, 0, communicator, &s);
            MPI_Send(array, elem_per_proc_count, MPI_POINT_TYPE,
                comparator.first, 0, communicator);

            int start = elem_per_proc_count - 1;
            for(int i = start, j = start, k = start; i >= 0; i--){
                Point a = array[j];
                Point b = from_another_proc[k];
                // a > b
                if(compare_points(&a, &b) == 1){
                    tmp[i] = a;
                    j--;
                }else{
                    tmp[i] = b;
                    k--;
                }
            }

            Point *temp = array;
            array = tmp;
            tmp = temp;
        }
    }
    delete [] from_another_proc;
    delete [] tmp;
}

void local_decompose(Point *a, int *domains, int domain_start, int k,
        int array_start, int n)
{
    if(k == 1){
        for(int i = 0; i < n; i++)
            domains[array_start + i] = domain_start;
        return;
    }

    axis = !axis;
```

```
    qsort(a + array_start, n, sizeof(Point), compare_points);

    int k1 = (k + 1) / 2;
    int k2 = k - k1;
    int n1 = n * (k1 / (double)k);
    int n2 = n - n1;
    local_decompose(a, domains, domain_start, k1, array_start, n1);
    local_decompose(a, domains, domain_start + k1, k2, array_start + n1, n2);
}

int remove_fictive(Point **array, int size)
{
    Point *res = new Point[size];
    int j = 0;
    for(int i = 0; i < size; i++){
        if((*array)[i].index == -1)
            continue;
        res[j++] = (*array)[i];
    }
    delete [] (*array);
    *array = res;
    return j;
}

Point* align(Point *array, int size, int proc_count)
{
    int epp = ceil(size / (double)proc_count);
    Point *res = new Point[epp * proc_count];
    int k = 0;
    for(int i = 0; i < size; i++, k++){
        res[k] = array[i];
        array[i].index = -1;
        array[i].coord[0] = FLT_MAX;
        array[i].coord[1] = FLT_MAX;
    }
    for(; k < epp * proc_count; k++){
        res[k].index = -1;
        res[k].coord[0] = FLT_MAX;
        res[k].coord[1] = FLT_MAX;
    }
    return res;
}
```

```
void decompose(Point **array, int **domains, int domain_start, int k, int n,
      int *elem_per_proc, MPI_Comm communicator)
{
   int rank, proc_count, actual_size;
   MPI_Comm_rank(communicator, &rank);
   MPI_Comm_size(communicator, &proc_count);
   if(proc_count == 1){
      actual_size = remove_fictive(array, *elem_per_proc);
      *domains = new int[actual_size];
      local_decompose(*array, *domains, domain_start, k, 0, actual_size);
      *elem_per_proc = actual_size;
      return;
   }
   if(k == 1){
      actual_size = remove_fictive(array, *elem_per_proc);
      *domains = new int[actual_size];
      for(int i = 0; i < actual_size; i++)
         (*domains)[i] = domain_start;
      *elem_per_proc = actual_size;
      return;
   }

   MPI_Status s;
   int k1 = (k + 1) / 2;
   int k2 = k - k1;
   int n1 = n * (k1 / (double)k);
   int n2 = n - n1;
   int pc = n1 / (*elem_per_proc);
   int middle = n1 % (*elem_per_proc);
   int color;
   if(pc == 0)
      color = rank > pc ? 0 : 1;
   else
      color = rank >= pc ? 0 : 1;

   axis = !axis;
   comparators.clear();
   create_shedule(0, proc_count);
   sort_array(*array, *elem_per_proc, communicator);
   MPI_Comm new_comm;
   MPI_Comm_split(communicator, color, rank, &new_comm);
```

```
if(pc == 0){
   int epp = ceil(((*elem_per_proc) - middle) / (double)(proc_count - pc));
   if(rank == pc){
      Point *temp = align((*array) + middle, (*elem_per_proc) - middle,
            proc_count - pc - 1);
      for(int i = pc + 1, j = 0; i < proc_count; i++, j++)
         MPI_Send(temp + j*epp, epp, MPI_POINT_TYPE, i, 0, communicator);
      delete [] temp;
      decompose(array, domains, domain_start, k1, n1, elem_per_proc,
            new_comm);
   }else{
      Point *arr = new Point[*elem_per_proc + epp];
      MPI_Recv(arr + (*elem_per_proc), epp, MPI_POINT_TYPE, pc, 0,
            communicator, &s);
      memcpy(arr, *array, (*elem_per_proc) * sizeof(Point));
      *elem_per_proc += epp;
      delete [] (*array);
      *array = arr;
      decompose(array, domains, domain_start + k1, k2, n2, elem_per_proc,
            new_comm);
   }
   return;
}
if(rank <= pc){
   int epp = ceil(middle / (double)pc);
   if(rank == pc){
      Point *temp = align(*array, middle, pc);
      for(int i = 0; i < pc; i++)
         MPI_Send(temp + i*epp, epp, MPI_POINT_TYPE, i, 0, communicator);
      delete [] temp;
   }else{
      Point *arr = new Point[*elem_per_proc + epp];
      MPI_Recv(arr + (*elem_per_proc), epp, MPI_POINT_TYPE, pc, 0,
            communicator, &s);
      memcpy(arr, *array, (*elem_per_proc) * sizeof(Point));
      *elem_per_proc += epp;
      delete [] (*array);
      *array = arr;
   }
}
if(rank < pc){
   decompose(array, domains, domain_start, k1, n1, elem_per_proc,
```

```
            new_comm);
    }else{
        decompose(array, domains, domain_start + k1, k2, n2, elem_per_proc,
            new_comm);
    }
}

int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank, proc_count;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &proc_count);
    if(argc != 5){
        if(rank == 0){
            printf("Usage: mpirun -np <proc_num> decompose <k> <n1> <n2> "
                "<output file>\n");
        }
        MPI_Finalize();
        return 0;
    }
    create_point_type();
    create_point_domain_type();
    int k = strtol(argv[1], NULL, 10);
    int n1 = strtol(argv[2], NULL, 10);
    int n2 = strtol(argv[3], NULL, 10);
    int elem_per_proc = ceil(n1 * n2 / (double)proc_count);
    srand(time(NULL) + rank * proc_count);

    // initialization
    Point *array = init_array(n1, n2, rank, proc_count);
    int *domains;

    // decomposition
    decompose(&array, &domains, 0, k, n1 * n2, &elem_per_proc,
MPI_COMM_WORLD);

    // writing
    int wcount = 0;
    PointDomain *res = new PointDomain[elem_per_proc];
    for(int i = 0; i < elem_per_proc; i++){
        if(array[i].index == -1)
```

```
            continue;
        res[wcount].coord[0] = array[i].coord[0];
        res[wcount].coord[1] = array[i].coord[1];
        res[wcount].i = array[i].index / n2;
        res[wcount].j = array[i].index % n2;
        res[wcount].domain = domains[i];
        wcount++;
    }
    delete [] domains;
    delete [] array;
    write_array(res, wcount, argv[4], n1, n2, rank);
    delete [] res;

    MPI_Finalize();
    return 0;
}
```

ПРИЛОЖЕНИЕ Б

Листинг кода корректного вывода данных результатов программы представленной в приложении А

```
#include <stdio.h>

int main(int argc, char **argv)
{
   if(argc != 2){
      printf("Usage: ./view <file>\n");
      return 1;
   }

   FILE *input = fopen(argv[1], "rb");
   if(input == NULL){
      printf("File '%s' can't be opened.\n", argv[1]);
      return 1;
   }

   int n1, n2, i, j, domain;
   float x, y;
   fread(&n1, sizeof(n1), 1, input);
   fread(&n2, sizeof(n2), 1, input);

   for(int k = 0; k < n1 * n2; k++){
      fread(&i, sizeof(i), 1, input);
      fread(&j, sizeof(j), 1, input);
      fread(&x, sizeof(x), 1, input);
      fread(&y, sizeof(y), 1, input);
      fread(&domain, sizeof(domain), 1, input);
      printf("%d %d %f %f %d\n", i, j, x, y, domain);
   }

   fclose(input);

   return 0;
}
```

ПРИЛОЖЕНИЕ В

Расчёт экспоненты с применением технологии MPI

```
#include "mpi.h"
#include <iostream>
#include <windows.h>
using namespace std;

double F(int n)
{
  if (n==0)
   return 1;
  else
   return n*F(n-1);
}

int main(int argc, char *argv[])
{
  SetConsoleOutputCP(1251);
  int n;
  int myid;
  int nump;
  int i;
  int r;
  long double drob,drobSm=0,Reult, sum;
  double startwtime = 0.0;
  double endwtime;

  n = atoi(argv[1]);

  if (r= MPI_Init(&argc, &argv))
  {
    cout << "Ошибка запуска" << endl;
    MPI_Abort(MPI_COMM_WORLD, rc);
  }

MPI_Comm_size(MPI_COMM_WORLD,&nump);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);


  if (myid == 0)
  {
    startwtime = MPI_Wtime();
  }
```

```
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  for (i = myeid; i <= n; i += numprocs)
  {
   drob = 1/Fact(i);
   drobSum += drob;
  }
 MPI_Reduce(&drobSum, &Result, 1, MPI_LONG_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
 cout.precision(20);
  if (myid == 0)
  {
   cout << Result << endl;
   endwtime = MPI_Wtime();
   cout << (endwtime-startwtime)*1000 << endl;
  }
```