УДК 519.687.1

АЛГОРИТМЫ ДИНАМИЧЕСКОЙ БАЛАНСИРОВКИ ВЫЧИСЛИТЕЛЬНОЙ НАГРУЗКИ И ИХ РЕАЛИЗАЦИИ

С.П. Копысов, А.К. Новиков, В.Н. Рычков

Институт механики УрО РАН, Ижевск

E-mail s.kopysov@gmail.com, sc_work@mail.ru, bob.r@mail.ru

Аннотация

В работе рассматриваются алгоритмы балансировки вычислительной нагрузки в задачах связанных с адаптивными перестроениями сетки, локальными повышениями порядка аппроксимирующих функций выполняемых на многопроцессорных вычислительных системах, в том числе неоднородных/гибридных. Рассматривается балансировка на уровне: операционной системы, промежуточного программного обеспечения и пользовательского приложения.

Ключевые слова: параллельные вычисления, динамическая балансировка вычислительной нагрузки процессоров, метод конечных элементов, разделение графов, уровни балансировки

Введение

Под балансировкой нагрузки (БН) понимается такое разделение вычислительной нагрузки между процессорами многопроцессорной вычислительной системы (МВС), которое позволяет занять каждый процессор полезной нагрузкой по возможности большую часть времени.

Специфические особенности параллельных алгоритмов и обрабатываемых данных пользовательских приложений определяют алгоритмический дисбаланс нагрузки. В рассматриваемой предметной области сеточных задач проблема разбалансировки связана с адаптивными перестроениями сетки, локальными повышениями порядка аппроксимирующих функций и т.д. Такие вычислительные алгоритмы сами способны подвергаться динамическим изменениям, следствием чего становятся ещё более серьезные нарушения балансировки.

Коммуникационный дисбаланс нагрузки обусловлен различной производительностью коммуникационных связей между процессорами МВС (многоядерные процессоры) или другого кластера в группе МВС. С другой стороны дисбаланс в обменах может быть связан и с параллельным приложением и определяться алгоритмом решения задачи.

Важным примером для балансировки являются также гетерогенные вычислительные системы с разделенной памятью. Здесь требуемые ресурсы могут быть определены только в процессе работы приложения, и балансировка должна проводиться динамически, возможно под разными операционными системами.

Во всех этих случаях балансировка нагрузки может осуществляться как статически, так и динамически. При статической балансировке приложение (в системах с локальными памятью и данными) распределяется между процессорами до начала вычислений. Полудинамическая БН предполагает, что разделение определяется на этапе инициализации параллельного приложения, до начала выполнения основных вычислений приложения. Динамическая БН – когда разделение и распределение объектов (например, сеточной области) периодически обновляется в течение всего времени выполнения пользовательского приложения и объекты перемещаются по МВС в соответствии с новым более оптимальным планом (т.е. осуществляется миграция объектов). Причем новое разделение может определяться по различным критериям – производительности узлов МВС, их загруженности или каким-либо эвристическим критериям и т.п. Тогда время выполнения параллельного приложения с динамической балансировкой нагрузки состоит из

$$t_{\sum} = \gamma(t_c + t_p) + t_r + t_m,$$

где $t_{\sum}$ – суммарное время выполнения параллельного приложения, t_c – время на коммуникаций, t_p – параллельной части программы, t_r – время вычисления нового разделения, t_m – время затрачиваемое на перемещения данных из старого разделения в новое; γ – параметр, определяющий периодичность выполнения балансировки нагрузки.

Важно отметить, что алгоритмы, реализующие динамическую балансировку, должны сами обладать высокой эффективностью, в том числе обеспечивать их параллельную реализацию, т.е. $t_r \ll \gamma t_p$. В ряде случаев это связано с невозможностью размещения сетки на одном процессоре, а также выполнением локальной БН, не рассматривая задачу для всей сетки в целом. Затраты связанные с перемещением данных и обменами $t_m + \gamma t_c$ должны уменьшатся с каждым шагом балансировки.

В работе выделяются несколько уровней и дается характеристика алгоритмов их реализующих для достижения балансирования нагрузки процессоров для отдельного параллельного приложения:

- Балансировка на уровне операционной системы (ОС). Механизмы – кластеризация, разделение нагрузки и миграция процессов на этапе выполнения.
- Балансировка на уровне промежуточного программного обеспечения (ППО). Механизм – высокоуровневая балансировка нагрузки в контексте сессии или запроса. Распределение команд на стадии трансляции.
- Балансировка на уровне пользовательского приложения. Реализация алгоритмов балансировки в прикладных параллельных программах.

1. Балансировка на уровне операционной системы

Существующие ОС полагаются на однозначное статическое распределение заданий пользователем, которое может приводить к значительной разбалансировке процессоров.

Кластеризация поможет преодолеть возникающие трудности. Технология кластеризации обладает двумя основными преимуществами. Она повышает масштабируемость и вычислительную мощность. Большинство кластерных решений обеспечивают сбалансированность нагрузки и автоматическое переключение с одного узла на другой в случае перегрузки или отказа. Одним из таких кластерных решений для Linux и является система MOSIX.

OpenMOSIX [1] – системное программное обеспечение для ядер, таких как Linux, состоящее из адаптивных алгоритмов разделения ресурсов. Алгоритмы разделения ресурсов openMOSIX разработаны в соответствии с использованием ресурсов узлов в режиме реального времени.

Все эти алгоритмы реализуются благодаря механизму миграции процессов. С каждым запускаемым процессом ассоциируется идентификатор уникального домашнего узла (UHN), с которого он был запущен. Для миграции процесс разбивается на две части: пользовательский контекст и системный контекст. Пользовательская часть состоит из кода программы, данных, стека, карт памяти и регистров процесса. Системная часть включает описание ресурсов, принадлежащие данному процессу и определяет машинно-зависимую часть, которая всегда остается на UHN процесса. Мигрировавший процесс использует ресурсы нового узла, на сколько это возможно, но взаимодействует с ОС через домашний узел(UHN).

Миграция базируется на информации, обеспечиваемой одним из алгоритмов разделения ресурсов. Стратегия назначения заданий, основана также на экономических принципах и конкурентном анализе. Эта стратегия даёт возможность управлять гетерогенными ресурсами способом, близким к оптимальному. Ключевая идея стратегии состоит в том, чтобы преобразовать общее использование нескольких гетерогенных ресурсов, таких как память и CPU, в единую гомогенную "стоимость" основываясь на кооперации. Тогда задания назначаются на ту машину, где они имеют самую низкую стоимость.

ППО MPI и PVM обеспечивают исходное фиксированное размещение процессов по узлам кластера, в то время как openMOSIX выполняет это динамически в зависимости от конфигурации доступных ресурсов. ППО MPI и PVM работают на пользовательском уровне, на котором действуют обычные приложения. Решение openMOSIX функционирует как модуль ядра операционной системы и потому является полностью прозрачным для приложений. Нет необходимости модифицировать приложения под openMOSIX или связывать их с какими-либо библиотеками. Опера-

ционная система openMOSIX – это, с одной стороны, альтернатива технологиям MPI и PVM, а, с другой стороны, их развитие. Здесь необходимо отметить, что openMOSIX и MPI, PVM могут работать одновременно на одном и том же кластере.

Пользователи и программы могут напрямую взаимодействовать с openMOSIX через API интерфейс, который обеспечивает информацию о состоянии локальных процессорах и процессах.

Очевидным недостатком openMOSIX являются большие накладные расходы при выполнении системных вызовов. Дополнительные издержки появляются при выполнении операций сетевого доступа. Например, все сокеты создаются в UHN, это приводит к большим коммуникационным затратам если процесс мигрирует из UHN.

Общие недостатки такой балансировки в гибкости и адаптивности. Первые возникают из-за невозможности в режиме выполнения принимать решения о балансировке нагрузки в приложении; вторые – из-за отсутствия обратной связи с репликами, объектами, которые работают на стороне сервера балансировки, и под его управлением все вместе представляют процессы, подвергаемые балансировке. Еще один недостаток – стандартное промежуточное ПО ограниченно использует эти возможности ОС, они реализованы по-разному в разных ОС.

2. Балансировка на уровне промежуточного программного обеспечения

Основное промежуточное программное обеспечение (ППО) [2] для распределения процессов в параллельных системах, такое как MPI, PVM и т.д. предусматривают среду выполнения, требующую адаптированных приложений и осведомленность пользователя об этом. Оно включает в себя утилиты для инициализации привязки процесса к узлу, игнорируя доступные ресурсы, например, свободная память и процессы ввода-вывода. Это ППО запускается на уровне пользователя, как обычное приложение, таким образом, становятся неспособным реагировать на неустойчивую загрузку и

адаптивно перераспределять вычислительную нагрузку.

Балансировка нагрузки на промежуточном уровне оказывается предпочтительнее, чем балансировки на более низких уровнях, сети или ОС, которые отличаются отсутствием гибкости и адаптируемости. Промежуточное ПО может обеспечить богатый набор метрик для балансировки, в том числе пользовательских, зависимых от приложений (гибкость); в то время как сетевые или ОС балансировщики работают лишь с фиксированными описаниями нагрузки. Промежуточное ПО может быть использовано совместно как со стандартными, так и со специализированными сетями, ОС, а также с системами балансировки нагрузки, тогда как низкоуровневые балансировщики тесно связаны с аппаратно-программной средой, для которой они предназначены.

Механизмы балансировки нагрузки реализуемые в ППО:

- миграция процессов – MPVM [3], tmPVM [4], MIST [5], Dynamite [6], DAMPVM [7], DynamicPVM [8], CoCheck [9], mpC [10] и др. В системах обмена сообщениями процессы, содержащие код всей прикладной программы, запускаются на разных вычислительных узлах в соответствии с заданной конфигурацией. При возникающих во время выполнения разбалансировках их необходимо прервать, переместить между узлами и вновь запустить в прежнем контексте. В системах обмена сообщениями балансировка нагрузки формулируется через эффективное перераспределение процессов между вычислительным узлами, при этом миграция процессов – это основной механизм балансировки.

- планирование ресурсов – Condor [11], Globus [12], Legion [13], CORBA LoadBalancing [14] и др. Распределенные системы изначально состоят из отдельных модулей, которые, взаимодействуя друг с другом, приводят к разбалансировке системы. Поэтому необходимо эффективно связывать модули во время работы распределенной системы для выравнивания нагрузки. В распределенных системах балансировка может быть описана с помощью структур данных (или объектно-ориентированных интерфейсов), которые позволяют построить соответствие между по-

ставщиками и потребителями ресурсов.

Промежуточное программное обеспечение динамической балансировки параллельных и распределённых МВС должно обеспечивать оптимальное распределение параллельных приложений при динамически изменяющихся ресурсах, за счёт миграции таких процессов между вычислительными узлами МВС.

В состав промежуточного ПО для динамической балансировки нагрузки входят следующие компоненты: монитор нагрузки, планирования процесса выполнения, миграции задач и др.

Текущая нагрузка кластера или распределённой системы отслеживается специальным монитором ресурсов. Менеджер балансировки нагрузки рассчитывает балансировку нагрузки. Если система разбалансирована начинает работать менеджер миграции, определяющий новые места выполнения процессоров и миграции. Отдельный сервис создает копию процесса, включая в него информацию об используемых установленных коммуникационных соединениях и открытых файлах. Для перемещения процесса на новый узел или процессор данные передаются мигратору задач. Процесс с помощью системы перезапуска задач восстанавливает исходное состояние процесса на новом узле, и параллельное приложение продолжает выполняться.

2.1. Миграция процессов

ППО обмена сообщениями (MPI, PVM) состоят из сервиса и библиотеки, которая включается в код прикладной программы. Путем расширения, т.е. модификации исходного кода, можно реализовать дополнительную функцию миграции процессов, как это сделано в MPVM, tmPVM, MIST и др. Основная идея состоит в следующем:

- Мигрирующий процесс приостанавливается. Принимающий сервис определяет соответствующий этому процессу исполняемый файл;
- Состояние мигрирующего процесса загруженные в память код и данные, стек, открытые каналы (файлы, сетевые соединения и др.) и т.д.)

передается на принимающую сторону;

- Все накопленные за первые два шага сообщения, адресованные мигрирующему процессу, также передаются на принимающую сторону;
- Исходный экземпляр мигрирующего процесса полностью останавливается, новый – запускается с того места (состояния), на котором был приостановлен исходный.

Для реализации миграции процессов необходимо наличие соответствующих возможностей (API – прикладной программный интерфейс) операционных систем, на которых будет использоваться расширенное ППО. API должно включать функции приостановки процессов, определения адресного пространства процессов, запуска процессов в контексте и т.д. Миграция процессов может быть реализована в однородной вычислительной сети, т.к. в процедуре миграции задействованы такие параметры, как адресное пространство процесса (перемещение процесса между узлами с разными размерами оперативной памяти реализовать затруднительно).

Решение, мигрировать задаче или нет, зависит от ряда параметров, которые должны быть оценены управляющей программой перемещения (демон UNIX). В случае параллельного задания, выполняющегося на кластере, управляющая программа принимает во внимание: нагрузку каждого узла; среднюю нагрузку в МВС, производительность сети; время, необходимое для введения контрольных точек, перемещения и перезапуска задачи; прогнозируемая дальнейшая нагрузка процессора.

Практически все ППО реализующие миграцию процессов основаны на расширении PVM. В настоящее время поддерживается и развивается несколько таких систем, одна из которых DAMPVM [7].

Построение подобных систем на базе MPI представлено в проектах Hector [15], AMPI [16], TOMPI [17], TMPI [18]), FT-MPI [19].

Рассмотрим одно из таких решений. Charm++/AMPI подход состоит в том, что система поддержки назначает виртуальные процессы на физические процессоры во время выполнения приложения. В отличии от обычного MPI в AMPI число виртуальных процессов задается большим, чем

число доступных процессоров. В основе AMPI лежит ППО Charm++ и использует его средства связи, стратегии балансировки нагрузки, модель пользовательских потоков.

Charm++ [20] программа состоит из множества объектов, распределенных на доступных процессорах. Таким образом, основной модуль параллельных вычислений в программе – это `chare`-объект, который создается на любом доступном процессоре и может обращаться к любому процессору. Эти объекты создаются динамически и их множество может действовать одновременно. `Chare`-объекты посылают сообщения друг другу для запуска своих методов асинхронно. Концептуально, система обрабатывает пул работы, состоящий из сообщений между `chare` и данных для создания новых `chare`. Runtime система (`Charm Kernel`) может выбирать элементы из этого пула и выполнять их. Она не будет обрабатывать два сообщения для одного `chare` одновременно, но она свободна отметить их в любое время.

Charm++ обеспечивает динамическую начальную балансировку нагрузки. Таким образом расположение (номер процессора) не нужно определять при создании удаленного `chare`. Система сама разместит `chare` на процессор с наименьшей нагрузкой.

`Charm Kernel` идентифицирует `chare` по его идентификатору `ChareID`. Так как пользовательский код не знает, на каком процессоре находится `chare`, то объект потенциально может мигрировать с одного процессора на другой (это поведение используется для динамической балансировки структуры `chare`-контейнеров, таких как массивы). Другой вид Charm++ объектов – это `chare` контейнеры. Его разновидности – это `chare`-группы, `chare`-группы узлов и `chare`-массивы, соотносящиеся с группами, группами узлов и массивами. Группа – это набор `chare`, по одному на каждый процессор, который адресуется использованием уникального для всей системы имени. Массив – это набор произвольного числа мигрирующих `chare`, отображенных на процессоре согласно определенной пользователем картой групп.

При запуске приложения пользовательские объекты нужно зарегистри-

ровать в системе поддержки `Charm Kernel`, который при этом назначает каждому из них уникальный идентификатор. При вызове методов на удаленном объекте эти идентификаторы должны быть указаны системе. Регистрация пользовательских объектов и поддержание этих идентификаторов может быть очень громоздким. Поэтому в Charm++ добавлен интерфейсный транслятор. Он генерирует определения прокси объектов. Кроме того, интерфейсный транслятор позволяет расширять базовую функциональность ядра Charm++ введением потоков и `future`-объектов пользовательского уровня. Эти потоковые `entry`-методы могут блокироваться в ожидании данных, выполнением синхронного вызова методов удаленного объекта, которые возвращают данные в виде сообщения.

Существует ряд параллельных распределенных Java-технологий включающих возможности балансирования нагрузки.

Выделим только систему ParJava [21] – это технология SIMD-программирования на Java. В качестве коммуникационной среды выбран стандарт MPI. С помощью JNI (Java Native Interface – интерфейс вызова стандартных функций C++ из Java) инкапсулирована MPI-система LAM. Поверх интерфейса коммуникационной среды реализована библиотека масштабирования Java-приложений на однородные и неоднородные сети.

К промежуточному программному обеспечению с возможностью балансировки нагрузки можно отнести и российские разработки – mpC [10] и DVM [22].

В языке программирования mpC – расширении ANSI C – принят подход, когда пользователь распределяет не только данные, но и вычисления. Переменные и массивы распределяются по виртуальным сетям и подсетям, при этом в описаниях сетей указываются относительные мощности узлов и скорости связей между ними. В процессе выполнения mpC-программы система поддержки языка стремится максимально эффективно отобразить виртуальные сети на группы процессоров. В результате пользователь получает возможность равномерно нагружать узлы. Этот подход позволяет эффективно использовать гетерогенные сети и решать нерегулярные зада-

чи, отличающиеся тем, что объем вычислений, производимых на каждом узле, выясняется динамически в процессе решения задачи. mpC содержит средства, позволяющие программисту изменять оценку производительности реальных процессоров, используемую при отображении на них виртуальных процессоров, настраивая её на те вычисления, которые будут выполняться этими виртуальными процессорами.

В DVM-системе разработанной в ИПМ им. М.В.Келдыша РАН реализованы возможности задания производительностей процессоров (или их автоматического определения при запуске программы) и их учета при распределении данных и вычислений между процессорами. Это позволяет DVM-программам распределять нагрузку на неоднородных кластерах.

Общий недостаток рассмотренного промежуточного ПО в том, что оно ограниченно использует возможности ОС и реализуется по-разному в разных ОС.

2.2. Планирование ресурсов

Сегодня все чаще при организации вычислений применяется разделение работы, данных и вычислительных ресурсов, предусматривающее использование распределенных ресурсов. Распределенные вычисления становятся магистральным направлением развития вычислительных технологий. На смену отдельным, независимым суперкомпьютерам приходят группы высокопроизводительных систем, объединенных в кластеры. Эти ВС динамически формируется из распределенных гетерогенных узлов, которые могут свободно подключаться и отключаться и управлять выделением своих ресурсов в общее пользование.

Однако системы Grid вычислений не предназначены для решения параллельных задач, а нацелены по большей части на выполнение пакетных приложений – где каждое отдельное приложение выполняется полностью на одном кластере. Система управления Grid занимается диспетчеризацией отдельных заданий, а не взаимосвязью между параллельными ветвями одной задачи. Несколько независимых менеджеров распределяют вычисли-

тельные задачи на кластеры в соответствии с некоторым критерием. Далее каждый кластер распределяет задачи, присвоенные ему локальным планировщиком. Эти функции выполняют системы пакетной обработки заданий – CODINE, PBS, Condor и др.

В среде Grid одно из возможных решений проблемы планирования заключается в создании своего рода метадиспетчера – между службами Grid и менеджерами ресурсов локальных вычислительных систем.

Рассмотрим одну из существующих моделей планирования. Все задачи, которые обеспечивают/запрашивают какой-либо сервис, оповещают о своих характеристиках/требованиях. Далее специальный сервис строит соответствия между заявками подзадач так, чтобы максимально удовлетворить потребности одних и использовать возможности других, после чего он информирует сущности о построенном соответствии. Затем задачи устанавливают контакт и выполняют необходимый сервис. Данная модель реализована в системе Condor.

При централизованном подходе в распределении ресурсов сервис `Manager` отвечает за проблему размещения. Каждый `worker` (узел МВС) постоянно обращается к `Manager` с запросами (например, сообщение об окончании выполнения задачи) получает задачу от него и исполняет ее. В свою очередь `workers` могут послать новые задачи `Manager`, которые затем могут быть размещены по другим рабочим. Эффективность такой стратегии зависит от числа `workers` и относительной стоимости получения задач от `Manager` и их выполнения.

В полностью децентрализованной схеме нет центрального управляющего. Соответственно, отдельный пул задач назначается на каждый узел МВС, и незанятые процессоры требуют задач от других процессоров. В результате пул задач становится распределенной структурой данных, которая доступна различным задачам асинхронно. При этом определяется какая-либо процедура доступа. Например, `worker` может требовать задания только от небольшого числа соседей или выбирать другие процессоры по "заданному"случайному закону.

Конечно возможны и смешанные подходы. В этом случае доступ к распределенному пулу задач может быть получен несколькими способами. `Workers` могут быть назначены ответственными как за вычисления, так и за управление очередью задач. В этом случае каждый `workers` должен периодически переключаться с вычислений на проверку ожидающих запросов.

Составление расписаний требует механизма для обнаружения момента времени, когда весь алгоритм будет завершен, в противном случае простаивающие `workers` никогда не остановятся, требуя задания от других `workers` .

Ограничиваясь только вопросами программного обеспечения балансировки нагрузки, рассмотрим систему Condor [11]. Кроме выполнения миграции на свободные машины Condor обеспечивает управление распределенными ресурсами. Система Condor – это прежде всего ПО для поддержки распределенных ресурсов. Механизм управления заданиями, предполагает: ведение очередей, составление расписаний, назначение приоритетов, классификацию доступных ресурсов, слежение за состоянием заданий. Планирование ресурсов в Condor ориентировано на выполнение продолжительных заданий.

В подобных системах планирования ресурсов распределенных параллельных вычислений широко применяются экономико-математические модели и алгоритмы.

Condor работает по типу брокера между продавцами – пулом свободных вычислительных узлов и покупателями – пользовательскими приложениями. Кроме этого, оба субъекта сделки задают диапазоны своих пожеланий и скидки (премии). Таким образом, кроме имеющихся ресурсов, узлы сообщают при каких условиях они буду выполнять задания от Condor и какой тип приложений предпочтителен. Например, покупатель выставляет требования к товару (МВС): наличие определенного числа процессоров, размера памяти и т.д. Продавец может отдать предпочтительное право занимать заданиями свой узел с теми или иными приложениями и т.д.

Система Condor также может быть использована для выполнения параллельных продолжительных заданий по технологиям MPI и PVM путем динамического предоставления свободных в данный момент машин.

Для работы в режиме Condor-PVM, в отличии от рассмотренных в предыдущем параграфе технологий миграции, не требуется изменений программы – используется существующий стандартный интерфейс вызовов PVM. При этом Condor-PVM действует в роли менеджера ресурсов для демона PVM и как только PVM – программе потребуется дополнительные узлы будет сформирован запрос для Condor на поиск свободных машин из пула и их размещение в виртуальную машину PVM.

ППО Condor поддерживает выполнение параллельных приложений, использующих MPI. В отличие от PVM текущие реализации MPICH не допускают динамического распределения ресурсов. Кроме этого, пока в реализации Condor MPICH не существует диспечеризации, возможна ситуация, когда ни одна из задач не сможет выполняться.

В последнее время в распределенные системы включаются элементы, позволяющие учитывать архитектуру ЭВМ. Так в приложениях CORBA включают сервисы, повышающие качество взаимодействий между распределенными объектами в зависимости от возможностей коммуникационной среды [14]. Модели планирования ресурсов используются в параллельно-распределенных системах реального времени.

Проект OMG разрабатывает стандарт балансировки нагрузки для CORBA – Load Balancing and Monitoring. Одна из реализаций представлена в проекте TAO (The ACE ORB, ACE – Adaptive Communication Environment) [14] как раз для применения в системах реального времени. TAO `LoadBalancing` состоит из сервиса балансировки нагрузки `LoadBalancer`, который манипулирует копиями CORBA-объектов, репликами (`replica`). Основная задача балансировщика такая же – эффективное связывание сущностей, в данном случае CORBA клиентов и серверов (объектов). Интерфейс балансировщика включает описания разного вида нагрузки (производительность процессора, характеристики жесткого

диска, оперативная память, вид коммуникационной сети), но структурные описания ресурсов, подобные `classad`, отсутствуют (они могут быть реализованы в пользовательских балансировщиках на уровне приложения, в виде объектно-ориентированных интерфейсов). В TAO `LoadBalancing` особенное внимание уделяется функциональным возможностям сервиса, для которого описываются и реализовываются различные стратегии балансировки. Стратегии можно рассматривать с двух независимых позиций: по степени связывания клиента с сервером (как связывать) и по способу балансировки (с кем связывать). Клиент может быть связан с сервером в течение сессии, запроса или по необходимости. В прикладной программе можно определять, в зависимости от ее логики, когда и какая связь наиболее приемлема (сессии выгодно использовать в тех случаях, когда клиент должен держать сервер в единоличном пользовании в течение некоторого времени; запросы применяются для серверов, используемых часто различными клиентами). Также в прикладной программе можно определить тот или иной способ балансировки. Выделяют два вида недаптивные и адаптивные. В первых балансировщик определяет для клиента сервер в соответствии с каким-либо алгоритмом (по кругу, случайным образом и т.д.), а во вторых для выбора используется информация об аппаратно-программной среде и ее характеристиках. Системы балансировки нагрузки в распределенных системах могут работать как в однородных, так и в неоднородных вычислительных системах. Здесь не возникает проблемы миграции процессов, т.к. изначально прикладная программа проектируется в виде набора распределенных модулей.

Сервис балансировки нагрузки в CORBA обеспечивает: поддержку объектно-ориентированной модели балансировки нагрузки, т.е. детализация балансировки должна быть основана на объектах, а не на TCP/IP адресах или процессах; прозрачность приложений, т.е. простой интерфейс и минимум модификации существующих программ; поддержку пользовательских метрик нагрузки и стратегий балансировки. Кроме того, необходимо, чтобы сервис балансировки нагрузки CORBA позволял приложени-

ям определять семантику метрики для того, чтобы измерять нагрузку и устанавливать стратегии, для определения семантики балансировки и т.д.

На данный момент технология CORBA не решает всех проблем балансировки нагрузки. Основные задачи, которые необходимо решить при реализации сервиса балансировки: платформонезависимость сервиса балансировки нагрузки; обеспечение обратной связи и контроля; поддержка стратегий в виде модулей; использование возможностей адаптивной балансировки; уникальная идентификация объектов; эффективная интеграция всех компонентов сервиса балансировки.

Преимущества ПО для балансировки нагрузки на основе CORBA – может использоваться совместно со средствами балансировки на уровне сети, ОС и обеспечивает широкий круг условий балансировки нагрузки на уровне приложения.

Однако, не существует одной модели или стратегии, которые подходили бы для всех приложений. Следовательно, многие решения для балансировки нагрузки часто требуют внесения изменений "программно".

3. Балансировка на уровне пользовательского приложения

Методы балансировки нагрузки, реализуемые программно в пользовательском приложении основываются, на следующих допущениях:

- вычислительные узлы (или процессоры на параллельных компьютерах) как правило однородны;
- рассматриваемое параллельное приложение работает в монопольном режиме;
- расчётная(сеточная) область может быть произвольно разделена на необходимое число подобластей.

Отказаться от допущения достаточно сложно при большом числе вычислительных узлов с различной производительностью и доступной памятью. Один из возможных вариантов будет рассмотрен ниже. Второе ограничение требует при выполнении приложения разрешения администратора си-

стемы и обычно вызывает неудобство другим пользователям. Кроме того, большие вычислительные ресурсы редко выделяют для выполнения одного задания. Третье условие практически всегда выполняется во всех приложениях.

Во избежания этих ограничений, примем подход, который балансирует нагрузку процессоров и на этапе разделения области, и во время выполнения программы. На этапе разделения область задачи или приложения разделяется на n_S подобластей.

Принимается, что существует n_p вычислительных узлов с сетевой структурой и выполняется параллельное приложение с числом подобластей $n_S \geq n_p$.

3.1. Постановка задачи

Один из наиболее общих алгоритмов реализации БН в приложении состоит в том, чтобы записать достаточно простую функцию стоимости, имеющую смысл временных затрат на выполнение приложения, независимую от особенностей задачи, минимизация которой и давала бы распределения задачи по процессорам.

Запишем функцию стоимости как сумму частей, которые минимизируют дисбаланс нагрузки процессоров и межпроцессорных обменов [23]:

$$H = H_{calc} + \mu H_{comm}, \tag{1}$$

где H_{calc} – является частью функции стоимости, которая минимальна, когда каждый процессор имеет равную работу, H_{comm} – минимальна, когда время обменов минимизировано, и μ – параметр, выражающий равновесие между этими двумя частями. Соответствующие составляющие функции стоимости имеют следующий вид:

$$H = \sum_k \sum_i (W_{ik}\delta_{ik} + \mu \sum_l \sum_j V_{ij} S_{kl} \delta_{ik} \delta_{jl}), \tag{2}$$

где первое слагаемое соответствует нагрузке, а второе связям и W_{ik} – стоимость обработки i-й части программы на k-м процессоре; V_{ij} – объем данных, посылаемых от процессора i к процессору j, при $V_{ij} = 0$ процессоры

не взаимодействуют; S_{kl} – мера стоимости связи процессоров k и l, при $S_{kl} = \infty$ процессоры не связаны; $\delta_{ik} = 1$, если i-я часть обрабатывается k-м процессором; μ – константа зависящая от конкретной ВС и характера задачи.

Если первое слагаемое в функции стоимости не вызывает особых затруднений для сеточных задач, то второе слагаемое необходимо рассмотреть более подробно. Стоимость связи от процессоров k к l зависит от архитектуры мультипроцессорной системы. Конечно же, необходимо учитывать количество физических каналов пересылки данных на каждом узле и возможность пересылки в разных направлениях. Для некоторых параллельных систем возможно записать этот показатель явно. Тогда надо учитывать и то, что коэффициент μ зависит от программного обеспечения передачи данных. Стоимость передачи объема данных V_{ij} также во многом определяется стилем программирования, например: стоимость будет намного ниже, если передавать V_{ij} как одно сообщение. Обратим внимание и на то что, что любое перекрытие между вычислением и связью в функции стоимости игнорируется, а это очень часто используется, например в MPI. Для сеточных задач актуальным является и выбор самой меры оценки коммуникационных затрат.

Формально можно представить простую физическую интерпретацию выражения (2). Пусть каждое состояние отображаемой системы (приложения) характеризуется определенным значением функции стоимости – аналогом функции энергии. Для нахождения глобального минимума функции стоимости (энергии) моделируется процесс медленного охлаждения от состояния с высокой температурой до "замораживания" системы, что означает достижение конфигурации с минимальной средней энергией.

Этот метод был разработан Метрополисом [24] и доработан Киркпатриком [25] в виде аналогии термодинамического процесса нагревания и медленного охлаждения материала для получения кристаллической структуры.

Начиная со случайно выбранной точки в пространстве поиска, делается

шаг в случайном направлении. Если этот шаг приводит в точку с более низким уровнем значения функции оптимизации, то он принимается. Если же он приводит в точку с большим значением функции оптимизации, то он принимается с вероятностью P. Функция P сначала близка к единице, но затем постепенно уменьшается до нуля – по аналогии с охлаждением твердого тела. Таким образом, в начале процесса моделирования принимаются любые ходы, но, когда "температура" падает, вероятность совершения негативных шагов уменьшается. Негативные шаги иногда необходимы в том случае, когда нужно избежать локального оптимума, но принятие слишком многих негативных шагов может увести в сторону от глобального оптимума.

Использование моделируемого отжига для декомпозиции структуры многогранников Вороного (МВ) рассмотрим на примере алгоритма, моделирующего нормальный рост зерен и использующего параллелизм по данным [23].

В качестве функции стоимости была использована функциональная зависимость времени одного MCS от конфигурации разбиения структуры многогранников, параметров алгоритма и устройства вычислительной системы.

Согласно схеме алгоритма, моделирующего рост зерен, если на i-м ($i = 1, 2, \ldots n_p$) процессоре вычислительной системы размещены N_i узлов диаграммы, то за один MCS, процессор произведет N_i попыток изменения ориентации. Обозначим среднее время, затрачиваемое на один шаг (одну попытку) i-м процессором как, $\tilde{t}_i$. Тогда вычислительная часть временных затрат на один MCS составит $W_i = \tilde{t}_i \times N_i$. Для многопроцессорных систем, состоящих из процессоров различной производительности, значение $\tilde{t}_i$ будет различным.

Для поддержания единой конфигурации системы процессорам во время работы необходимо обмениваться значениями ориентации в граничных узлах. Пусть n_B^{ij} – число узлов структуры, принадлежащих i-й подобласти и находящихся на границе с j-й подобластью ($j = 1, 2 \ldots n_p$). Тогда за

один MCS с i-го процессора на j-й процессор будет совершено в среднем n_B^{ij} передач значений измененных ориентаций. Обозначим среднее время, затрачиваемое на одну передачу с i-го на j-й процессор, как t_{ij}. Тогда часть временных затрат, связанная с обменами за один MCS, на i-м процессоре составит $\sum_{j=1}^{n_p} t_{ij} n_B^{ij}$. Общее время, затрачиваемое i-м процессором на выполнение одного MCS, равно сумме времен, затрачиваемых на вычисления и обмены. Значения параметров t_{ij} зависят от структуры системных связей многопроцессорной системы и схемы наложения на них программных каналов. Изменение пользователем определенных значений матрицы t_{ij} позволяет адаптировать критерий разбиения на пользовательскую структуру программных связей. Например, увеличивая значения компонент матрицы, для которых $|i-j| = 2$, можно уменьшить вероятность дальних (диагональных) передач или совсем их исключить, обеспечивая тем самым структуру программных каналов типа кольцо.

Учитывая необходимость синхронизации алгоритма после каждого MCS, функцию стоимости всего алгоритма можно представить в виде:

$$H = \sum_{i=1}^{n_p} \{\tilde{t}_i N_i + \sum_{j=1}^{n_p} t_{ij} n_B^{ij}\}. \qquad (3)$$

Таким образом, процессоры, ранее закончившие выполнение шага МК, при синхронизации ожидают завершения выполнения того же шага самым загруженным процессором.

Для определения параметров $\tilde{t}_i$ и t_{ij} были проведены замеры затрат времени на вычисления и обмены, а также вычислены значения N_i, n_B^{ij} и H для структуры распараллеленной методом рекурсивной координатной бисекции. Для рассматриваемого алгоритма роста зерен были получены следующие значения: $\tilde{t} = \tilde{t}_i = 0.00002$ сек; $t_{12} = 0.000012$ сек; $H = 0.178$ сек.

Для распараллеливания алгоритма моделируемого отжига была использована стратегия многократных независимых исполнений. Алгоритм моделируемого отжига начинается с определения температуры, близкой к температуре Кюри системы. Для этого каждый процессор формирует случайное распределение узлов всей сетки по процессорам, затем на каждом процессоре выполняется несколько попыток изменения распределения (в

нашем случае выполнялось N попыток, то есть один MCS). При этом, рассматриваются все попытки и для каждой определяется абсолютное значение изменения функции стоимости $|\triangle H|$. Далее находится среднее значение абсолютного изменения функции стоимости по всем проведенным попыткам $|\triangle H|_m$. Температура определяется исходя из предположения, что в точке Кюри вероятность принятия неблагоприятного изменения распределения близка к 0.5, следовательно $T = -|\triangle H|_m / ln(0.5)$. Полученная таким образом температура принимается в качестве начальной, если используется случайное распределение. Далее, используя простое геометрическое распределение охлаждения $T_{i+1} = \lambda T_i$, каждый процессор выполняет одинаковое число шагов по температуре, при этом, на каждом таком шаге производится достаточное для определения $< H >$ число попыток изменения распределения. Значения λ выбирались различными для каждого процессора. По полученным значениям $< H >$ строится приближенный график зависимости $< H >$ от T. В случае использования заданной исходной конфигурации распределения, начальная температура находится из полученного графика, исходя из значения функции стоимости для нее. Основная фаза отжига состоит в том, что каждый процессор выполняет собственную цепочку охлаждения системы распределения. На каждом шаге температуры T_i выполняется определенное число шагов алгоритма и определяется абсолютное изменение стоимости, произошедшее за этот период. Исходя из графика зависимости $< H >$ от T определяется новое значение температуры T_{i+1} для $H_{i+1} = H_i - \triangle H$. Конечная температура охлаждения T_f выбирается из тех соображений, что вероятность принятия попытки переворота с минимальным положительным $\triangle H_{min}$ должна быть больше 0.5, то есть $T_f > -\triangle H_{min} / \ln(0.5)$. Минимальное значение $\triangle H_{min}$ определяется на всех этапах работы алгоритма. В качестве конечного результата выбирается конфигурация с минимальным значением функции стоимости. Благодаря почти полному отсутствию обменов между процессорами эффективность представленного параллельного алгоритма выше 90%. Эффективность алгоритма увеличивается с ростом числа процессо-

ров.

При моделировании роста зерен на рассматриваемой структуре данных моделируемый отжиг не давал заметного преимущества ($H = 0.170$ сек.) по сравнению с более простым координатным разделением, которое было принято в качестве первоначального, исходя из геометрии области (куб). Для областей сложной геометрии преимущество отжига становится очевидным.

Другую возможную методику поиска оптимального разделения области, которая основана скорее на биологических, а не физических процессах, составляют генетические алгоритмы [26]. Конечно, эволюция биологических систем не единственная аналогия для получения новых методов разделения области. Нейронные сети [27], например, основаны на моделировании поведения нейронов в мозге. Они могут использованы также для задач разделения вычислительной нагрузки в МВС.

Как показывает практическое применение, моделируемый отжиг является наиболее уязвимым из эвристических методов, он более всего зависит от выбранных параметров моделирования, т.е. больше всех остальных нуждается в дополнительной настройке. Тем не менее, в отличие от других методов, использование моделируемого отжига для решения задачи разделения сеточных структур позволяет явно учитывать структурное устройство ВС, в том числе структуру системных связей между процессорами, а также привязывать критерий декомпозиции к конкретному виду параллельного алгоритма.

Недостатком метода является то, что затраты на выполнение алгоритма разделения часто соизмеримы с затратами самого приложения. Конечно, возможна дополнительная постановка различных ограничений для (2) на память, на однородность МВС, на порядок выполнения и т.д. Однако задача в такой формулировке остается весьма сложной и с дополнительными упрощающими предположениями. Более подробно его применение и параллельная реализация будут рассмотрены в последнем разделе.

3.2. Методы балансировки для сеточных задач

Для большинства задач вычислительной механики естественными объектами для балансировки нагрузки являются компоненты вычислительной сетки: конечные элементы или объемы; узлы; грани и ребра. Геометрические и топологические свойства сеток гарантируют, что хорошие разделение для них существуют.

Расчетным сеткам можно поставить в соответствие несколько видов графов. Наиболее употребительными являются графы узлов и элементов. Для более точной балансировки нагрузки необходимо использовать комбинированные графы, как например, граф связи элементов и узлов.

Разделение в сеточных задачах можно определить в терминах нескольких критериев или метрик [28, 29]: балансирование нагрузки (подобласти должны иметь почти равное число элементов или узлов); минимальные длины границ (минимальное число общих граней или узлов между подобластями); минимальная связность подобластей (минимальное число соседних подобластей); минимальная ширина ленты матрицы жесткости, принадлежащей каждому процессору и оптимальной обусловленности матриц подобластей (локальные матрицы должны быть хорошо обусловлены); в ряде случаев топология и размеры подобластей должны соответствовать возможностям сеточных генераторов.

В сеточных задачах вычислительная нагрузка на процессор складывается из суммы нагрузок, связанных с каждым объектом сетки. В МКЭ – это конечный элемент, в методе конечных объемов – контрольный объем (число ребер сетки).

Представим неструктурированную конечно-элементную сетку как взвешенный граф. Под вершинами графа будем понимать центры элементов, т.е. строится двойственный граф по отношению к конечно-элементной сетке.

Каждому конечному элементу – вершине графа соответствуют своя вычислительная нагрузка. Ребро графа, соединяющее два элемента, соответствует связи и определяет коммуникационную нагрузку. Тогда процесс балансировки нагрузки можно представить как нахождение разделение гра-

фа сетки $G_M(V, E, W)$ со множеством вершин $V = \{v_i | i = 1, 2, \ldots, N\}$, каждая из которых связана ребром $E = \{E_{i,j} | \{v_i, v_j\}\}$. Вершины и ребра графа могут быть заданы с весами, что необходимо при рассмотрении разделения сеточных моделей, в которых нагрузка на элементах может быть не равномерной, например для *p-версии* МКЭ: $W_V = \{w_v(v_i) \in \mathbb{N} | v_i \in V\}$ – вес для вершин(конечных элементов) и $W_E = \{w_E(E_{i,j}) \in \mathbb{N} | E_{i,j} \in E)$ – соответственно, вес для ребер. Соответствующий пример приложения приведен в последнем параграфе.

Задача нахождения разделения графа $G_M(V, E, W)$ с весами W_V и W_E на n_p подобластей $(V_1, V_2, \ldots V_p)$ заключается в следующем:

$$V = \bigcup_{i=1}^{n_p} V_i, \quad V_i \bigcap V_j = \emptyset \quad \text{при} \quad \forall \quad i \neq j, \tag{4}$$

с учетом, того что

$$\sum_{v_i \in V_j} w_v(v_i) = \sum_{v_k \in V_l} w_v(v_k) \quad \forall j \neq l \quad j, l \in 1, 2, \ldots, n_p, \tag{5}$$

и число получаемых общих ребер или сумма весов всех ребер, принадлежащих границе между подобластями, должна быть минимально

$$\min |E_c| = \{E_{i,j} \in E | v_i \in V_p, v_j \in V_q \quad \text{при} \quad p \neq q\}. \tag{6}$$

Последнее определяет коммуникационную нагрузку на всех этапах решения задачи. При использовании узлового графа сетки такая мера является сильно завышенной.

Разделение и распределения вычислительной нагрузки в адаптивном МКЭ на основе поэлементной декомпозиции с использованием дуального графа сеточной модели запишем в виде следующего алгоритма.

Алгоритм 1 Разделение и перераспределение нагрузки в адаптивном МКЭ

1: **if** $k = 0$ **then**
2: Выполняем статическую балансировку на основе разделения $D = (\mathcal{T}, F)$.
3: **end if**
4: 1. Определяется вычислительная нагрузка $W_i^k \forall \mathcal{T}_i \in \mathcal{T}$ сеточной модели $\mathcal{T}$.
5: 2. Для $\mathcal{T}_i$ строится дуальный граф $G^k(V^k, E^k, W^k) = \bigcup_{i=1}^{n_p} G_i^k(V_i^k, E_i^k, W_i^k)$.
6: 3. Находится величина разбалансированности $\triangle W$.
7: **if** $\triangle W > W^*$ **then**
8: 4.1. Строится новое разделение $\bigcup_{i=1}^{n_p} G_i^k(V_i^{k+1}, E_i^{k+1}, W_i^{k+1})$.
9: 4.2. Перераспределяется вычислительная нагрузка в соответствии с новым разделением $\bigcup_{i=1}^{n_p} G_i^k(V_i^{k+1}, E_i^{k+1}, W_i^{k+1})$.
10: **end if**

Здесь вычислительные затраты W_i^k полагаются весами дуального графа $G_i^k(V_i^k, E_i^k, W_i^k)$ на k-ом шаге адаптации сеточной модели $\mathcal{T} = \bigcup_{i=1}^{n_p} \mathcal{T}_i$.

В случае вычислений на гетерогенном кластере [30] в весах графа $G^k(V^k, E^k, W^k)$ учитываются производительность процессора, коммуникационные затраты (латентность и пропускная способность), требуемый объем оперативной памяти. Современным развитием гетерогенных ВС стали гибридные системы с графическими ускорителями и теперь актуальным является распределение вычислительной нагрузки между ядрами центральных и графических процессоров. Важная роль в этом отводится многоуровневым алгоритмам разделения графов, с выделением графов сеточной модели, подобласти и матриц системы уравнений, например в методе дополнения Шура [31].

Отметим, что задача разделения (4)–(6) остается NP-сложной, поэтому при её решении используются эвристические методы разделения графов.

3.3. Алгоритмы разделения графов

Большинство алгоритмов разделения графов могут быть отнесены к двум основным классам: геометрическому и топологическому. Рассмотрим некоторые из них. Основная характеристика методов первого класса – то, что они игнорируют информацию связности графа G сетки. Они основаны на сортировке координаты вершин графа и разделении их по декартовой,

полярной или инерционной осям графа G [32]. В большинстве случаев используется рекурсивное разделение графа. Первоначально граф разбивается на две части по одной оси, минимизируя связи между частями. Далее, рекурсивно делится пополам каждый подграф вдоль другой оси. У такого подхода есть два преимущества: во-первых, разделение каждой подобласти проще, чем рассмотрение всей области; во вторых, имеется естественный параллелизм.

К этой группе также можно отнести построение упорядочиваний на основе восьмеричного дерева(oct-tree) [33], кривые Гильберта, Мортона и т.д. [34]. Эти кривые связаны с любопытным понятием теории функций, а именно – всюду плотными кривыми. Алгоритмы на их основе имеют время выполнения и качество разбиения сравнимое с простыми геометрическими методами. Перечисленные алгоритмы быстрее, но более сложны в представлении. Подобно геометрическим алгоритмам, данные методы не используют связность вершин сетки. Качество сгенерированных разделений обычно хуже, чем в схемах разделения основанных на графах. Такие методы привлекательны по нескольким причинам: возможность получения почти оптимального разделения для регулярных сеток $|E_c| \approx C \cdot (|V|/n_p)^{(d-1)/d}$, здесь d – размерность пространства; затраты на разделение намного меньше, чем для алгоритмов, основанных на графах; главное преимущество – создаваемая глобальная нумерация может упростить некоторые аспекты распараллеливания программ.

Привлекательность алгоритмов, основанных только на геометрической информации заключается в их скорости и простоте реализации. Кроме того, алгоритмы являются инкрементальными. Недостатки этого подхода очевидны.

Дополнительная информация о графе, содержащая смежные вершины или связность ребер, используется в алгоритмах другой группы при исследовании различных свойств графа. Один из лучших методов разделения, но требующий значительных вычислительных затрат, основан на нахождении специфического собственного вектора разреженной матрицы, имеющей

структуру матрицы графа, и использовании его как разделителя [35–38]. Согласно методу, вершины V сортируются в порядке, определенном размером компонент собственного вектора, или комбинации собственных векторов матрицы Лапласа $L_{i,j}(G)$ графа G. Данный подход зависит от выбора собственного вектора. Фидлер М. [35] показал, что второй собственный вектор $L(G)$ представляет хорошую меру связности графа G. Одним из недостатков метода является плохая параллелизуемость.

Часто используемый алгоритм Кернигана-Лина(KL) был предложен в [39] для разделения графов для размещения СБИС и стал основой большой группы методов разделения графов. При заданном начальном разделении для каждой вершины вычисляется увеличение числа разрезанных ребер при перемещении вершины из одной подобласти в другую. Для больших графов качество полученного разделения во многом будет зависеть от начального разделения.

Метод основан на заданном разделении (V_1, V_2) множества V графа G и улучшении его, перестановками подмножеств в V_1 и V_2. Критерий выбора подмножеств определяется функцией выгоды, которую определим следующим образом: для $v \in V_1$, $g_v = d_{V_2}(v) - d_{V_1}(v)$, где $d_A(v)$ – число соседей узла v, которые принадлежат A и для $\omega \in V_2$, $g_\omega = d_{V_1}(\omega) - d_{V_1}(\omega)$. Тогда выгода, получаемая при обмене узла(вершины) $v \in V_1$ на узел $\omega \in V_2$ равна

$$g_{v,\omega} = g_v + g_\omega - 2\delta(v, \omega), \tag{7}$$

где $\delta(v, \omega)$ – определяется как

$$\delta(v, \omega) = \begin{cases} 1, & \text{если} \quad (v, \omega) \in E \\ 0, & \text{в противном случае.} \end{cases}$$

На каждой внутренней итерации алгоритма перемещается вершина графа, имеющая самое высокое увеличение множества разделителя. Вершина фиксируется и модифицированная прибыль. Процедура повторяется до тех пор, пока самое высокое увеличение не станет отрицательным, т.е. пока вся вершина не зафиксируются.

Каждое повторение KL-алгоритма имеет сложность $\mathcal{O}(E \log(E))$. Разработано несколько усовершенствований первоначального KL алгоритма.

Один такой алгоритм Федуччи-Маттеуса (FM) [40], который уменьшает сложность до $\mathcal{O}(E)$, используя соответствующие структуры данных. FM-алгоритм немного отличается от первоначального алгоритма KL, тем, что на каждом шаге FM перемещает один узел из одной части в другую, в то время как KL-алгоритм выбирает пару узлов, по одному в каждом разделении и обменивается ими.

KL – локальный алгоритм оптимизации с ограниченной возможностью выхода из локальных минимумов и разрешения шагов с отрицательным приращением. Имеет временные затраты для каждого прохода графа, лучше чем, у моделируемого отжига при меньшем числе разрезанных ребер.

В настоящее время много внимания уделяется модификации существующих графовых алгоритмов разделения, которая выполняется в основном в трех направлениях: многоуровневые разделения, гиперграфовое разделение, комбинации алгоритмов разных классов и параллельное выполнение алгоритмов.

Идея многоуровневого подхода состоит в том, чтобы сформировать информацию о связности исходного графа G_0 множеством более грубых графов $G_1, G_2, \ldots G_n$ таких, что $|V_0| > |V_1| > \ldots |V_n|$. Алгоритм, основанный на многоуровневом подходе обычно имеет три этапа: первый – огрубление графа, в нём первоначальный граф сокращяется в набор последовательно более грубых графов; этап непосредственно разделения, где делится грубый граф на части; стадия разогрубления и уточнения, при котором разделение грубого графа последовательно интерполируется на более мелкие графы, и модифицируется разделение графа каждого уровня. Процесс продолжается до получения разделения исходного графа. Многоуровневый подход был разработан в [41] для ускорения алгоритма рекурсивного спектрального деления и затем активно использовался в комбинации с алгоритмами типа KL [42].

Хотя многоуровневый подход приводит к значительному сокращению времени вычислений для больших сеток, возможно еще одно ограничение – на память.

Известно, что при использовании узлового графа в предсталении расчетной сетки, коммуникационные затраты оцениваются приближенно. В этом случае нарушается симметричность зависимости данных. Тогда целесообразнее использовать гиперграфовое представление для лучшей минимизации коммуникаций по сравнению с рассмотренным ранее графовым представлением.

Гиперграф является обобщением неориентированного графа $G_{\mathcal{H}}(V, E)$ – совокупность множества вершин $V = 1, 2, \ldots, n$ и множества гиперребер (связей этих вершин) E. Гиперребра выражают n-ое отношения между вершинами, в отличие от графа n может быть более двух $e_j \subseteq V$. Другими словами, ребро может быть инцидентным более чем двум вершинам. Аналогично могут быть определены веса ω_{v_i} для каждой вершины гиперграфа $v_i \in V$ и гиперребра ω_{E_i}.

Для расчетных сеток можно выделить несколько способов определения гиперграфа: 1. Вершинами гиперграфа являются узлы расчетной сетки, гиперребром – узлы сетки связанные с данной вершиной и сама вершина; 2. Вершины гиперграфа – ячейки сетки, гиперребро – ячейки сетки ее окружающие.

В частности, для параллельного перемножения разряженных матриц и векторов, гиперграфовое разделение точно определяет объем коммуникационных затрат. Так при разделении матрицы по строкам в гиперграфовой модели за вершины принимаются столбцы матрицы, а гиперребра соответствуют строкам матрицы.

При разделении гиперграфа должны выполняться те же условия, что и для обычного разделения графа (4),(5). При этом разделение гиперграфа $G_{\mathcal{H}}(V, E)$ на n_p подобластей $P = (V_1, V_2, \ldots V_{n_p})$ будет сбалансированным если:

$$W_p \leq W_0(1 + \epsilon), \quad p = 1, 2, \ldots, n_p \tag{8}$$

здесь $W_p = \sum_{v_i \in V_p} \omega_i$; $W_0 = \left(\sum_{v_i \in V_p} \omega_i\right)/_{n_p}$; ϵ – заданная величина разбалансировки.

Гиперребро e_j считается считается разрезанным если оно связывает свя-

зывает более чем одну часть разделения ($\lambda_j > 1$, где λ_j – число частей связанных с e_j). На практике можно определить разные функции стоимости разделения гиперграфа

$$E_c(P) = \sum_{e_j \in N_E} \omega_{E_j}, \qquad E_c(P) = \sum_{e_j \in N_E} \omega_{E_j}(\lambda_j - 1), \tag{9}$$

где N_E – множество разрезанных гиперебер. Первая функция разделения в (9) минимизирует сумму весов разрезанных гиперребер. Вторая – число гиперребер принадлежащих нескольким частям разделения.

Задача нахождения гиперграфового разделения заключается в следующем: найти разделение такое, чтобы число разрезанных гиперребер (9) было минимальным при заданном критерии разделения по вершинам (8).

Гиперграфовое разделение показывает лучшие результаты по числу разрезанных гиперребер, но часто является значительно более затратным.

Последовательные алгоритмы гиперграфового разделения реализованы в таких библиотеках как [43, 45, 47]. Для параллельно выполняемого приложения имеет смысл, чтобы разделение выполнялось также параллельно. Алгоритмы параллельного гиперграфового разделения можно найти в [44, 48].

3.4. Динамическая балансировка и перераспределение нагрузки

Любой из алгоритмов разбиения графа, рассмотренных выше, может быть использован для динамической балансировки. Однако целью стандартных алгоритмов разделения графов является только получение множества разделителя. Для статической балансировки определяется только две принципиальных характеристики: качество и время выполнения.

Существенное отличие статической и динамической балансировки заключается в том, что в случае динамической балансировки после очередного разделения необходимо перераспределить нагрузку процессоров, что само по себе несет существенные затраты.

В большинстве случаев перераспределение происходит после оценки раз-

балансировки процессров и получения нового разделения. Однако, в некоторых работах [46] предпренимаются попытки прогноза разбалансированности приложения, что позволяет учитывать изменение в приложении при выполнении текущего этапа балансировки. Так в задачах с адаптивным перестроением сетки прогнозирование может быть основано на апостериорной оценке погрешности.

В том и другом случаях необходимость в приращении нагрузки добавляет третью ключевую характеристику динамической балансировке нагрузки. Эта проблема также многогранна и во многом определяется приложением и его структурами данных. Так, при решении трехмерных задач с адаптивным порядком аппроксимации (*p-версии* МКЭ) объем данных, связанных с каждым элементом, оказывается весьма большой. В то время, как для задач с адаптацией треугольных сеток (*h-версии* МКЭ) выгоднее при новом разделении сетки построить заново структуры данных элементов, чем пересылать эти данные. Для приложений, в которых потребность балансировки возникает постоянно, необходимо выбирать между затратами на модификацию структур данных и их полной передачей другому процессору, т.е. минимизировать функцию вида $|E_c| + S|V_m|$, где $|E_c|$ – число разрезанных ребер, определяемых по (6); $|V_m|$ – общая стоимость перераспределения данных между процессорами; S – мера стоимости связи.

В большинстве задач, требующих динамическую балансировку нагрузки, алгоритм балансировки реализуется непосредственно в приложении "программно", и необходимо связать структуру данных приложения и балансирующего нагрузку алгоритма.

Успешная реализация динамической балансировки из приложения возможна, если: программы балансировки нагрузки и приложение имеют простой пользовательский интерфейс; алгоритмы, обеспечивающие БН, сами выполняются параллельно. Представляется маловероятным, чтобы один алгоритм мог соответствовать всем требованиям и применяться для большего числа приложений. Поэтому хорошее программное обеспечение динамической балансировки должно содержать набор программ с разными

алгоритмами разделения для получения разделений высокого качества, другие – с минимальными пересылками и т.д. Технологии, реализующие такое обеспечение также могут быть различными. В некоторых случаях эффективным может оказаться подход, основанный на использовании модели клиент-сервер, когда один выделенный процессор управляет работой остальных процессоров. В этом случае реализуется централизованная динамическая балансировка. Как правило же, на практике, реализуется децентрализованная балансировка, и процессоры работают каждый над своей задачей. Методы в этом случае распараллеливаются с помощью MPI.

Такие свойства как простота, скорость и инкрементальность очень привлекательны для динамической балансировки. Алгоритмы динамической балансировки должны удовлетворять кроме двух рассмотренных ранее еще дополнительным требованиям:

- взаимодействовать с прикладной программой. Входные данные от приложения содержат текущее распределение сетки, параметры функции стоимости, определяющие затраты на вычисление, связь, время выполнения и т.д. На выходе полученное разделение сетки и его связь с предыдущим разделением. Перемещать данные прикладной программы согласно новому разделению.

- выполняться параллельно и не требовать много связей. При этом можно допустить, что качество разделения будет ниже чем у алгоритмов выполняющих статическую балансировку нагрузки.

- обеспечивать получение разделения, которое близко к предыдущему, и минимизировать перемещение данных из старого в новое разделение данных.

- необходимые запросы памяти для алгоритма балансировки минимальны, в противном случае возможны конфликты с приложением или ограничение объема приложения.

- обеспечение определенных конфигураций межпроцессорных связей. При получении нового разделения важно, насколько изменились взаи-

мосвязи подобластей сетки по сравнению со старым разделением. Для алгоритмов балансировки, основанных на геометрической информации о графе, такие изменения достаточно легко отследить, для других алгоритмов появляются дополнительные затраты на определение такой конфигурации.

В связи с этим логика динамической балансировки нагрузки может реализовываться в основном с помощью двух подходов:

- каждое разделение вычисляется при рассмотрении новой глобальной задачи разделения для всего графа или области. Использование в таких случаях алгоритмов статической балансировки приведет к почти оптимальному разделению, но стоимость перераспределения данных может быть очень высокой. Конечно, возможна адаптация алгоритмов исходя из приложения, возможности распараллеливания и т.д.

- текущее разделение модифицируется перемещением части данных к соседним процессорам – инкрементальный подход основан на локальных и распределенных решениях. Если нагрузка на процессор достаточно стабильна, то такая локальная процедура позволяет получать новое разделение, не сильно отличающиеся от первоначального разделения. В том случае, если наблюдается неустойчивость нагрузки, локальная процедура перемещения данных потребует большого числа приближений и дополнительных обменов для получения оптимального разделение.

На практике, конечно, более выгодно объединение обоих подходов при использовании условий перехода от одного подхода к другому в зависимости от приложения. Необходимо понимать, что реализация стратегии динамической БН обходится далеко небесплатно и беспокоится о стоимости каждого приема выравнивания нагрузки для подтверждения того, что преимущества производительности и масштабируемости, обеспечиваемые динамической балансировкой нагрузки, не перекроются дополнительными затратами. И здесь возникает вопрос, имеет ли смысл допускать некото-

рую несбалансированность нагрузки, если это уменьшает коммуникационные затраты. Конечно ответить на этот вопрос достаточно сложно. Можно только показать для конкретной МВС и определенной задачи или приложения.

К сожалению, развитие программного обеспечения для динамической балансировки значительно уступает обеспечению для статической балансировки. Можно назвать порядка 20 пакетов для статической балансировки и лишь четыре библиотеки методов динамической балансировки: ParJostle, ParMetis, Zoltan. Каждая из них имеет свою структуру данных, а в последних двух имеется интерфейс для ParJostle и ParMetis.

Во всех этих библиотеках содержится примерно один и тот же набор методов, но параллельное выполнение существенно отличается. Относительно выходных данных можно отметить наличие в Zoltan функций отображения полученного разделения на процессоры, тогда как в ParMetis вычисляется только новое разделение и пользователь сам должен выполнять отображение.

Сравнение методов балансировки на уровне приложения на примере библиотек ParMetis и Zoltan, Chaco, Jostle будет рассмотрено в следующем параграфе.

4. Сравнение методов балансировки нагрузки

Эффективность выполнения программ всегда являлась очень важным фактором, определявшим в значительной степени успех и распространение того или иного промежуточного программного обеспечения.

Динамическая балансировка предполагает выполнение двух шагов: вычисления нового разделения (или уточнения существующего разделения) сетки и перераспределения подобластей сетки. Особенность динамической балансировки вычислительной нагрузки состоит в том, что она выполняется над распределенными по процессорам данными.

Рассмотрим сравнение ряда алгоритмов динамической балансировки из библиотек ParMetis [51], Zoltan [48] и Chaco [49], ParJostle [50].

Алгоритмы, реализованные в библиотеке ParMetis, используют в основном многоуровневое графовое разделение. Для сравнения из библиотеки Zoltan были выбраны алгоритмы, основанные на геометрической информации: рекурсивной координатной и инерциальной бисекции, кривых заполняющих пространство. Алгоритм рекурсивной спектральной бисекции был выбран из библиотеки Chaco. Все алгоритмы, за исключением последнего, выполнялись параллельно при вызове из пользовательского приложения.

В качестве тестовой рассматривалась задача адаптивного перестроения сетки [52] на основе апостериорной погрешности для цилиндра, деформируемого под действием давления на его внутреннюю поверхность.

Сравнение алгоритмов динамической балансировки нагрузки при рассмотрении данной задачи проводилось по трем параметрам: дисбалансу вычислительной нагрузки, числу разрезанных ребер $|E_c|$ и числу пересылаемых конечных элементов. Первые два параметра характеризуют качество разделения сетки. Дисбаланс отвечает за равномерность вычислительной нагрузки во время вычислений, а число общих ребер – определяет объем межпроцессорных обменов. От третьего параметра зависит время выполнения перераспределения сетки. Из результатов сравнения отметим, что

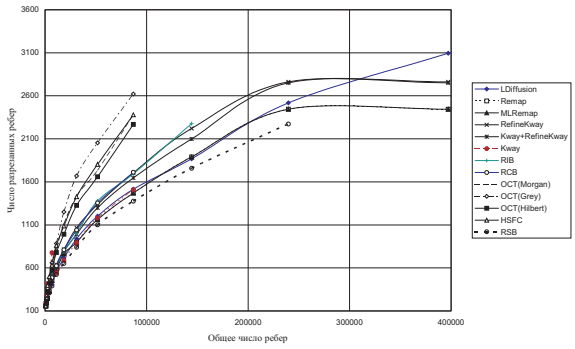

Рис. 1: Число разрезанных ребер E_c

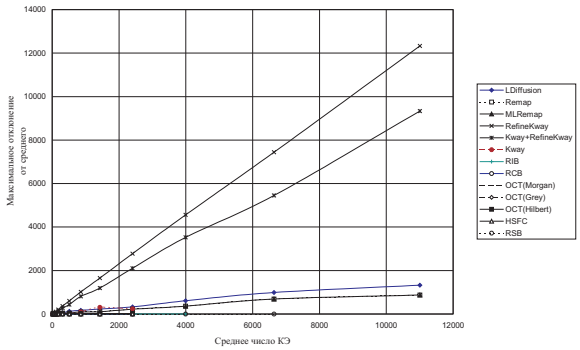

Рис. 2: Разбалансировка $\triangle W$

геометрические алгоритмы SFC (OCT(Morgan), OCT(Grey), OCT(Hilbert) создавали разделение сетки с большим в $1.5 - 2$ раза числом разрезанных ребер, чем алгоритмы основанные на графовом представлении (рис. 1). Другие алгоритмы, основанные также на геометрической информации о

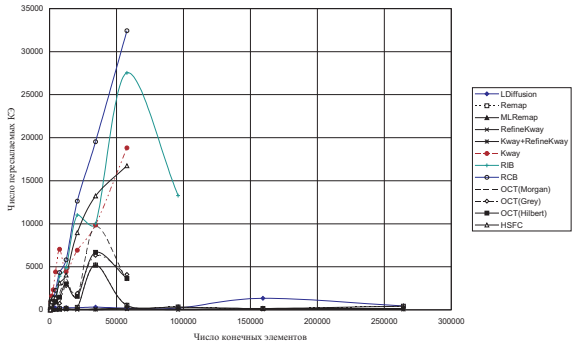

Рис. 3: Число пересылаемых КЭ по шагам перестроения сетки

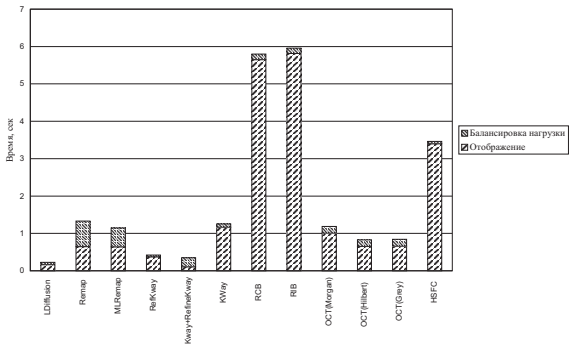

Рис. 4: Время выполнения одного шага

сетке, такие, как координатная (RCB) и инерциальная (RIB) бисекция, показали результаты близкие к тем, что получают графовые алгоритмы. Несколько алгоритмов показали близкие результаты. Перечислим их в порядке увеличения числа разрезанных ребер $|E_c|$: RSB – рекурсивная спектральная бисекция; Remap(MLRemap) – многоуровневая графовая бисекция с переотображением; LDiffusion – многоуровневая графовая бисекция с локальной диффузией. По второму критерию – наименьшому дисбалансу нагрузки (рис. 2) алгоритмы SFC, использующий кривые заполняющие пространство, показали наилучший результат (разбалансировка составила один конечный элемент). Анализ характеристик разделения сетки и затрат на отображение позволяет в дальнейшем выполнять динамическую балансировку нагрузки с учетом особенностей задачи и вычислительной системы. По результатам сравнения в качестве основного алгоритма для динамической балансировки был выбран алгоритм локальной диффузии.

При балансировке с использованием алгоритма локальной диффузии некоторые подобласти сетки разрывались. Отметим также, что топология связей подобластей, принадлежащих разным процессорам для метода LDiffusion, изменилась незначительно от начального разделения в отличии от других методов. Для вычислительных систем определенной архитектуры физических связей и реализуемых на них алгоритмов декомпозиции заданная связность подобластей имеет принципиальное значение.

Среди рассмотренных алгоритмов балансировки можно выделить мето-

ды (LDiffusion, Remap, Kway, RIB и др.), при применении которых получилось приблизительно одинаковое общее время выполнения приложения, но принадлежащие к разным группам (рис. 4). Методы имеют разные характеристики разделения и затраты на отображение. Анализ их позволяет более рационально проводить динамическую балансировку с учетом особенностей задачи и вычислительной системы. Рис. 2 характеризует качество разделения на каждом шаге перестроения. Наибольшая разбалансированность получилась при использовании метода RefineKway, а наименьшая – SFC (дисбаланс один конечный элемент). При идеальной загрузке процессоров $w_i = 1/n_p$, а качество балансировки определим например используя одну из мер:

$$\Delta W_1 = \frac{\max\limits_i \frac{|V_i|}{w_i}}{|V|}, \quad \Delta W_2 = \frac{\max\limits_i \frac{|V_i|}{w_i} - \min\limits_i \frac{|V_i|}{w_i}}{|V|}, \tag{10}$$

здесь $\Delta W_1 > 0$, а $\Delta W_2 \geq 0$.

Затраты на отображение перестроенной сетки и его нового разделения оценивалась по числу пересылаемых КЭ (рис. 3). Группа методов прежде всего основанная на геометрической информации о сетке, на каждом шаге перестроения сетки получает новое разделение без учета старого (рис. 4). Поэтому все сеточные данные приходиться пересылать согласно текущему разделению сетки. Если с сеткой связаны большие объемы данных, то перераспредение может увеличить накладные расходы адаптивного перестроения. Время адаптивного алгоритма включало в себя: формирование и решение основной и вспомогательной системы уравнений по оценке погрешности, время перестроения сетки, а также балансировку нагрузки (вычисление нового разделения) и отображение сетки. Основной вклад во временные затраты вносит решение систем. Эффективность параллельного адаптивного решения задачи в целом составила 63%. Наименьшие суммарные затраты оказались при применении простейшего геометрического алгоритма Kway и разделения LDiffusion, основанного на приращении вычислительной нагрузки. Затраты на отображение нового разделения на процессоры показаны на рис. 4. Отметим, что для алгоритма LDiffusion

они практически отсутствуют.

Таким образом, для задач с адаптивным перестроением сеток выполнение динамической балансировки является важным этапом и определяет успех решения всей задачи. Метод динамической балансировки основанный на локальной диффузии нагрузки является оптимальным при условии выбора хорошего начального разделения например, полученного PartKway. Использование для динамической балансировки нагрузки алгоритмов, основанных только на геометрической информации (RCB, RIB), также достаточно эффективно для рассмотренной геометрии задачи.

Безусловно, применимость того или иного подхода к балансировке нагрузки зависит прежде всего от класса решаемых задач, конкретного алгоритма метода декомпозиции матрично-векторных операций, а также от реализации самого параллельного пользовательского приложения, на эффективность которого оказывают влияние опыт разработчика.

Таким образом, при рассмотрении и сравнении существующих стратегий балансировки необходимо: тщательно проанализировать все преимущества и недостатки выбранного варианта балансировки нагрузки; учесть, что более простые подходы дают лучшие результаты; выбрать ту модель и метод балансировки нагрузки, которые больше всего подходит к конкретному приложению; стараться реализовывать новые алгоритмы в виде отдельных программных модулей или продуктов с удобным интерфейсом для того ПО с которым предполагается использовать пользовательское приложение.

Работа выполнена при поддержке РФФИ проекты №13-01-00101-а, №11-01-00275-а.

Список литературы

[1] *Barak A., Wheeler R.* MOSIX: An Integrated Multiprocessor UNIX // USENIX Conference Proceedings, San Diego, CA, January 1989. – P. 101–112.

[2] *Рычков В.Н., Красноперов И.В., Копысов С.П.* Промежуточное программное обеспечение для высокопроизводительных вычислений //

Вычислительные методы и программирование. – 2001. – Т 2, № 2. – С. 109–124.

[3] *Casas J., Clark D. L., Konuru R., Otto S. W., Prouty R. M. and Walpole J.* Mpvm: A migration transparent version of pvm // Computing Systems. – 1995. – V. 8, № 2. – P. 171–216.

[4] *Tan C.P., Wong W.F. and Yuen C.K.* tmPVM - Task Migratable PVM // Proceedings of the 2nd Merged Symposium IPPS/SPDP. April 1999. – P. 196–202.

[5] *Casas J., Clark D., Galbiati P., Konuru R., Otto S.,Prouty R., Walpole J.* MIST: PVM with Transparent Migration and Checkpointing. – Department of Computer Science and Engineering Oregon Graduate Institute of Science and Technology, May 1995. – `http://www.cse.ogi.edu/DISC/projects/mist`

[6] *Gehring J., Hendrikse Z. W., Iskra K. A. et al* Experiments with Migration of Message-passing Tasks // Lecture Notes in Computer Science. – 2000. – № 1971. – P. 203–213.

[7] *Czarnul P., Tomko K. and Krawczyk H.* Dynamic Partitioning of the Divide-and-Conquer Scheme with Migration in PVM Environment // Lecture Notes in Computer Science. – 2001. – № 2131. – P. 174.

[8] *Dikken L.,Linden F., Vesseur J., Sloot P.* DynamicPVM : Dynamic load balancing on parallel systems // Lecture Notes in Computer Science. – 1994. – № 797. – P. 273–277.

[9] *Stellner G.* CoCheck: Checkpointing and Process Migration for MPI // Proceedings of the International Parallel Processing Symposium, Honolulu, HI, April 1996. – P. 526–531.

[10] *Ластовецкий А.Л., Калинов А.Я., Ледовских И.Н., Арапов Д.М., Посыпкин Н. А.* Язык и система программирования для высокопроизводительных параллельных вычислений на неоднородных сетях // Программирование. – 2000. – №4. – С. 55–80.

[11] *Basney J., Livny M.* Managing Network Resources in Condor // Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing (HPDC9), Pittsburgh, Pennsylvania, 2000. – P. 298–299.

[12] *Foster I., Kesselman C.* Globus: A Metacomputing Infrastructure Toolkit // Int. J. Supercomputer Applications. – 1997. – V. 11, № 2. – P. 115–128.

[13] *Grimshaw A.* The Legion Vision of a Worldwide Virtual Computer // Communications of the ACM. – 1997. – V. 40, № 1. – P. 39–45.

[14] *Слама Д., Гарбис Д., Рассел П.* Корпоративные системы на основе CORBA. – М.: Издательский дом "Вильямс", 2000. – 368 с.

[15] *Russ S.H., Robinson J., Flachs B.K., Heckel B.* The Hector Distributed Run-Time Environment // IEEE Transactions on Parallel and Distributed Systems. – 1998. – V. 9, № 11. – P. 1104–1112.

[16] *Bhandarkar M., Kale L. V. , de Sturler E., Hoeflinger J.* Adaptive Load Balancing for MPI Programs // Lecture Notes in Computer Science. – 2001. – № 2074. – P. 108–117.

[17] *Demaine E.D.* A Threads-Only MPI Implementation for the Development of Parallel Programs // Proceedings of the 11th International Symposium on High Performance Computing Systems (HPCS'97), Winnipeg, Manitoba, Canada, 1997. – P. 153–163.

[18] *Tang H., Shen K., Yang T.* Compile/Run-time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines // Proceedings of 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), 1999.

[19] *Fagg, G., Dongarra, J.* HARNESS Fault Tolerant MPI Design, Usage and Performance Issues // Future Generation Computer Systems. – 2002. – V. 18, № 8. – P. 1127–1142.

[20] *Kale L. V., Krishnan S.* Charm++: Parallel Programming with Message-Driven Objects // Parallel Programming using C++ editors, G. V. Wilson and P. Lu. – MIT Press, 1996. – P. 175–213.

[21] *Аветисян А.И., Арапов И.В., Гайсарян С.С., Падарян В.А.* Параллельное программирование с распределением по данным в системе ParJava // Вычислительные методы и программирование. – 2001. – Т. 2. – С. 88–108.

[22] *Коновалов Н.А., Крюков В.А., Михайлов С.Н., Погребцов А.А.* Fortran DVM – язык разработки мобильных параллельных программ // Программирование. – 1995. – № 1. – С. 49–54.

[23] *Альес М.Ю., Копысов С.П., Варнавский А.И.* Моделирование структуры материала многранниками Вороного // Применение математического моделирования для решения задач в науке и технике. – Ижевск.: Изд–во ИПМ УрО РАН, 1996. – С. 32–43.

[24] *Metropolys N., Rosenbluth A.W., Rosenbluth M.N., Teller A.H., Teller E.* Equation of state calculation by Fast Computer Machines // J.Chemical Physics. – 1953. – V. 21. – P. 1087–1092.

[25] *Kirkpatric S., Gelatt C.D. Jr., Vecchi M.P.* Optimization by Simulated Annealing // Science. – 1983. – V. 220. – P. 671–680.

[26] *Goldberg E.* Genetic Algorithms in search, optimization and machine learning. – Addison-Wesley, 1989.

[27] *Осовский С.* Нейронные сети для обработки информации. – М.: Финансы и статистика, 2002. – 344 с.

[28] *Копысов С.П., Новиков А.К.* Параллельные алгоритмы адаптивного перестроения и разделения неструктурированных сеток // Матем. моделирование. – 2002. – Т. 14, № 9. – С. 91–96.

[29] *Копысов С.П.* Динамическая балансировка нагрузки для параллельного распределенного МДО // Методы и средства обработки информации / Под ред. Л.Н. Королева. – М.: МГУ, 2003. – С. 222–227.

[30] *Kopysov S. P., Novikov A. K.* Parallel adaptive mesh refinement with load balancing on heterogeneous cluster // Algorithms and Tools for Parallel Computing on Heterogeneous Clusters: Nova Science Publishers, 2007. – P. 45–53.

[31] *Копысов С. П., Кузьмин И. М., Недожогин Н. С., Новиков А. К.* Параллельные алгоритмы формирования и решения системы дополнения Шура на графических ускорителях // Ученые записки Казанского ун-та. Серия Физико-матем. науки. – 2012. – Т. 154, Кн. 3. – С. 202–215.

[32] *Gilbert J. R., Miller G.L., Teng S.-H.* Geometric mesh partitioning // Technical Report CSL-94-13. – Xerox PARC, 1994.

[33] *Препарата Ф., Шеймос М.* Вычислительная геометрия: Введение. – М.: Мир, 1989. – 478 с.

[34] *Александров П.С.* Введение в общую теорию множеств и функций. – М.: Гостехиздат, 1948.

[35] *Fiedler M.* A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory // Czechoslovak Mathematical Journal. – 1975. – V. 25. – P. 619–633.

[36] *Четверушкин Б.Н.* Высокопроизводительные многопроцессорные вычислительные системы // Вестник РАН. – 2002. – Т. 72, № 9. – С. 786-794.

[37] *Четверушкин Б.Н.* Кинетические схемы и квазигазодинамическая система уравнений. – М.: Макс Пресс, 2004. – 332 с.

[38] *Корнилина М.А., Якобовский М.В.* Динамическая балансировка загруз-ки процессоров при моделировании задач горения // Материалы Всероссийской научной конференции "Высокопроизводительные вычисления и их приложения" Черноголовка-2000. – Москва: МГУ, 2000. – С. 34-38.

[39] *Kernighan B.W., Lin S.* An efficient heuristic procedure for partitioning graphs // The Bell System Technical Journal. – 1970. – V. 29, № 2. – P. 291–307.

[40] *Fiduccia C. M., Mattheyses R. M.* A linear time heuristic for improving network partitions // Proceedings of the 19th IEEE Design Automation Conference, 1982. – P. 175-181.

[41] *Hendrickson B., Leland R.* A multilevel algorithm for partitioning graphs // Tech. Rep. SAND93-0074. – Sandia National Laboratory, Albuquerque, NM, 1993.

[42] *Karypis G., Kumar V.* Multilevel k-way partitioning scheme for irregular graphs // J. Parallel Distrib. Comput. – 1998. – V. 48. – P. 96–129.

[43] *Catalyurek U. V., Aykanat C.* PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0. – Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey, 1999.

[44] *Trifunovic A., Knottenbelt W. J.* Parkway 2.0: A parallel multilevel hypergraph partitioning tool // Proceedings of the 19th International Symposium on Computer and Information Sciences, 2004.

[45] *Vastenhouw B., Bisseling R. H.* A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication // SIAM Review. – 2005. – V. 47, № 1. – P. 67–95.

[46] *Андрианов А.Н., Березин А.В., Воронцов А.С., Ефимкин К.Н., Марков М.Б.* Моделирование электоромагнитных полей радиационного происхождения на многопроцессорных вычислительных системах // Препринт ИПМ РАН № 74. – Москва, 2006. – 20 с.

[47] *Karypis G., Kumar V.* Multilevel k-way Hypergraph Partitioning // VLSI Design. – 2000. – V. 11, № 3. – P. 285–300.

[48] *Devine K., Hendrickson B., Boman E., John M. St., Vaughan C.* Zoltan: A Dynamic Load-Balancing Library for Parallel Applications; User's

Guide // Tech. Rep. SAND99-1377. – Sandia National Laboratories, Albuquerque, NM, 1999.

[49] *Hendrickson B., Leland R.* The Chaco user's guide: Version 2.0 // Tech. Rep. SAND94-2692. – Sandia National Laboratories, Albuquerque, NM, 1994.

[50] *Walshaw C.* Parallel Jostle. User guide version 1.2.9. // University of Greenwich, London, UK, 1998.

[51] *Karypis G., Schloegel K., Kumar V.* ParMetis: Parallel Graph Partitioning and Sparse Matrix Ordering Library. Version 2.0. // Technical Report, Department of Computer Science, University of Minnesota, 1998.

[52] *Копысов С. П., Новиков А. К.* Метод декомпозиции для параллельного адаптивного конечно-элементного алгоритма // Вестник Удмуртского ун-та. Математика. Механика. Компьют. науки. – 2010. – № 3. – С. 141–154.