САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
КАФЕДРА КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ И МНОГОПРОЦЕССОРНЫХ СИСТЕМ

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Тема: «Разработка эффективного алгоритма запуска вычислительных задач на кластере на основе распределенного конвейера»

Направление: 02.04.02 – ФИиИТ

Магистерская программа: ВМ.5502.2014 – Вычислительные технологии

Выполнил студент гр. 633 Типикин Юрий Александрович

Научный руководитель,

д. т. н., профессор А. Б. Дегтярев

Санкт-Петербург – 2016

Оглавление

Введение 3

Постановка задачи 6

Обзор литературы 7

Глава 1. Формализация журнала 10

Глава 2. Модель актеров 15

Глава 3. Модель вычислительных ядер 21

3.1. Определения 21

3.1.1. Программа-пример 23

3.2. Правила разбиения 24

3.2.1. Разбиение программы-примера 26

3.3. Иерархия вычислительных ядер 28

Глава 4. Имплементация вычислительных ядер 33

4.1. Ведение журнала 35

4.2. Вычислительный сервер, управляющий задачи 39

4.3. Работа модели 40

4.4. Тестирование 44

Выводы 46

Заключение 48

Список литературы 49

Введение

Современные проблемы вычислений на кластерах

Сегодня никого не удивить существованием больших суперкомпьютеров, составляющих основу для ЦОДов. Свои собственные вычислительные системы (кластера) есть у лабораторий, университетов, небольших частных компаний. Их повседневное использование и обслуживание — предмет интереса широкого круга людей. Мой же интерес к теме HPC (high performance computing, высокопроизводительные вычисления) формировался все последние годы. Мне, правда, не интересен сам факт существования подобных вычислительных систем – мне интересны проблемы, с которыми сталкиваются их пользователи. Проблемы высокопроизводительных вычислений можно грубо разделить на три группы:

- В первую можно отнести проблемы производительности оборудования. Сюда входят всевозможные алгоритмы оптимизации и практики использования графических ускорителей при вычислениях, производительность сети (локальная/распределенная), производительность центральных процессоров, взаимодействие между устройствами по шине обмена данными, производительность подсистемы оперативной памяти и т.д.

- Вторая группа содержит проблемы надежности. Надежность в данном контексте стоит понимать как свойство аппаратуры и ПО, которое позволяет выполнить задачу вне зависимости от характера и степени внешнего или внутреннего противодействия. К негативным воздействиям относятся аппаратные ошибки и сбои, логические ошибки программной реализации и т.д.

- В третью группу можно отнести проблемы алгоритмического характера, которые можно описать следующим вопросом: как наилучшим образом реализовать задачу в виде программы для ЭВМ, задействовав при этом вычислительный ресурс максимально эффективно?

В среде computer science проблемы первой и третьей группы намного популярнее проблем второй группы. Например, запросы в Google Scholar “cluster computation performance” и “cluster computation reliability” дают результаты в 1300000 и 513000 ответов соответственно. Почему такая большая разница в количестве ответов? Скорее всего потому, что прямая и явная выгода от прироста производительности очевидна всем, а повышение надежности будет иметь вес только в определенном круге задач. Так, если задача рассчитывается быстрее, то в выигрыше все: и тот, кто считает и те, кто предоставляет ресурсы. Но что касается повышения надежности?

Степень надежности процесса вычислений исторически принято связывать с надежностью оборудования. С одной стороны, это логично: если техника работает штатно, то внутренним программным процессам не угрожают фатальные ошибки. Здесь стоит уточнить, что программные фатальные ошибки, например, некорректный доступ к общей памяти, остаются на совести тех, кто целенаправленно их добивается. Сегодня существует достаточно уровней программных разграничений, чтобы исполняемый код не привел к непредсказуемым последствиям.

Но вернемся к технике. Для обеспечения аппаратной надёжности за последние 50 лет разработано достаточное количество методик. В общем, они сводятся к резервированию ресурсов и замещению процессов, подвергнутых сбою. Так, если выходит из строя один жесткий диск, то проще будет перенаправить потоки данных на резервный, ведь таких резервных дисков еще N штук. Тоже — с целыми вычислительными узлами.

Подобные методы резервирования ресурсов оправданы, если стоит цель сохранить статические данные. Здесь под статическими данными я понимаю такие массивы информации, скорость изменения которых не требует их постоянного хранения в оперативной памяти, или такие, которые настолько велики, что могут обработаться только частями, при этом операций чтения с диска много больше операций записи на диск. Потеря таких данных в случае аварии будет худшим из сценариев, потому что эти данные — предмет вычислений. Что же до самого процесса расчета, то его начинают заново. С момента, когда требуется полный перезапуск задачи, начинается мое исследование.

Перезапуск задачи всегда влечет за собой потери. Прежде всего это временные потери, которые в худшем случае составляют полное время выполнения задачи до поломки. Если расчет идет на множестве узлов и сам алгоритм решения задачи реализован с помощью технологий обмена сообщениями, такими как MPI/OpenMP, то вероятность перезапуска всей задачи значительна, потому значительное негативное влияние имеет выход из строя любого узла по любой причине. Временные потери при повторном вычислении ранее рассчитанных, но прерванных операций, приводят к неэффективному использованию кластера в целом. Так, уменьшается показатель эффективности “задача/час”, повышается энергопотребление и т.д. Какие существуют способы этого избежать?

Постановка задачи

Рассмотрим вычислительную задачу, значительную по объему операций. Пусть ее алгоритм содержит независимые участки и его возможно распределить на узлы кластера. Тогда необходимо реализовать такой алгоритм расчета задачи в распределенной среде, который мог бы восстановиться после выхода одного, нескольких или всех узлов кластера из строя.

Эффективность алгоритма будет измеряться отношением суммы времени восстановления T_r и времени выполнения до сбоя T_e ко времени выполнения задачи без сбоя T_n. Алгоритм считается эффективным, если такое отношение будет меньше двух:

$$E = \frac{T_r + T_e}{T_n} < 2.$$

Так как среда выполнения задачи распределенная, алгоритм должен учитывать влияние процессов, протекающих в подобных средах, например, несогласованность времени, затраты на передачу данных по сети и т.д. Резюмируя, искомый алгоритм:

1. Должен быть применим к широкому кругу существующих задач.
2. Должен минимизировать внешние воздействия на общее время расчета задачи.
3. Должен быть сопоставимым или лучше существующих способов распределения задач по общему времени выполнению.
4. Должен учитывать динамическое изменение конфигурации кластера.

Обзор литературы

Приступая к решению поставленной задачи, я попытался в начале найти методы или модели, которые максимально соответствуют установленным ранее требованиям. Так, я выделил две условные группы алгоритмов, борющихся с проблемой прерывания вычислений. К первой группе я отнес подходы, которые пытаются встроить процесс восстановления в уже имеющиеся библиотеки. Основная идея алгоритмов первой группы в том, чтобы не переделывать существующие программы. Вторая группа состоит из подходов, которые отказываются модифицировать известные библиотеки MPI/OpenMP и предлагают решение проблемы в принципиально отличных алгоритмах.

В [1] дан хороший обзор подходов и техник, которые призваны повысить надежность вычислений. Авторы делят известные алгоритмы на две группы: те, что решают проблему восстановления после сбоя и те, которые пытаются не допустить программный сбой из-за сбоя аппаратного.

Например, в первую группу включены такие техники:

1. Перезапуск задачи — самый простой способ восстановить. расчет [2].
2. Перемещение задачи на другой узел.
3. Контрольные точки [3].

Во второй группе авторы отметили следующие методы.

1. “Программное омоложение” [4], когда система периодически полностью перезагружается.
2. “Упреждающая миграция”, когда процесс наблюдатель может заранее выполнить миграцию задачи на другие узлы, относительно текущей статистики.

3. “Самолечение” узлов — все процессы запускаются в виртуальных машинах, одновременно с этим запускаются копии этих процессов, также в виртуальных машина. В случае необходимости они подменяются.

Статья является хорошим стартом в деле выбора механизма восстановления.

Примером подхода, относящегося к первой группе является архитектура приложений RADIC [5]. Так, восстановлению поддаются MPI процессы, которые в случае сбоя перезапускаются на другом активном узле. Очевидно, при таком подходе восстанавливаемые процессы могут перегрузить узел, на котором они восстанавливаются. Авторы не упоминают о таких ситуациях и не указывают способы их избежать. Сам процесс сохранения состояния выглядит следующим образом: к каждому активному вычислительному процессу добавляется отдельный процесс, который по таймеру сохраняет его состояние на соседние узлы. При этом, какие именно данные сохраняются не указано, возможно это данные процесса в оперативной памяти. Подход позволяет не изменять код уже готовых MPI программ, однако накладные расходы при работе такого алгоритма существенны.

Мне изначально были более интересными методы второй группы. После некоторого поиска, я вышел на модель параллельных вычислений — “модель актеров”. Статьи по этой теме делятся на ранние, теоретически и поздние, практические. Так, очень подробно описано применение модели для параллельных вычислений в [6]. У автора этой диссертация научным руководителем был К. Хьюит, создатель модели и автор оригинальной статьи [7]. Так, в [6] подробно описывается внутренняя структура актера, а также его жизненный цикл. При этом такие компоненты как асинхронная “почтовая” очередь рассматриваются только в теории.

С распространением многоядерных систем, многие исследователи вновь обратились к модели актеров. Ее преимущество в том, что она дает ясные и простые формулировки процессам, протекающим в распределенной системе. Например, такой популярный язык программирования как Scala, по словам ее автора, основывается на модели актеров. Однако, как выяснили авторы [8], в 80% проектов, реализованных на Scala и при этом декларирующих, что они реализуют модель актеров, используется не одна, а несколько вычислительных моделей. Этот факт является следствием того, что иногда очень сложно переосмыслить какой-либо функционал в логике «актеров», и его реализуют "как умеют". Такое смешение, правда, ведет к росту количества блокировок (deadlocks) во время работы программы. Ответ на вопрос о совмещении разных параллельных моделей в рамках одного проекта, например, при включении Scala компонентов в Java, проект дается в [9].

Агха, автор [6], не потерял интерес к дальнейшим исследованиям модели актеров и ее приложений и не так давно написал обзорную статью [10], где сравнил различные JVM (Java Virtual Machine) фреймворки-реализации модели. Здесь акцент смещается от абстрактных определений к преимуществам и недостаткам конкретной реализации в рамках JVM.

Я не мог не добавить в обзор статью [11], соавтором которой я являюсь. В ней описаны промежуточные результаты работы группы нашей кафедры над проблемой восстановления прерванных вычислений. Так, представлена реализация алгоритма (модель), которая способна достаточно быстро восстановить процессы убывшего узла на оставшихся, при этом избегается переполнение узлов восстановленными процессами, за счет иерархии узлов кластера. Также, узлы в иерархии не требуют дополнительных операций для своей настройки в момент их добавления в ранее существующий вычислительный кластер.

Глава 1

Формализация журнала

Так или иначе, решение проблемы перезапуска вычислений достигается за счет введения «журнала». Под журналом я понимаю место на устройстве или устройствах постоянной памяти, которое обслуживает процесс, который может сохранить достаточно значимой для возобновления расчетов информации, при этом, сам по себе такой процесс также должен являться надежным. Так, системы управления ресурсами, вроде PBS Pro, могут создавать контрольные точки восстановления (checkpoint) процессов с помощью сторонних библиотек вроде BLCR [12]. Эти точки представляют из себя сохраненные на диск срезы оперативной памяти параллельных процессов. Такие алгоритмы восстановления существенно влияют на производительность, потому что требуют временной остановки расчета перед выгрузкой данных из памяти и времени для записи среза на диск. В кластерах из множества узлов падение производительности будет увеличиваться с ростом количества узлов.

Чтобы избежать роста издержек с ростом узлов, журналирование должно происходить на фоне активной задачи, оно должно быть постоянным, а за единицу журнала должен быть принят объект небольшой по размеру, но, в тоже время, способный исчерпывающе описать темпоральное состояние исполняемой задачи. Для простоты будем далее называть подобный объект «записью» в «журнале операций». Но как получить подобного рода запись?

Рассмотрим обобщенную задачу для кластера. Она состоит из данных или ссылок на них и обрабатывающего алгоритма. Алгоритм реализован на одном из языков программирования и имеет возможность считывать и

сохранять данные в процессе своего исполнения. Имплементация предполагает наличие программных кодов с последовательными и, если есть такая возможность, параллельными участками вызовов вычислительных операций. Рассмотрим скомпилированные коды. Можно ли здесь выделить минимальную запись для журнала? Я думаю, это будет большой проблемой, так как при переходе на более низкий языковой уровень теряется логическая структура верхнего уровня, а если рассматривать все операции как одну запись, то такой подход будет аналогичен контрольным точкам. В процессе выполнения кода, влиянию подвергается только виртуальная память его процесса в ОС, поэтому механизму контрольных точек приходится ее полностью выгружать. Здесь ни о какой минимизации данных для журнала говорить не приходится. Значит, нужно, чтобы искомая запись журнала предоставлялась не внешним, по отношению к этому процессу событием, а внутренним; другими словами, программа должна сама обеспечить создание искомой записи, что ведет к необходимости модификации исходного кода программы.

Изменять исходный код можно явно или неявно. Явное изменение — использование при имплементации алгоритма методов сторонних библиотек и расширений. Так, явным изменением кода будет использование специальной библиотеки журналирования, например, Log4j для программ на языке Java, которая будет записывать выводимую информацию в своем собственном формате, и сама же контролировать процесс записи. Неявное изменение — запуск программы через посредников (агентов). Здесь в процессе имплементации не будет явных вызовов методов сторонних библиотек, тем не менее, могут быть вызваны методы сервера с нужным набором логики. Примером неявного изменения могут служить веб-сервисы или вызовы удаленных процедур, которые могут записывать запрос в виде состояния запрашивающего процесса.

В случае, когда вычисления реализуются на кластерах, использование неявной схемы приведет к неэффективному использованию ресурса, ведь необходимо резервировать ресурсы специально для серверов журналирования, которые, в свою очередь, сами не защищены от аппаратных ошибок. Кроме этого, дополнительный сетевой обмен снижает пропускную способность канала для передачи программных данных. Можно сделать вывод, что в кластерных средах использование явного метода изменения кода предпочтительнее, за счет того, что затраты на обслуживание процесса, исполняющего код, уже присутствуют и нет необходимости их увеличивать за счет дополнительных процессов-сателлитов. И вот почему: добавление методов в уже созданный процесс не требует от ОС затрат на создание полной инфраструктуры виртуальной памяти процесса, на обеспечение работы операций ввода/вывода, на маршрутизацию данных и т.д. Говоря о процессе, я подразумеваю абстракцию, которая включает себя задачу целиком. В зависимости от реализации, программа может быть последовательной, параллельной, распределенной или комбинированной и процесс будет одним только в первых двух случаях. Уточню, что здесь под «параллельной» реализацией подразумевается использование технологии потоков, когда параллелизм достигается в рамках одного узла без создания дополнительных процессов программы.

Примером явной модификации кода при написании параллельной программы является использование библиотек MPI/OpenMP. На сегодняшний день их использование является де-факто стандартом при написании программ для кластерных окружений, как и использование языков C/C++. Эти библиотеки обеспечивают обмен сообщениями между потоками/процессами/узлами за счет использования небольшого количества специальных функций и аннотаций. Но что есть использование аннотаций в коде программы? Это есть простое структурирование, явное выделение парал-

лельных и последовательных частей алгоритма. Но целью MPI/OpenMP является управление сообщениями между параллельными частями, библиотеки не обеспечивают надежность процесса выполнения. Если в момент работы алгоритма, реализованного с их помощью, произойдет системная или аппаратная ошибка, то выполнение всех процессов задачи на всех узлах придется остановить и запустить заново. Понятно, что такое развитие событий многих не устраивало, поэтому были разработаны модификации библиотек, которые, в общем, реализовывали журнал по принципу контрольных точек. В обзоре литературы я описал одну такую модификацию [5].

В случае MPI/OpenMP, структурирование кода служит для обособления параллельных участков алгоритма, но что если явно и жестко структурировать код всей задачи? Подобное структурирование, или дробление задачи, в общем случае, даст нам искомую запись для журнала в виде состояния «дроби». Описание отображения «дроби» будет дано далее. Дробление алгоритма можно осуществлять по-разному, и, если не учитывать его исходную логическую структуру, «дроби» могут неэффективно отражать текущее состояние задачи в целом.

Для оптимального разбиения исходного алгоритма нужно найти в нем условные «компоненты связности», внутри которых операции логически связаны и не могут быть разделены. Очевидно, что внутри таких компонент операции выполняются последовательно. Назовем такую компоненту вычислительным ядром (ядром). После нахождения всех «компонент» записью для журнала операций будет текущее состояние каждого вычислительного ядра. А так как ядра между собой независимы, то записывать изменение состояния каждого можно параллельно, в фоне, не мешая выполнению остальных.

Дальнейшее обсуждение вычислительных ядер я построю в виде от-

ветов на следующие вопросы:

- Каков принцип выделения ядер в алгоритме?
- Как описать взаимодействие между ядрами?
- Как с помощью ядер реализовать параллельный алгоритм?
- Как реализовать вычислительное ядро?
- Как преобразовать ядро в журнальную запись?
- Как ядра помогают восстанавливать процессы при сбоях?

Совокупность ответов на эти вопросы составляет описание эффективного алгоритма запуска задач на кластере, который, в свою очередь, является переосмыслением модели актеров К. Хьюита.

Глава 2

Модель актеров

Первые статьи по данной модели появились в начале 70-ых годов 20 века и были направлены на исследование искусственного интеллекта. Так, оригинальная статья К. Хьюита и Г. Бейкера вышла в 1977 году [7]. Однако скоро акцент применения модели сместился в сторону распределенных вычислений. Диссертация [6] Г. Агха, описывающая модель актеров в применении к такого рода вычислениям, вышла в 1983 году. Тогда еще не существовало ни MPI, ни настоящих, действительно распределенных систем и даже сама сеть в ее современном виде только появилась. Тем не менее, это не мешало обсуждать всевозможные теоретические способы создания систем, способных обрабатывать данные последовательно.

Так, суть модели актеров очень проста — распределенные вычисления нужно выполнять в среде гомогенных независимых узлов, каждый из которых либо может выполнить операцию сам, либо переслать ее другому. Операции объединены в «задания», которые имеют уникальный заголовок, цель — адрес актера и сообщение, которое актер прочитает из своего почтового ящика, когда задача туда попадет. При этом, сообщение внутри себя может содержать «задачи» для других актеров, которые в ходе выполнения будут «разосланы». То есть, идея модели в том, что в ней все есть актер — и узлы, и сообщения, передаваемые между ними. Такой подход позволяет реализовать т.н. вызов-по-запросу, используя «сообщения» и их содержимое в качестве «обещания» (promise). Обещание будет исполнено тем актером, которому было передано сообщение с задачей, когда он его обработает.

Особенности модели:

- Гомогенность вычислительных узлов
- Асинхронное выполнение задач
- Нативное распределение актеров по узлам

Рассмотрим внутреннею структуру актера 2.1. Ранее, я частично упоминал о его структуре, но не вдавался в подробности. Актер представляет из себя совокупность очереди, которая здесь именуется почтовым ящиком, конвейера, обрабатывающего сообщения в почтовом ящике, «адреса» и состояния. Так как у каждого актера есть условный адрес, то между собой они могут обмениваться сообщениями напрямую. Конвейер задач исполняет задачи из почтовой очереди и изменяет состояние всего актера.

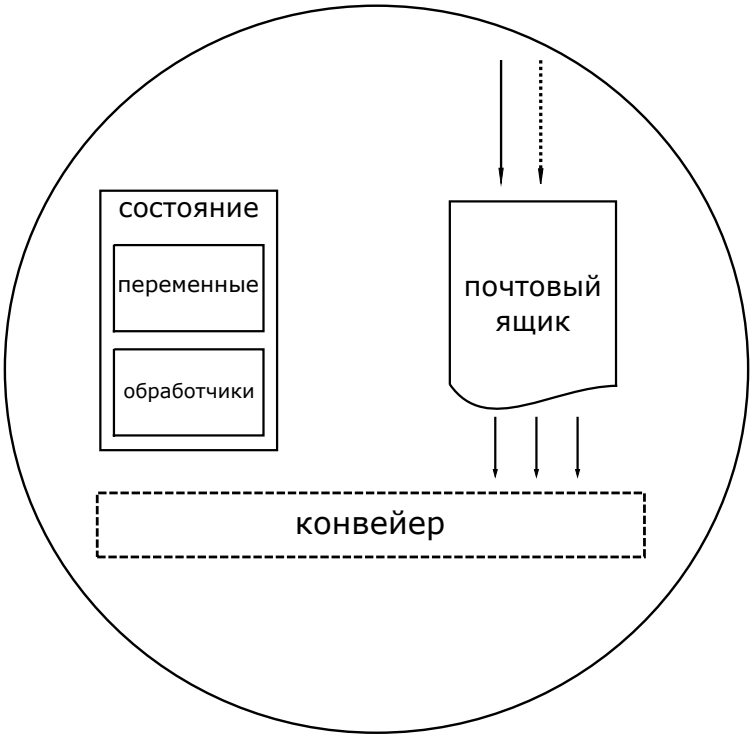

Рис. 2.1. Актер

Взаимодействие актеров происходит через обработку «заданий». В каждом задании есть целевой актер, в отношении которого необходимо

применить сообщение. Верификация цели происходит по следующему алгоритму:

1. Цель является ранее известной для актера.

2. Цель становится известной в момент получения актером сообщения с задачей, содержащей цель.

3. Цель является адресом актера, который будет создан после выполнения пришедшего сообщения.

Таким образом сеть адресов актеров является заранее известной, связной, а ситуация, когда актер получает корректное сообщение с неизвестным адресом, невозможна.

Вычисления в данной модели протекают за счет обработки сообщений в «заданиях» актерами. То есть, процесс вычислений является «управляемым данными» (data-driven), а не процессами (process-driven). Каждый актер обрабатывает только те задачи, которые попали в его почтовый ящик, при этом в процессе обработки ему также необходимо поддерживать «замещающее поведение».

Внутри почтовой очереди сообщения имеют порядок, в котором они вошли. Отдельно в оригинальных работах обсуждается механизм имплементации условно безграничной почтовой очереди, которая может обрабатывать сообщения, поступающие одновременно. Сегодня такая обработка полностью реализуется транспортными уровнями ПО с помощью буферизации. Описанный ранее конвейер задач определяет поведение актера. В общем случае оно представляет собой функцию обработки входящих сообщений. Так, если актер принимает сообщение и оно попадает на конвейер выполнения, происходят последовательно несколько действий:

1. Указатель очереди помешается на текущее необработанное сообщение.

2. Далее, процесс обработчик:

   а. Создает замещающую функцию-обработчик, чтобы она продолжила обрабатывать сообщения почтового ящика.

   б. Если есть указания, создает задание (задания) в соответствии с исходным сообщением.

   в. Если есть указания, создает новых актеров.

3. Процесс-обработчик завершает свою работу, замещающая функция становится основной.

Жизненный цикл конвейера задач можно представить следующей иллюстрацией 2.2. Обладая таким жизненным циклом актеров, модель позиционируется как высоко параллельная, с «наследуемым» параллелизмом порожденных актеров. И, так как практической реализации на момент написания оригинальных статей не последовало, авторы на практике не сталкивались с проблемами подобного рода моделей.

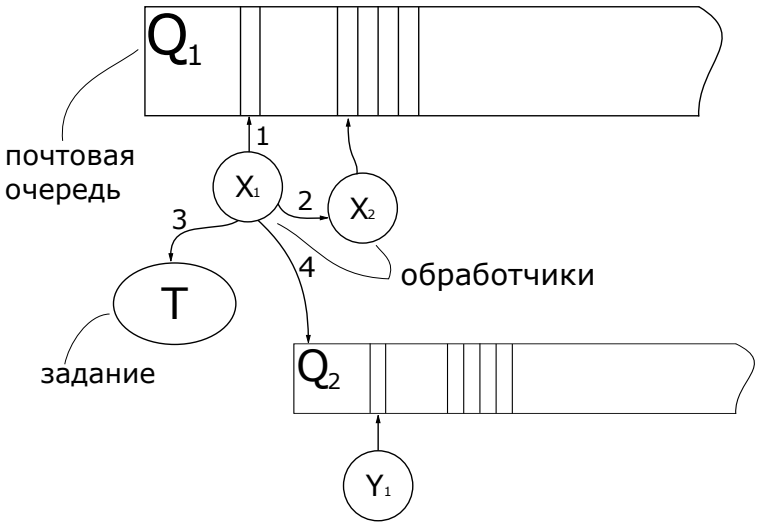

Рис. 2.2. Жизненный цикл актера

Лишь с началом массового распространения многоядерных/многопроцессорных систем в конце 1990-ых модель имплементировали и стали использовать для вычислений. Например, измененная идея актеров лежит в основе таких языков программирования как Erlang и Scala [13]. На сегодняшний день существует множество имплементаций базовой модели актеров в виде фреймворков для разных языков программирования.

Одним из самых популярных является фреймворк Orbit для языка Java. Он реализует актеров в рамках процессов JVM (Java Virtual Machine), которые в любой момент времени могут быть активными или нет. Актеры обмениваются сообщениями асинхронно и вместе составляют распределенную систему. Включение и выключение актера зависит от приходящих ему сообщений и нацелено на экономию ресурсов. Состояние актера хранится вне него, например, в базе данных. Фреймворк гарантирует, что в системе одновременно может быть активна только одна копия актера, и что обращения к актерам выполняются последовательно. В качестве основы для работы актеров Orbit [14] предлагает окружение выполнения «сцена» (stage), через которую разработчику становятся доступны актеры. На самом деле сцена есть ни что иное как сервер для актеров-процессов. Предполагается, что на каждый узел в кластере будет установлена только одна «сцена». В отличии от исходного описания модели актеров, «задания» в Orbit являются возвращаемым значением после передачи асинхронного сообщениями и выполнения действий, то есть принцип «все есть актер» здесь в полной мере не соблюдается.

Рассмотрим модель с точки зрения надежности. В данной модели никак не рассматривается вопрос надежности актеров, надежности передачи сообщений между ними, надежности результатов расчета. Фактически, процесс вычислений невозможно восстановить никаким образом, кроме его полного перезапуска. Из-за того, что любой актер может взаимодейство-

вать с любым другим, в случае ошибки в одном из них, нельзя будет определить от кого пришло «задание» и как его восстановить. В модели отсутствует иерархия процессов, иерархия актеров, нет никаких связей между порождаемыми заданиями и заданиями-«родителями», хотя в реальных приложениях подобная связь всегда есть. Возможно, это отголоски первоначального замысла модели, но применительно к высокопроизводительным вычислениям неиерархичность модели ведет к проблемам при сбоях.

Далее, введение иерархии заметно упрощает процесс распределения ресурсов [15]. Рассмотрим случай простого бинарного дерева — если один из физических узлов порождает множество подзадач во время счета, то он может делегировать их выполнение своим соседям по дереву и далее к листьям. Тот же случай, но без иерархии приведет к заметным временным тратам при нахождении наименее загруженного узла, потому что сведется к полному перебору. Хотя, стоит отметить, что в оригинальных статьях на тему модели актеров проблема распределения задач вообще не рассматривалась.

Основная идея модели актеров с одной стороны довольно проста, но в тоже время фундаментальна. Явное разделение участников процесса вычислений на независимые узлы, явное описание процесса их взаимодействия и накладываемые на него ограничения позволяют преобразовать почти любой алгоритм в набор заданий и актеров. Тем не менее, надежной эту модель назвать нельзя по описанным выше причинам. Однако, можно использовать ее как основу для нового механизма, который, не нарушая базовых принципов, сделает возможным сохранение и восстановление состояния элементов системы в любой момент времени после технического сбоя.

Глава 3

Модель вычислительных ядер

3.1. Определения

Чтобы решить первоначально поставленную задачу надежности, я решил модифицировать модель актеров так, чтобы стало возможным восстанавливать работу системы после сбоя. Назовем новую модель «моделью вычислительных ядер». Основное отличие разрабатываемой модели — наличие иерархии. Причем иерархия вводится как внутри процесса расчета задачи, так и между вычислительными узлами. Для большего понимания отличий моделей ядер и актеров, введем следующие определения.

Определение 3.1.1. *Вычислительное ядро(ядро) — минимальная единица модели, но в то же время основная. Представляет из себя объект, который содержит информацию о своем состоянии, переменные и последовательный набор методов, определяющих его поведение. Набор вычислительных ядер формирует древовидную иерархию, в ядре сохранится его подчиненное отношение к родителю. Каждое ядро в процессе выполнения может как породить любое количество других ядер, так и содержать только вычислительные методы.*

Определение 3.1.2. *Ядро-потомок — порожденное ядро, содержащее независимую (параллельную) часть алгоритма. Такие ядра жестко связаны с породившими их узлами, они обязаны сообщать родителю о своем завершении.*

Определение 3.1.3. *Вычислительный сервер(сервер) — процесс, который распределяет ядра по локальным процессам. Каждое перемещенное*

ядро задачи будет запущено в отдельном, для каждой задачи своем, процессе. Так как виртуальная общая память у каждого процесса своя, то сервер выполняет роль коммутатора между процессами. Он ждет поступления ядер от других серверов и сам может их отправлять, а также следит за корректной работой наблюдаемых процессов. Сервера также находятся в иерархии, что упрощает балансировку ядер между ними. Такжже, этот объект занимается распределением файлов журналов внутри кластера, чтобы обеспечить процесс восстановления после сбоя.

Определение 3.1.4. *Журнал операций — абстрактное представление всех записей состояний вычислительных ядер, расположенных в порядке их появления относительно иерархии дерева. В любой момент работы системы в штатном режиме физически не существует, а создается только в момент сбоя, когда необходимо запустить процесс восстановления. Формируется из записей журналов-на-поток. Журнал операций «распределен» среди серверов, в отличии от журнала-на-поток.*

Определение 3.1.5. *Журнал-на-поток — основа процесса восстановления. Представляет собой последовательно обновляемый файл на устройстве постоянной памяти, содержащий последовательные записи состояний вычислительных ядер, выполняемых потоком-обработчиком.*

Определение 3.1.6. *Поток-обработчик — «процесс», протекающий в рамках задачи на сервере, который обрабатывает ядра, получаемые из очереди ядер. Для каждой задачи на одном сервере количество поток-обработчиков фиксировано. Управление числом потоков является настройкой вычислительного сервера, целью которой является повышение производительности за счет лучшей утилизации ресурсов.*

Определение 3.1.7. *Управляющий задачи — представляет собой конструкцию, которая соединяет вместе все вышеперечисленные определе-*

ния. В минимальном виде представляет собой очередь, составленную из ядер одной задачи на одном севере, которая обрабатывается потоками-обработчиками. На разных серверах могут быть разные управляющие одной и той же задачи, но набор ядер у них будет непересекающимся.

Базовые определения, даже данные в порядке их появления, не дадут представления о модели человеку, который не работал ранее, например, с моделью актеров. Поэтому я буду рассматривать работу модели на простом примере. Так я смогу расширить каждое определение в рамках фактической задачи, что должно упростить общее восприятие модели.

3.1.1. Программа-пример

Пусть нам необходимо рассчитать сумму трех интегралов от достаточно сложных функций по формуле

$$\int_{b_{00}}^{b_{01}} f(x)dx + \int_{b_{10}}^{b_{11}} k(x)dx + \int_{b_{20}}^{b_{21}} g(x)dx$$

и сохранить результат в базу данных. Запускать программу мы будем в распределенной среде, значит для расчета этой задачи у нас в распоряжении целый кластер. Сделаем уточнение — пусть мы считаем множество случаев, в зависимости от условий интегрирования, и для нас прерывание расчетов, в случае временного отключения кластера, крайне нежелательно. Условием надежности посчитаем возможность восстановить вычисления интегралов в момент, максимально близкий к точке сбоя, в том числе, возможность восстановить процесс записи в базу данных без перерасчета. Такой возможностью библиотеки MPI/OpenMP по умолчанию не обладают, поэтому выберем, не глядя, библиотеку, основанную на модели вычислительных ядер.

3.2. Правила разбиения

Теперь у нас есть задача, есть ее алгоритм (интегралы) и есть средство программной реализации. Приступим к программной реализации. В первую очередь, нам необходимо преобразовать исходный алгоритм в набор связанных вычислительных ядер. Опишем принцип формирования вычислительного ядра, тем самым ответим на ранее поставленный первый вопрос. Ядра, на этапе проектирования программы, рассматриваются как абстрактные части исходного алгоритма, полученные в ходе его разбиение по *правилам разбиения*. Если объединить все ядра вместе, то полученная последовательность операций должна в точности повторить неразбитый алгоритм. Правила разбиения можно сформулировать следующим образом:

1. Внутри вычислительного ядра должен находится логически законченный участок исходного алгоритма. То есть, результат выполнения объединяемых операций должен выражаться в чем-то конкретном, например, в возвращаемом значении. Чем лучше мы сможем сгруппировать операции, тем меньшим по объему получится множество ядер и тем меньшими будут затраты на поддержание их жизнедеятельности. В общем случае, примером логически законченного участка алгоритма можно считать функцию — она обособлена, она может возвращать значение, она не меняет своего смысла при смене места вызова. Ядра же являются расширением функций и могут включать в себя как функцию, так и ее контекст.

2. Внутри вычислительного ядра все операции последовательны. Это скорее рекомендация, чем правило — библиотека не запрещает своего совместного использования с другими средствами реализации распределенных вычислений. Однако, так как модель ядер сама предостав-

ляет средство распределения задачи, использование внутри модели сторонних средств может привести к путанице и ошибкам, а также к отсутствию гарантированного восстановления выведенной из модели части алгоритма.

3. Параллельные участки алгоритма должны быть инкапсулированы внутри одноуровневой иерархии ядер. То есть, часть алгоритма, выполнение которой можно распределить, заключается в одном вычислительном ядре (ядре-«обработчике»), которое на вход принимает часть данных. Разделением данных на части и контролем их обработки, в свою очередь, занимается ядро-«распределитель», которое в процессе деления данных порождает ядра-«обработчики» и является их родителем. Так, степень параллелизма будет определяться количеством одновременно запущенных ядер-«обработчиков».

4. Каждый участок алгоритма обладает своим контекстом данных. В программирование это еще называется областью видимости переменных. Так, в вычислительных ядрах необходимо придерживаться локальной области видимости переменных и не допускать ситуации, когда переменная обрабатывается в ядре, не связанном отношением потомок-родитель с ядром, в котором эта переменная хранилась изначально.

5. Принцип локальной видимости переменных влечет также необходимость дублирования переменной в каждом ядре, где она подвергается изменению. Это нужно, потому что между ядрами не существует общей памяти, вместо этого у них есть связь с общим предком в иерархии.

Если исходный алгоритм разбит на вычислительные ядра в соответ-

ствии с этими правилами, тогда в любой момент вычислений состояние всей системы будет отражено в журнале операций, а значит может быть восстановлено.

Вернемся к нашей задаче расчета интегралов. Исходный алгоритм вычислений можно описать так:

1. Считываем границы интегрирования.
2. Вычисляем интегралы, используя любой вычислительный алгоритм, поддающийся распараллеливанию.
3. Распараллеливаем вычисления по данным — по границам интегрирования.
4. Суммируем полученные значения.
5. Записываем их в базу данных.

Тот же алгоритм, но в виде псевдокода:

Листинг 3.1. Исходная программа

```
1  START PROGRAM;
2  B[3][2] <- read_boundaries(ARGS[]);
3  DO PARALLEL num_proc;
4  for (step_int1 <- (B[1][1] + B[1][2]) / num_proc)
5          int1_sum := calc_int1(step_int1);
6  for (step_int2 <- (B[2][1] + B[2][2]) / num_proc)
7          int2_sum := calc_int2(step_int2);
8  for (step_int3 <- (B[3][1] + B[3][2]) / num_proc)
9          int1_sum := calc_int3(step_int3);
10 END DO;
11 summery := int1_sum + int2_sum + int3_sum;
12 store_value(summery) <- db_connection;
13 END PROGRAM;
```

3.2.1. Разбиение программы-примера

Для демонстрации правила разбиения опустим параллельную часть алгоритма, заменив ее вызовом трех последовательных функций. Тогда

первое ядро будет включать в себя функцию считывания пределов интегрирования, еще три ядра будут относится к вычислению трех интегралов, с входным параметром шага интегрирования. Запись суммы в базу данных также следует выделить в отдельное ядро. У каждого ядра, помимо функции поведения, аналогичной оной в модели актеров, будут также переменные контекста. У первого — это матрица пределов и итоговая сумма, в ядрах-«интегралах» это текущее значение и заданный шаг, в ядре коннекторе к БД будет соответствующая информация о подключении к базе данных и передаваемое значение. Переписанный псевдокод для случая последовательной программы:

Листинг 3.2. Последовательный случай

```
1  START PROGRAM;
2  kernel1 {
3          B[3][2];
4      function read_boundaries(B, ARGS[]);
5      summery;
6
7      add Kernel3 <- Kernel1.step {
8          value;
9          step;
10         function calc_int1(step);
11         ...
12     }
13     add Kernel4 <- Kernel1.step {
14         value;
15         step;
16         function calc_int2(step);
17         ...
18     }
19     add Kernel5 <- Kernel1.step {
20         value;
21         step;
22         function calc_int3(step);
23         ...
24     }
25     add Kernel2 <- Kernel1.summery {
26         db_connection;
27         summery;
28         function store_value(summery);
29         ...
```

```
30          }
31      }
32  END PROGRAM;
```

Но исходная задача содержит параллельные участки, очевидно, что последовательная реализация нас не устроит. Чтобы перейти от последовательного случая к параллельному, ответим на следующие два вопроса:

- Как описать взаимодействие между ядрами?
- Как с помощью ядер реализовать параллельный алгоритм?

3.3. Иерархия вычислительных ядер

Из определения вычислительного ядра следует, что помимо функции-поведения (или функции обработки), в нем содержится связь ядра с иерархией. Связь эта выражается в указании, какое ядро является подчиняющим текущее, или иначе является ядром-родителем. Кроме этой связи, никаких других элементов межъядерного взаимодействия не предусмотрено. Однако, одной этой связи достаточно, чтобы построить эффективный обмен сообщениями между процессами в задачах и реализовать параллельные участки алгоритма.

Что передается по этой связи, какие у нее есть направления? Если вспомнить модель актеров [7], в ней обмен сообщениями был реализован через адреса и подсистему почтовых ящиков. Адреса фактически хранились каждый актером в условной «книге адресов», при этом было несколько способов пополнения этой «книги», так что любой актер мог теоретически связать с любым другим. Отдельным объектом было передаваемое сообщение, которое могло породить множество новых актеров. Но такой вид отношений — через периодически передаваемые сообщения, имеет смысл только в случае, когда актеры постоянны. То есть, единожды появившись,

актер существует практически бесконечно, независимо от того, обрабатывает он что-то или нет.

В реальных приложениях такой подход малоприменим, потому что течение задачи меняется иногда очень быстро, а написание же универсальных обработчиков практически невозможно. При расчетах реальных алгоритмов, чтобы не терять в производительности, нужно соблюдать принцип локальности, т.е. всегда стараться обрабатывать данные тем процессом/-процессором, который наиболее близок к данным. При пересылке сообщения любому актеру, принцип локальности нарушается, к тому же заранее неизвестны временные потери на этапе передачи. Подход, основанный на принципе локальности, описывается моделью параллельных вычислений BSP (*bulk, synchronous, parallel*) [16]. В нем каждый процесс может выполнять вычисления только над локальными данными, процесс коммуникаций используется для получения/передачи удаленных (по расстоянию) данных, а для синхронизации используется барьер.

Руководствуясь BSP, вычислительным ядрам выгоднее обмениваться данными только на уровнях родитель-потомок. Так как внутри ядра, по определению, должен быть логически связный алгоритм с четким результатом работы («ответом»), то обмениваться сообщениями между потомком и родителем постоянно также бессмысленно — вместо этого, выгоднее осуществлять только две передачи:

- Первая передача осуществляется родителем в момент создания ядра-потомка, “сообщением” здесь является сам потомок.

- Вторая передача происходит, когда потомок выполнит задание, данное ему родителем, или иначе выполнит свой собственный алгоритм. Здесь сообщением будет потомок с сохраненным внутри себя результатом.

Так, отдельного класса сообщения в модели вычислительных ядер не существует, вместо этого в системе передаются сами ядра. Стоит уточнить, что в момент порождения ядра заранее неизвестно, где оно будет выполнено. Почему, я объясню, когда буду описывать внутренние устройство вычислительного ядра. Но пока что, чтобы не терять контекст, скажу, что ядро может быть выслано на менее загруженный вычислительный сервер управляющим задачей, если у него самого не хватает времени его быстро обработать. Однако за счет иерархии родитель-потомок, вероятность выполнения порожденного узла локально выше, а значит меньше времени будет тратиться на синхронизацию задачи среди всех серверов.

Выяснив процесс взаимодействия ядер, становится очевидным способ реализации параллельных алгоритмов. Так, точкой входа в параллельную часть алгоритма будет являться ядро-«распределитель». Оно порождает однотипные ядра-потомки — количеством равные степени параллелизма. В каждом таком ядре-потомке происходит обработка небольшой части задачи, которая содержалась в ядре-«распределителе». Родитель в такой схеме одновременно является барьером, в соответствии с BSP. Он ожидает ответов от своих потомков и только после получения всех ответов считает свою задачу выполненной. При этом, родителю не важно, выполняют ли потомки свою задачу быстро, локально или медленно и удаленно. После порождения потомков он освобождает ресурсы процессора и переходит в режим ожидания. При этом потомки сами могут иметь длинные, параллельные цепочки ядер-потомков и также будут ожидать их выполнения. В отличии от модели актеров, ядра не держаться за ресурс все время — как только они выполнят свою часть работы или передадут ее дальше по иерархии, они высвобождают вычислительный ресурс.

Вернемся к исходной задаче и теперь не будем делать упрощений. Снова разобьем этот алгоритм на вычислительные ядра по вышеописан-

ным правилам. Так, последовательный процесс считывания пределов интегрирования поместим в первое ядро. Далее, расчет каждого интеграла, представленный в виде функции от шага интегрирования, которая возвращает значение интеграла, вынесем в поддерево ядер, где есть однотипные и простые ядра-обработчики и ядро-“распределитель”, который делит задачу между обработчиками и следит, когда они справятся с заданием. Деление задачи, очевидно, будет происходить по переделам интегрирования с вниманием к степени параллелизма. Ядром-“распределителем” в нашем случае выгоднее назначит самое первое ядро, которое считывает пределы интегрирования. Оно же будет ждать выполнения суммирования, а после породит ядро, которое запишет в базу данных итоговый результат вычислений (*store_value(summery)*). В результате разбиения, мы получили:

1. Ядро, считывающее границы интегрирования (1). Оно также управляет порождением ядер, вычисляющих интегралы.

2. Ядро, записывающее результат в базу данных (2)

3. 3 множества ядер $(3, 4, 5)$, рассчитывающих интегралы, находящиеся в подчинении к ядру (1).

Перепишем теперь псевдокод, используя разбиение.

Листинг 3.3. Параллельный случай

```
1  START PROGRAM;
2  Kernel1 {
3          B[3][2];
4      function read_boundaries(B, ARGS[]);
5      summery;
6      Kernel3 {
7          value;
8          step;
9          function calc_int1(step);
10         ...
11     }
12     Kernel4 {
```

```
13          value;
14          step;
15          function calc_int2(step);
16          ...
17      }
18      Kernel5 {
19          value;
20          step;
21          function calc_int3(step);
22          ...
23      }
24      Kernel2 {
25          db_connection;
26          summery;
27          function store_value(summery);
28          ...
29      }
30      for (step <- num_proc){
31          add kernel3(step);
32          add kernel4(step);
33          add kernel5(step);
34      }
35      ...
36  }
37  END PROGRAM;
```

Глава 4

Имплементация вычислительных ядер

На данном этапе мы смогли разбить исходный алгоритм, но получившиеся ядра все еще абстрактные. Перед тем, как мы перейдем к их фактическому наполнению, необходимо дать формальное описание внутреннего устройства вычислительного ядра. Ранее мной уже были упомянуты некоторые компоненты вычислительного ядра, такие как связь с ядром-родителем, локальные переменные. Но для полного функционирования этого недостаточно, потому что не определены действия ядра. Действиями в модели ядер я называю функции, вызовы которых обеспечивают эволюцию всей системы. Всего таких функций три.

Первая функция — функция **act()**. Она обеспечивает рост иерархии, непосредственно в этой функции происходят вычисления, порождаются новые ядра-потомки, вызываются процедуры. Данная функция "толкает" задачу в сторону выполнения и при этом не забоится о внешних воздействиях. **act()** не универсальна, она лишь называется одинаково во всех ядрах. Тем не менее, ее фактическое наполнение зависит от назначения ядра. Так, внутри параллельных участков в **act()** будет находится одинаковая функция для всех ядер-потомков, в случае нашего примера это функция вычисления интеграла. Хотя модель не запрещает внутри параллельных участков использовать любое количество ядер с отличными функциями **act()**, главное, чтобы они были логически независимыми. Эта функция также ответственна за передачу первого, из двух возможных, сообщения.

Вторая функция — **react()**. Эта функция фиксирует рост иерархии и вызывается потомками у своих родителей, после того, как они справятся со своей частью задачи. После успешного вызова данной функции ядро-

потомок считается выполненным, а ядро родитель больше никогда к нему не обратиться. При этом, фиксируется выполнение не только ядра, которое непосредственно вызвало эту функцию, но и всех его потомков. Для иерархического дерева это означает, что все поддерево задачи такого ядра выполнено успешно и не требует для себя дополнительных операций. Основное назначение этой функции в полной мере раскроется, когда мы перейдем к механизмам восстановления вычислений.

Третья функция — **rollback()**. Это очень важная функция, хотя и ситуативная. Из названия понятно, что она связана с процессом восстановления. Собственно, она вызывается только в случае нештатного завершения работы вычислительного сервера (серверов), и никогда при штатной работе. Эта функция содержит в себе логику, которая очищает систему от последствий выполнения ядра, в случае его некорректного завершения. Как и все предыдущие функции, она является уникальной для каждого ядра. Хотя она может быть и пустой, если внутри функции **act()** ядра выполняются идемпотентные операции. Идемпотентные операции — это такие операции, выполнение которых возможно сколько угодно раз без вреда для состояния системы в целом, например, таковой будет операция сложения двух констант. Реализуя логику восстановления корректного состояния, **rollback()** всегда вызывается для ядер, подвергшихся влиянию сбоя, в первую очередь.

Совокупность этих трех функций формирует абстрактную функцию поведения ядра, которая определяет его жизненный цикл, проиллюстрированный на (рис). Также в каждом ядре присутствуют служебные поля: идентификатор, ревизия. Они требуются управляющему заданием, чтобы организовать корректное выполнение задачи в распределенной среде.

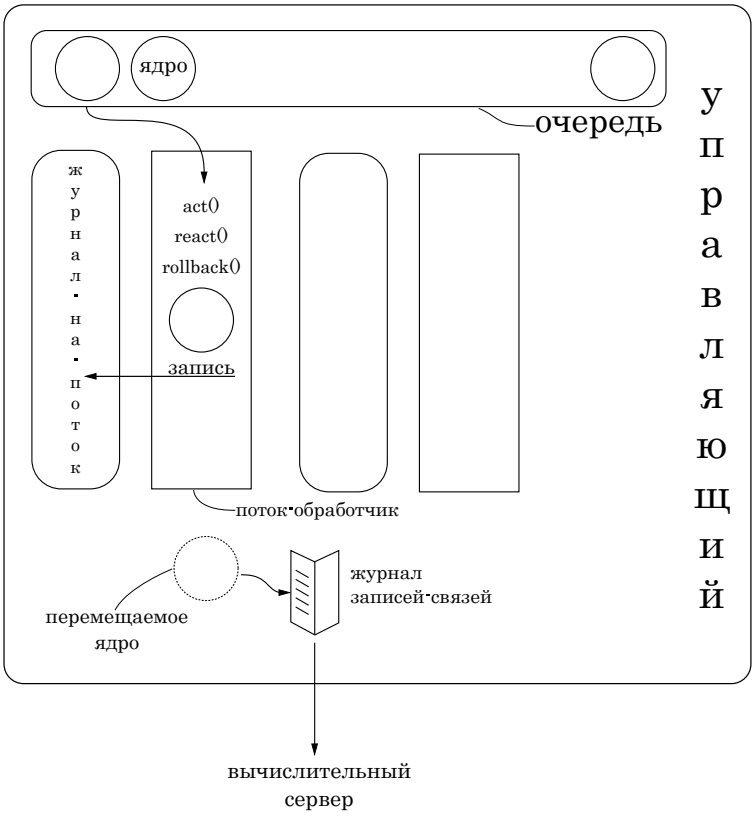

Рис. 4.1. Жизненный цикл вычислительных ядер

4.1. Ведение журнала

Перейдем к рассмотрению основы для восстановительного процесса — ведению журнала. Ранее мы определили критерии выбора минимального объекта записи, и, очевидно, что вычислительное ядро им удовлетворяет. Каждое ядро содержит небольшую и логически законченную часть алгоритма, состояние ядра отследить и сохранить легче, чем всей системы. Сохранение состояния может происходит достаточно часто, но не менее чем 3 раз за время жизни ядра:

1. В момент порождения ядра

2. После выполнения функции **act()** или, иначе, после выполнения своей части задания или порождения новых ядер-потомков, которым эта часть была передана.

3. После выполнения функции **react()**, или, иначе, после подтверждения успешного завершения всех порожденных ядер и их поддеревьев.

Дополнительно, для более эффективного поиска незавершенных ядер, состояние записывается:

- У ядра-потомка перед началом выполнения функции **react()** родителя.
- У ядра-родителя после каждого вызова функции **react()** потомками, так как в этот момент фиксируется выполнение подзадачи ядром-потомком. В случае потери результата выполнения подзадачу ядра придется пересчитывать, а система нацелена всеми силами избегать такой сценарий.

С точки зрения хранения информации для восстановления, в ядре нам интересны прежде всего его локальные переменные. К сожалению, ограничиться только значениями этих переменных для составления записи нельзя. Необходимо также каждый раз записывать вспомогательную информацию — идентификатор ядра, его ревизию, связь с его ядром-родителем. Такой набор полей для записи позволит восстановить иерархию ядер, а значит ход выполнения задачи. Также, в запись можно включить временную отметку, если того требует реализуемый алгоритм, но для процесса восстановления эта информация бесполезна, так как корректная временная последовательность появления ядер обеспечивается иерархией, а не фактически записанным в них временем.

Модель ядер является распределенной системой, а значит общего на все вычислительные сервера журнала физически не может существовать. Так, запись в журнал должна быть строго последовательной, объект журнала должен обладать блокировками, чтобы в любой момент времени у

него был только один владелец. Если сделать журнал общим даже в рамках потоков, то общее падение производительности будет столь велико, что подобной системой не будет смысла пользоваться. Поэтому ранее я ввел понятие журнала-на-поток. В рамках потока, или минимальной вычислительной единицы системы, в каждый момент времени может происходить только одна операция. Значит, в журнал, использование которого заблокировано только одним потоком, одновременно можно записать состояние только одного ядра, при этом, другие потоки не смогут получить до такого журнала доступ. В итоге, в системе одновременно ведется множество небольших журналов и невозможно заранее предсказать, каким будет распределение записей одного ядра среди этого множества.

Но, чтобы восстановить вычисление задачи, нам необходимо знать ее состояние на момент, предшествующий сбою. Так, нам надо знать все последние состояния ядер задачи, но без просмотра всего списка состояний, нельзя сказать, что текущее считанное — последнее. Так или иначе, в случае сбоя системы, необходимо прочитать все *журналы-на-поток* тех серверов, которые были подвержены сбою.

Это дорогая операция, но нужно учитывать, что:

1. Процесс восстановления всей системы — крайне редкое явление

2. На каждом вычислительном узле ведется ротация журналов, что означает, что журналы очищаются от записей состояний ядер выполненных задач достаточно часто, чтобы в момент чтения таких журналов, количество записей о невыполненных, или частично выполненных ядер было больше, чем о выполненных.

Вернемся к задаче об интегралах. Обладая полной информацией о модели ядер, теперь мы можем составить программу на псевдокоде целиком.

Листинг 4.1. Полный вариант программы по модели вычислительных ядер

```
START PROGRAM;
kernel1 {
    B[3][2];
    summery;
        _id;
        _rev;
        act() {
            B <- read_boundaries(B, ARGS[]);
            for (step <- B / num_proc){
                    add kernel3 (step) {
                            value;
                step;
                act() {
                    value <- calc_int1(step);
                }
                react {}
                rollback{}
            }
            add kernel4 (step) {
                            value;
                step;
                act() {
                    value <- calc_int2(step);
                }
                react {}
                rollback{}
            }
            add kernel5 (step) {
                            value;
                step;
                act() {
                    value <- calc_int3(step);
                }
                react {}
                rollback{}
            }
        }
    }
    react(child){
        summery += child -> value;
        wait(childs.not_done)
    }
    rollback{}
}
END PROGRAM;
```

4.2. Вычислительный сервер, управляющий задачи

Чтобы завершить обзор системы ядер, осталось рассмотреть два последних компонента. Первый – это управляющий задачи. Ядра не могут запускать методы внутри себя сами, им нужно рабочее окружение. Управляющий является таким окружением. При этом, это не отдельный процесс, в нем происходит прямая обработка вычислительных ядер. По своей сути, управляющий является процессом с внутренней очередью, в которой находятся ядра, принадлежащие одной задаче. Для работы с этой очередью у него есть неблокируемые функции взятия (*take*) и пополнения (*send*). Основной поток управляющего работает только с самой очередью (send), а для обработки ядер в ней создаются потоки-обработчики. Потоки-обработчики формально относятся к управляющему, но фактически обрабатывают задачи (*take*) асинхронно и не ожидают выполнения основного потока управляющего.

Так как модель является распределенной, то стоит описать процесс распределения задачи более чем на 1 узел. Это, частично, тоже обязанность управляющего задачи. Так как он оперирует ядрами, которые друг от друга формально не зависит и связаны только иерархией, управляющий может их перемещать. Так, он может передать вычислительному серверу сообщение с перемещаемым ядром, а тот, в свою очередь, решит, где его запустить. Дальнейшая судьба переданного ядра управляющего не интересует, однако он составляет специальную запись, состоящую из идентификаторов отправляемого ядра и его ядра-родителя, чтобы после получения ответа, вызвать функцию **react()** у нужного объекта, потратив на его поиск наименьшее количество времени.

Последний компонент системы — вычислительный сервер. Он контролирует появление и уничтожение управляющих на каждом узле кластера.

Он также осуществляет передачу ядер между узлами. Так, если на сервер приходит ядро задачи, которая не имеет еще “местного” управляющего, он будет создан. Также, сервер сам решает, куда именно отправить ядро, которое пришло от управляющего. Он может как переслать его на другой сервер, так и создать еще одну копию управляющего внутри себя. Поведение зависит от текущей загрузки сервера и выборке количества потоков-обработчиков относительно максимально установленного значения.

4.3. Работа модели

Теперь, рассмотрев в отдельности составные части модели вычислительных ядер, перейдем к описанию режимов работы программ, реализуемых в рамках данной модели. Далее будут представлены 2 режима работы: штатный и нештатный. Нештатный режим предполагает восстановление системы после какого-либо сбоя. В качестве примера, рассмотрим режимы работы программы вычисления интегралов.

Сперва опишем окружение выполнения программы. Пусть у нас есть кластер из N узлов, на каждом из которых есть процесс вычислительного сервера. Также имеется “точка входа”, через которую программа может быть запущена на этом кластере.

Штатный режим работы программы представляет собой последовательный относительно иерархии вызов функции **act()** у каждого ядра. Так, после запуска программы, сначала будет вызван act() у первого ядра — kernel1. В этот момент, будет порождено 3N новых ядер-потомков для расчета интегралов. Все они последовательно добавляются в очередь ядер, контролируемую управляющим задачи. Если скорость выполнения будет мала, то, начиная с некоторого места, управляющий начнет передавать поступающие ему новые ядра вычислительному серверу. Сервер в тот

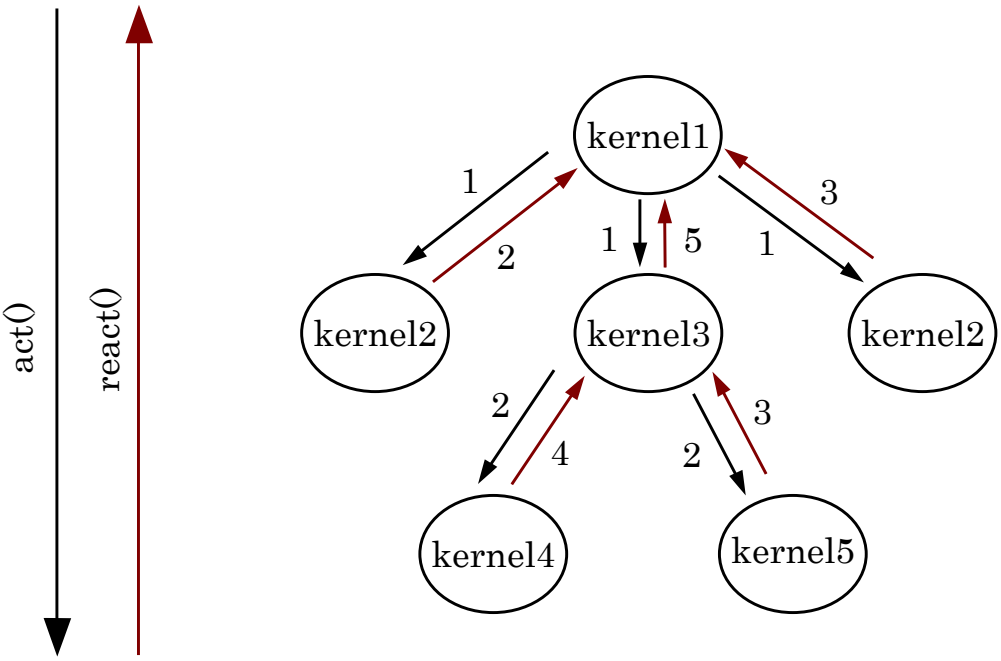

Рис. 4.2. Иерархия вычислительных ядер. Здесь цифрами обозначена очередность выполнения операции

момент решает, хватит ли ему потоков-обработчиков, чтобы создать еще одного управляющего. Если количество доступных потоков недостаточно, тогда переданные ему ядра отправляются “веером” другим вычислительным серверам. Асинхронно выполняется команда **react()** у ядер, справившихся со своей частью работы. Если ядро-потомок расположено на отличном от ядра-родителя вычислительном сервере, его ответ (то есть оно само, в соответствии с моделью) пересылается на нужный сервер. Там управляющий, отправивший ранее это ядро во вне, находит ядро-родитель в соответствии с записью-связью. В какой-то момент времени активное порождение новых ядер закончиться и программе останется дождаться всех вызовов **react()**. Как только будут удовлетворены внутренние условия завершения выполнения ядер, в нашем случае таким условием будет корректный расчет всех 3N ядер с частями интегралов с последующей подтвержденной записью в базу данных (успешный **react()** kernel2), программа считается исполненной. Самое первое ядро (kernel1) возвращает результат работы точке входа

программ.

Отвечая на последний поставленный вопрос о модели ядер, опишем нештатный режим работы программы. Возможно несколько вариантов сбоя:

1. Произошла программная ошибка при обработке какого-то ядра в потоке-обработчике
2. Произошел системный сбой в рамках работы одного управляющего
3. Вышло из строя 1 или несколько вычислительных серверов.
4. Весь кластер вышел из строя.

В первом случае восстановления вычислений не происходит. Генерируется сообщение об ошибке, а все оставшиеся ядра такой программы останавливаются. У таких ядер вызывается функция **rollback()**, чтобы очистить систему от влияния операций, которые могли потенциально успеть выполнится. За процесс остановки задачи отвечает вычислительный сервер, на котором произошел сбой. В случае программы расчета интегралов, непустой функцией **rollback()** будет у ядра, записывающего в БД — kernel2. Здесь стоит описать последовательность операций, которая удалит данные, достоверность которых поставлена под сомнение.

Если сбой произошел в одном из управляющих, например, из-за ошибки в подсистеме памяти, вычислительный сервер восстановит только ядра этого управляющего. Так, он считывает все журналы-на-поток остановленного управляющего и ищет в нем все ядра, чье состояние не фиксирует вызов функции **react()**.

Подробнее про процесс чтения журналов и восстановления ядер. Записи в журнале есть состояния ядер, которые при чтении преобразуются в фактические ядра. Так, прочитав все журналы одного управляющего мы

получаем множество ядер, которые соответствуют всем процессам, которые успел выполнить управляющий. Далее, нам нужно найти последнее состояние для всех ядер во множестве. Если состояние ядра фиксирует вызов функции **react()**, то такое ядро не требует дополнительных действий. Если состояние ядра показывает, что оно было только добавлено в очередь управляющего, тогда оно просто будет заново добавлено в очередь обработки. Если же ядро успело начать выполнять функцию **act()** до сбоя и не завершило ее, тогда сначала вызовется функция очистки состояния **rollback()**, а потом ядро добавится в очередь. После обработки всех последних состояний, в очереди на выполнение управляющего будет ровно столько ядер, сколько требуется для восстановления вычислений.

Последние два случая сбоев отличаются от предыдущих процессом добавления восстановленных ядер в очередь. Так, в случае частичного отказа кластера, задачи с отключенных узлов будут запущены заново на оставшихся, распределившись равномерно. Восстановить все ядра возможно, так как копии файлов журналов распределяются по всему кластеру. Однако, возможна ситуация, когда все журналы, включающие какую-то конкретную задачу, окажутся одновременно недоступными. Этот случай схож с тем, когда все узлы кластера отключаются. Так, после включения, вычислительный сервер сначала восстанавливает свой локальный журнал и восстанавливает управляющих. Некоторые ядра, правда, после такого восстановления могут не найти своих родителей, так как узел, который их передал, может оказаться недоступным. Тогда такое ядро ожидает восстановления внешних узлов некоторое, устанавливаемое время, и, в случае провала восстановления, вызывает свою функцию **rollback()**. Оставшаяся часть задачи, когда она станет доступна для восстановления, такие узлы создаст заново.

4.4. Тестирование

В качестве проверки работоспособности модели, реализуем в ядрах алгоритм рассчета некоторой задачи. Для примера, будем считать сумму ряда

$$\sum_{0}^{500000} tan(x+10)sin(x-10) + 1.34p,$$

где p некоторый коэффициент, расчитаный как

$$\sum_{0}^{5000000} 0.345sin(x) + cos(10x).$$

Полученная программа содержит 10 вычислительных ядер, каждое из которых считает ряды с шагом $p = p^{step}$. Ядро, в котором первоначально рассчитывается коэффициент p, будет являться корнем дерева ядер —родителем для всех счетных ядер. Будем теперь последовательно уменьшать колчество успешно выполненных ядер и замерять время работы процесса восстановления. О результате работы программы, можно сделать следующие выводы:

1. Процесс восстановления работает.
2. В среднем, время работы программы без восстановления и с ним отличается на 2 5 секунд.
3. По результатам тестирования, полученную модель можно считать удовлетворяющей требованиям, определенным в разеде «постановка задачи»

Так, верхняя область графика 4.3 отражает общее время работы программы, в случае восстановления 10 - указанное число оставшихся шагов. Нижняя область соответствует времени выполнения программы до сбоя. Пунктиром обозначено время работы программы без сбоев.

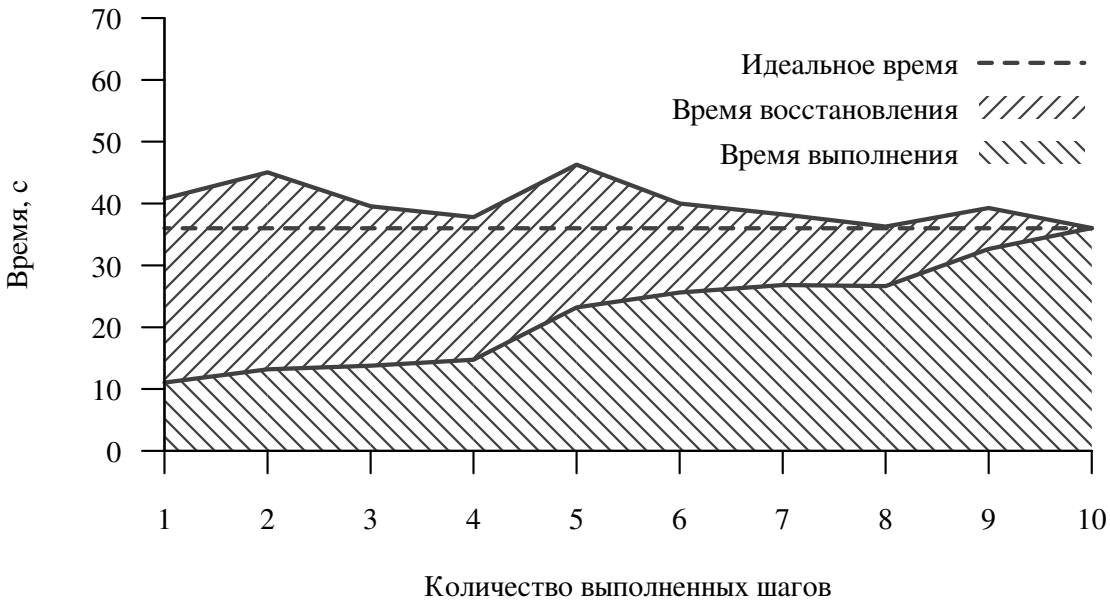

Рис. 4.3. Производительность алгоритма восстановления при различном количестве успешно выполненных вычислительных ядер по сравнению с производительностью без ошибок.

Выводы

Полученный в ходе исследования алгоритм запуска задач в полной мере решает проблему восстановления вычислений в кластерной среде после сбоя. Проведенное тестирование показывает, что программы, разработанные по модели актеров:

1. Восстанавливают свою работу после сбоя без полного перезапуска.
2. Могут быть как последовательными, так и параллельными.
3. Требуют для реализации переосмысления своего исходного алгоритма в логике вычислительных ядер.

Нельзя сказать, что полученная модель универсальна. Так, уже существующие программы не получится запустить в подобной системе. Они требуют такой переделки, которая переведет множество операций программы в множество вычислительных ядер, объединенных в иерархию.

Применимость модели — крайне широкая. Так как она, по сути, предназначена не для решение какой-то одной задачи, но является инструментом решения, то модель можно использовать во множестве алгоритмов и программ, вычисление которых требует суперкомпьютера. При том, вычислительные ядра можно использовать не только в расчетных задачах. “Индустриальные” задачи, требующие механизмов фиксации состояния, также могут быть решены в рамках полученной модели. Например, вызовы удаленных процедур или веб-сервисы, реализованные в логике ядер, могут рассматриваться как операции в реляционных базах данных, потому что могут реализовать механизм транзакций.

Дальнейшую работу, по моему мнению, нужно вести в двух направления. Первое заключается в исследовании алгоритмов распределения фай-

лов журнала по узлам кластера. Целью исследования должно стать нахождение или разработка наиболее эффективного способа. Часть работы модели, касающаяся распределение файлов, в данной работе не освещена, а только упоминается, однако она может вызывать значительные временные потери при выполнении программ.

Второе направление — программная реализация библиотеки или функционального языка программирования, полностью основанном на модели вычислительных ядер. Так, используя такой язык для реализации алгоритма, программист должен быть уверен, что итоговая программа будет параллельной, эффективной и без дополнительных изменений способной работать на суперкомпьютере. В данной работе реализован только прототип библиотеки, чтобы продемонстрировать идею.

Заключение

В ходе выполнения квалификационной работы были получены следующие результаты:

1. Разработан новый алгоритм запуска вычислительных задач в кластерных средах с восстановлением, на основе алгоритма без восстановления.

2. Пошагово, на примере, описан процесс использования нового алгоритма для получения программной реализации.

3. Оценена эффективность работы алгоритма в соответствии с целевым критерием.

4. Проведена верификация полученного алгоритма на предмет восстановления прерванных вычислений.

5. Разработана программная среда выполнения алгоритма запуска задач.

Список литературы

1. Bala A., Chana I. Fault tolerance-challenges, techniques and implementation in cloud computing // IJCSI International Journal of Computer Science Issues. 2012. Vol. 9, no. 1. P. 1694–0814.
2. Sindrilaru E., Costan A., Cristea V. Fault tolerance and recovery in grid workflow management systems // Complex, Intelligent and Software Intensive Systems (CISIS), 2010 International Conference on / IEEE. 2010. P. 475–480.
3. Nayeem G. M., Alam M. J. Analysis of Different Software Fault Tolerance Techniques. 2006.
4. Armbrust M., Fox A., Griffith R. et al. A view of cloud computing // Communications of the ACM. 2010. Vol. 53, no. 4. P. 50–58.
5. Meyer H., Rexachs D., Luque E. RADIC: A FaultTolerant Middleware with Automatic Management of Spare Nodes* // Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) / The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). 2012. P. 1.
6. Agha G. A. Actors: A model of concurrent computation in distributed systems.: Tech. rep.: DTIC Document, 1985.
7. Baker H., Hewitt C. Laws for communicating parallel processes. 1977.
8. Tasharofi S., Dinges P., Johnson R. E. Why do scala developers mix the actor model with other concurrency models? // ECOOP 2013–Object-Oriented Programming. Springer, 2013. P. 302–326.
9. Haller P. On the integration of the actor model in mainstream technologies: the scala perspective // Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentral-

ized control abstractions / ACM. 2012. P. 1–6.

10. Karmani R. K., Shali A., Agha G. Actor frameworks for the JVM platform: a comparative analysis // Proceedings of the 7th International Conference on Principles and Practice of Programming in Java / ACM. 2009. P. 11–20.
11. Gankevich I., Tipikin Y., Gaiduchok V. Subordination: Cluster management without distributed consensus // International Conference on High Performance Computing & Simulation (HPCS) / IEEE. 2015. P. 639–642.
12. Hargrove P. H., Duell J. C. Berkeley lab checkpoint/restart (blcr) for linux clusters // Journal of Physics: Conference Series / IOP Publishing. Vol. 46. 2006. P. 494.
13. Haller P., Odersky M. Scala actors: Unifying thread-based and event-based programming // Theoretical Computer Science. 2009. Vol. 410, no. 2. P. 202–220.
14. Orbit. https://github.com/orbit/orbit.
15. Armstrong J. Making reliable distributed systems in the presence of software errors: Ph. D. thesis / The Royal Institute of Technology Stockholm, Sweden. 2003.
16. Bisseling R. H., McColl W. F. Scientific computing on bulk synchronous parallel architectures. 2001.