Московский Государственный Университет
имени М.В. Ломоносова

Механико-математический факультет

Кафедра вычислительной математики

Создание программных средств обеспечения отказоустойчивости параллельных приложений для вычислительных систем с независимой дисковой памятью.

Дипломная работа

Выполнил: Шрамов П.А.
Научные руководители: проф. Васенин В.А., м.н.с. Инюхин А.В.

Москва
2005

Содержание

Введение

В последние годы использование вычислительных кластеров значительно расширилось, при этом возникли проблемы, ранее крайне редкие. Абсолютно надёжных аппаратных и программных средств не существует, поэтому в распределённых системах, объединяющих в единую вычислительную или информационную среду множество узлов, крайне остро встаёт вопрос о сбоях. В данной работе рассмотрены подходы к обеспечению отказоустойчивости системы с независимой дисковой памятью. Такого рода структуры в текущий момент являются наиболее распространёнными ввиду простоты создания и масштабируемости. В работе рассмотрены некоторые механизмы сохранения данных в случае выхода из строя узлов подобной системы. Подобные схемы обеспечивают восстановление как статических дисковых данных, так и работу с контрольными точками.

В связи с тем, что рост объёмов дискового пространства значительно опережает рост пропускной способности каналов связи, актуальной становится задача эффективного использования локальной дисковой памяти вычислительных узлов. Задачей было исследование различных алгоритмов хранения информации на локальной дисковой памяти и апробация наиболее интересных из них в реальных условиях. С этой целью был создан тестовый макет предполагаемой системы управления данными и реализованы основные из рассмотренных в работе механизмов.

1 Контрольные точки

Этот механизм, известный в технической литературе как «distributed checkpointing», позволяет уменьшить потери при выходе из строя вычислительного узла — процесс регулярно сохраняет свое состояние в виде некоторого файла, по которому впоследствии может восстановиться. Для задач, состоящих из одного процесса, трудности по созданию и восстановлению практически 0 — если сохранена информация, полностью описывающая внутреннее состояние на определённый момент времени. Для распределенных задач все сложнее, требуется учитывать то, что сбой может произойти только на некоторых узлах, что узлы могут обладать разной вычислительной мощностью и множество других факторов, влияющих на скорость вычисления. Для таких задач необходимо получать «координированную контрольную точку» — набор файлов, по которым возможно восстановить актуальное состояние вычислительных процессов. Таким образом, не потребуется запускать задачу заново, а необходимо всего лишь произвести вычисления с момента последнего создания файлов контрольной точки. Однако сохраненное состояние процесса необходимо оградить от сбоев узлов, на которых эти данные хранятся. Различные алгоритмы обеспечения отказоустойчивости рассмотрены, в частности, в [1].

В случае синхронных алгоритмов все процессы распределенной задачи сохраняют свое состояние одновременно с остальными, таким образом, такой набор позволяет восстановить задачу. Однако у такого алгоритма есть ощутимый недостаток — задача вынуждена синхронизироваться, и, если требуемые вычислительные ресурсы для разных частей задачи разные, быстрые процессы будут дожидаться медленных. К достоинствам синхронных алгоритмов стоит отнести их простоту — не требуется протоколирования сообщений, что несколько повышает производительность задачи.

Для распределённых систем передачи сообщений, например MPI, возможно в промежутках между созданиями точек сохранять на узле переданные данные. Тогда количество вычислительных ресурсов, затрачиваемых на восстановление после сбоя, будет значительно уменьшено, так как потребуется провести вычисления только для вышедших из строя узлов. Те процессы, которые не были подвержены сбою, не будут потреблять вычислительных ресурсов до тех пор, пока восстановленные не обработают все сообщения, переданные им с момента создания контрольной точки. Этот механизм необходим для асинхронных и квазисинхронных алгоритмов рассмотренных ниже.

В случае же асинхронных алгоритмов создания контрольных точек процесс сохраняет свое текущее состояние, не увязывая это с остальными. При этом из полученных файлов в общем случае довольно сложно выбрать координированную контрольную точку — то есть набор локальных точек, по которым можно восстановить процесс целиком. Например, рассмотрим следующий случай: в задаче было два процесса, и после создания локальной точки один передавал другому некоторое сообщение, и только после этого другой сохранял своё состояние. Тогда полученным файлам невозможно восстановить задачу — никакие локальные точки не образуют координированный набор. Таким образом, несмотря на кажущуюся простоту, этот такого рода алгоритмы не имеют практической ценности — слишком сложно восстановление.

Квазисинхронные алгоритмы объединяют в себе свойства как асинхронных так и синхронных алгоритмов. Создание локальной контрольной точки может происходить в результате каких-то взаимодействий с другими узлами, такие точки называются вынужденными («forced»). Например, в одном из алгоритмов локальная точка создается в случае, если процесс принимает сообщение, уже передав какое-то свое (No-Recieve-After-Send Method, NRAS). В этом классе, как и в асинхронных алгоритмах, точки могут создаваться в разное время, и поиск координированного набора представляет определённую трудность. Различные квазисинхронные алгоритмы имеют свои условия на вынужденные локальные точки, однако это позволяет уменьшить количество «бесполезных» точек, то есть тех, которые не входят ни в один координированный набор. Подробнее различные алгоритмы создания вынужденных точек рассмотрены в [5]. Практически все асинхронные и квазисинхронные алгоритмы подразумевают протоколирование сообщений, переданных узлом, и повторную обработку пропущенных после сбоя.

Независимо от того, получен ли набор файлов координированной контрольной точки по синхронному или квазисинхронному алгоритму, его следует оградить от сбоев дисковых накопителей. Общее у них то, что число файлов совпадает с числом узлов, или может быть объединено в такие группы, в общем-то одинакового размера. Любой алгоритм создания избыточной информации по набору файлов характеризуется следующим.

- **Дополнительные затраты времени:** на сколько приложение потребует больше времени для работы по сравнению с запуском без контрольных точек. Сюда следует отнести и время, которое задача ждет отстающие процессы в синхронных алгоритмах. Это характеристика исключительно реализации создания локальных точек, вычисление избыточной информации ведется в фоновом режиме совершенно иными приложениями.

- **Задержка на создание:** полное время, которое потребуется для создания отказоустойчивой контрольной точки. В случае, если оно меньше, чем предполагаемый интервал создания новых точек, особого влияния на производительность системы не оказывается. Поскольку на данный момент вычислительные мощности не являются «узким» местом, эта характеристика зависит от скорости дисковой и сетевой подсистем и объемов передаваемых данных.

- **Время восстановления:** полное время, необходимое на восстановление процесса по координированному набору контрольных точек. Так как этот процесс крайне редок, скорость восстановления не играет особой роли. Отдельно стоит вопрос о миграции процессов, в данной работе не затронутый.

- **Затраченные дисковые ресурсы:** полный объем дисковой памяти, необходимой для создания одного координированного набора контрольных точек. Так как предыдущий набор не может быть удалён, пока не закончится создание следующего, в системе должно иметься место по крайней мере для двух наборов. Для квазисинхронных алгоритмов может потребоваться больше места, так как координированная точка собирается из разрозненных локальных. Также сюда входят дополнительные затраты на создание избыточной информации для восстановления после сбоя.

- **Отказоустойчивость:** количество одновременных сбоев, которые может перенести данный контрольный набор. В некоторых ситуациях разумно совмещать частую запись точек, выдерживающую сбой одного узла и вызывающую небольшое падение производительности, и редкую запись точек для восстановления после множественного сбоя.

- **Влияние на общие ресурсы:** в случае, если вычислительная среда не отдана под исключительное использование данной задаче, имеет значение тот факт, насколько система построения координированного набора контрольных точек снижает производительность всего кластера.

2 Обеспечение отказоустойчивости набора файлов

Для распределенных вычислительных систем с локальной дисковой памятью необходимо защитить локальные контрольные точки от сбоев дисковой подсистемы. В силу большого количества узлов вероятность сбоя возрастает, к тому же возможны сбои в сетевой среде передачи данных, которые отрежут узел с ценной информацией от остального кластера. Рассмотрим некоторые схемы обеспечения отказоустойчивости набора файлов более подробно. Ниже будут использованы следующие обозначения:

N — количество контрольных точек в системе.

N_{sp} — количество избыточных точек в системе.

N_{all} — общее количество узлов в системе, $N_{all} = N + N_{sp}$.

τ_r — время восстановления после сбоя. Одновременными сбоями считаются те, которые произошли на промежутке времени, меньшем, чем длительность процесса восстановления τ_r. В эту величину входит, в частности, время реакции системы на произошедший сбой.

Под стандартным алгоритмом ниже понимается схема, создающая только локальные контрольные точки, также известная как Course-Grained Job Swapping (CGJS). Для распределения контрольной информации по кластеру используются следующие алгоритмы.

CGJS Локальные точки. Это простейший алгоритм, не дающий никакой отказоустойчивости. В нём, на каждом узле в локальной дисковой памяти создаётся контрольная точка. Таким образом, в случае выхода из строя любого узла, информация о задаче, работавшей на нём, будет безвозвратно утеряна.

MIR Передача «соседним». В этом случае копия локальной контрольной точки передаётся на «соседний» узел. Данная модель сходна с RAID1 — зеркалированием. Соседний узел определяется заранее исходя из топологии коммуникационной сети. К плюсам данной схемы следует отнести высокую скорость работы и малые вычислительные затраты. В случае коммутируемых сетей, передача данных между соседними узлами может идти независимо, что значительно снижает время на создание контрольных точек. Однако данная модель требует дискового пространства вдвое больше требуемого для CGJS и обеспечивает восстановление только при одном сбое. В случае, если выйдут из строя «соседние» узлы, одна из локальных контрольных точек будет утеряна.

CFS Передача на файловый сервер. Подходит только для систем с выделенным файловым сервером, заведомо более надёжным. Механизм обеспечения надёжности файловых серверов тщательно проработан, но выходит за рамки данной работы. В данной модели все локальные контрольные точки передаются на хранение на сервер. Такая схема обеспечивает восстановление после произвольного количества сбоев вычислительных узлов. Занимаемое дисковое пространство равно требуемому для CGJS. Вероятность невосстановимого сбоя в данном случае нулевая, так как схема выдерживает сбои любого количества узлов, если сервер считается надежным, вопросы обеспечения его отказоустойчивости выходят за рамки данной работы, хотя во многом сходны с рассмотренными здесь.

XOR Хранение на сервере контрольной информации. В этой схеме, как и в CFS, требуется дополнительный узел, однако не обязательно более надёжный. На данном узле сохраняется побитовый XOR всех контрольных точек данного набора. Эта модель выдерживает один сбой любого узла, в частности того, на котором хранится избыточная информация. Требования к дисковому пространству аналогичны требованиям для CGJS и дополнительно необходимо наличие места для одной точки; при N рабочих узлах затраты всего в $\frac{N+1}{N}$ раз больше, чем у CGJS. В качестве преимуществ стоит отметить простоту и малые вычислительные затраты, скорость создания избыточной информации превосходит RSC в 3-5 раз.

RSC Хранение контрольной информации на N_{sp} узлах. Этот случай имеет множество сходств с XOR, однако для вычисления избыточной информации используется не формальный XOR контрольных точек, а кодирование Рида–Соломона. Для произвольного N_{sp}, такого что $N + N_{sp} < 2^k$ и некоторого k, определяемого из используемого алгоритма кодирования, можно построить схему, устойчивую к N_{sp} сбоям и требующую в $\frac{N+N_{sp}}{N}$ раз больше дискового пространства, чем CGJS. Однако у данного алгоритма есть существенный недостаток — довольно большая вычислительная сложность, хотя в данное время этот минус не играет значительной роли. Подробнее об использованном в данной схеме методе кодирования будет рассказано ниже. Однако стоит упомянуть, что схема XOR является частным случаем данной, взятой с $N_{sp} = 1$, и алгоритмом построения избыточной информации — операцией исключающего ИЛИ (XOR). В качестве расширения данной схемы следует упомянуть следующее. В случае использования для вычисления кодов Рида–Соломона матрицы с дополнительными свойствами (3.6) появляется возможность строить многоуровневые схемы с XOR. При этом, так как на первом шаге используется более эффективный, чем RSC, метод, возможно ускорение операции восстановления после одного сбоя.

CGJS-FS Использование стандартной схемы на общей файловой системе. В качестве альтернативы возможно использование стандартного алгоритма создания координированной системы контрольных точек, однако на общей кластерной файловой системе. Тогда задача защиты от сбоев снимается со схемы создания контрольных точек. Количество сбоев, которое выдержит данная система, и скорость её работы зависят исключительно от модели, использованной в общей кластерной файловой системе. В качестве положительного момента такой схемы стоит отметить её простоту и малые непосредственные вычислительные затраты. Конечно, аппарат файловой системы потребует некоторых ресурсов — дисковых, процессорных и сетевых — для обеспечения хранения и защиты этих данных от сбоя, но это уже косвенные издержки. В зависимости от модели файловой системы данная схема близка к CFS, XOR или RSC. Однако если в случае кластерной файловой системе на общем дисковом накопителе особой разницы с CFS нет, то в случае вычислительных кластеров с распределёнными локальными дисковыми накопителями сильно возрастает время восстановления по сравнению со схемами XOR и RSC.

2.1 Многоуровневые схемы

Для большинства распределённых систем одиночные сбои происходят гораздо чаще, чем многократные. Соответственно, использование схемы, устойчивой к k сбоям, может создать неоправданно высокие затраты. Многоуровневые схемы представляют из себя комбинацию простой модели типа MIR или XOR и сложной высоко-затратной типа CFS или RSC. Тогда наиболее вероятные одиночные сбои будут восстанавливаться за меньшее время, чем сбои множественные. В частности, в [4] показано, что двухуровневые модели в среднем дают меньшие дополнительные затраты, нежели схемы одноуровневые. Так, в случае простых схем, выдерживающих один сбой, после выхода из строя второго узла вычисления должны начаться заново, а сложные схемы дают слишком большую прибавку времени. Многоуровневые схемы характеризуются, в частности, соотношением количества N_1 и N_2 контрольных точек первого и второго уровней. Если N_1/N_2 слишком велико, время восстановления после двойного сбоя также растёт. Однако в случае, если это соотношение мало, значительными становятся дополнительные временные затраты по сравнением с системой без контрольных точек. Количество тяжёлых и лёгких точек, равно как и максимальное количество сбоев, выдерживаемых схемой, должно быть установлено исходя из имеющейся вычислительной системы и требований на её отказоустойчивость.

В качестве одного из частных, но перспективных случаев многоуровневых схем следует рассмотреть RSC_k. Она позволяет сделать многоуровневую схему в виде одной модели создания избыточной информации всего лишь модификацией параметров. Обозначим RSC_k схему RSC при $N_{sp} = k$. Тогда RSC_1 является «лёгкой» схемой с малым временем восстановления и устойчивой к одному сбою. Переход от RSC_k к RSC_{k+1} — это добавление одной новой точки избыточной информации без модификации уже имеющихся k. Это же верно для большего шага, то есть для перехода от RSC_k к $\mathsf{RSC}_{k+\kappa}$; отличие в том, что создаётся не одна дополнительная точка избыточной информации, а κ. Таким образом, создание RSC_k может быть разбито на несколько аэтапов, постепенно повышающих отказоустойчивость системы. Так, быстро созданная RSC_1 позволит отработать наиболее частый одиночный сбой, даже если тот произошёл в процессе создания «тяжёлой» точки. Данная модель может иметь как два, так и много уровней. Например, для $k = 3$ возможны два варианта:

2 уровня, при использовании схем RSC_1 и RSC_3,

3 уровня, при использовании схем RSC_1, RSC_2 и RSC_3.

2.2 Сводная таблица потребляемых ресурсов

В следующей таблице кратко описаны требуемые ресурсы для рассмотренных выше схем контрольных точек, как дисковые, так и сетевые. Кроме этого указана отказоустойчивость схем. Поскольку время создания и восстановления напрямую зависят от объема сетевых передач, этих пунктов тут нет.

Схема	Дисковые ресурсы	Сетевые ресурсы	Отказоустойчивость
CGJS	N локальных контрольных точек	не требует	не обеспечивает
MIR	CGJS + одна копия каждой на удалённом узле	одновременная передача N точек на разные узлы	устойчива к одиночному сбою, в некоторых случаях к нескольким
CFS	N контрольных точек на центральном сервере	передача всех точек на центральный сервер	любое количество сбоев, кроме серверного
XOR	CGJS + одна точка избыточной информации	передача объёма, равного N точкам, по частично независимым каналам	один сбой, как узла содержащего контрольную точку, так и избыточного
RSC	CGJS + N_{sp} точек избыточной информации	передача объёма, равного $N \cdot N_{sp}$ точкам, по частично независимым каналам	N_{sp} сбоев, как узлов, содержащих контрольные точки, так и избыточных

3 Коды Рида–Соломона

Коды Рида–Соломона являются частным случаем кодов, исправляющих ошибки. Данная область является хорошо исследованной, однако практическое применение по восстановления больших объёмов данных было весьма слабым. Это обусловлено сложностью реализации и относительно низкой скоростью работы алгоритмов кодирования. Их вычислительная сложность перестала быть сдерживающим фактором лишь в последние годы, когда компьютерные системы стали высокопроизводительными. В качестве основы для создания системы были взяты коды Рида–Соломона, освещённые в [2].

3.1 Общее описание

Пусть имеется n блоков данных, которые необходимо защитить от m сбоев. Сбоем считается потеря целого блока, например, выход из строя узла, на котором эти данные хранились. По блокам данных строится m блоков избыточной информации с использованием линейных кодов таким образом, что в случае потери до m любых элементов из $n+m$ имеющихся есть возможность восстановить исходные данные.

3.2 Формализация

Все вычисления ведутся над полем $GF(2^k)$ (в нашем случае $k = 8$). Для простоты изложения будем считать блоком данных число из k бит, в случае бóльших блоков описанную ниже операцию надо будет повторить для каждого k-битного слова. Обозначим вектор данных $\vec{D} = (D_1, D_2, \dots, D_n)^T$. Рассмотрим матрицу A размера $n \times (n+m)$ со следующим свойством: при удалении любых m строк она остаётся матрицей ранга n, а верхние n строк составляют единичную матрицу. Тогда формальное произведение $A \cdot \vec{D} = \vec{S}$ даст вектор $\vec{S} = (S_1, S_2, \dots, S_{n+m})$, такой что $\forall i \in \{1, \dots, n\}$ имеем $S_i = D_i$. Таким образом, $\vec{D}$ является решением системы $A \cdot \vec{D} = \vec{S}$ из $n + m$ линейных уравнений с n неизвестными. Пусть $\vec{S'}$ обозначает вектор, полученный из $\vec{S}$ вычёркиванием некоторого количества элементов, а A' — матрицу A с удалёнными соответствующими строками. Если количество потерь не более m, то система $A' \cdot \vec{D'} = \vec{S'}$ будет иметь единственное решение, равное исходному вектору $\vec{D}$.

3.3 Вычисления над $GF(2^k)$

Рассмотрим поле $GF(2^k)$ формальных многочленов над $\{0, 1\}$ степени не выше k. Характеристика этого поля равна 2, таким образом, сложение и вычитание представляются логической операцией исключающего ИЛИ (XOR). Умножение над конечными полями Галуа является трудоёмкой задачей, однако для небольших значений k строятся таблицы логарифмов, значительно ускоряющие вычисления. Для данного k выбирается образующая e, так что любой элемент мультипликативной группы данного поля представляется в виде e^p. Умножение $P_1 \cdot P_2 = e^{p_1} \cdot e^{p_2} = e^{p_1+p_2}$ сводится к вычислению логарифмов $p_1 = \log P_1$ и $p_2 = \log P_2$, сложению и возведению в степень. В случае малых k возможно создать таблицы умножения для некоторых часто используемых множителей. Для этого строится вектор $\vec{M}_\mu$, такой, что на i месте у него находится произведение $\mu \cdot i$ над полем $GF(2^k)$. Использование таких предвычисленных таблиц умножения может значительно ускорить работу алгоритма кодирования Рида–Соломона. Ниже использование таких таблиц будет описано несколько подробнее.

3.4 Построение матрицы A

Рассмотрим матрицу Вандермонда V размера $n \times (n+m)$, где $n + m < 2^k$, любой минор $n \times n$ которой имеет ранг n.

$$V = \begin{pmatrix} a_1^0 & a_1^1 & \cdots & a_1^n \\ a_2^0 & a_2^1 & \cdots & a_2^n \\ \vdots & & \ddots & \vdots \\ a_{n+m}^0 & a_{n+m}^1 & \cdots & a_{n+m}^n \end{pmatrix}$$

Элементарными (столбцовыми) матричными преобразованиями верхний $n \times n$ минор приводится к единичному виду. Соответственно изменяются и нижние m строк данной матрицы. Так как элементарные преобразования не меняют ранга, любой минор $n \times n$ имеет полный ранг. В случае, если кодирование используется для построения дополнительных точек избыточной информации, матрица строится несколько иным образом. Если к имеющимся m точкам требуется достроить ещё μ, должна быть взята матрица Вандермонда размера $n\times(n+m+\mu)$. После приведения строки с номерами с $n+1$ по $n+m$ надо выкинуть. Тогда останется матрица размера $n\times(n+\mu)$, по которой уже строятся μ точек избыточной информации. Таким образом, после объединения данных наборов точек m и μ получается согласованное множество, идентичное построенному за один шаг. Так как первые n строк составляют единичную матрицу, вычисления первых n элементов вектора $\vec{S}'$ производить не требуется. Для последних m строк можно построить предвычисленные таблицы умножения; учитывая специфику приложения данного метода, это не потребует больших затрат памяти. Так как элементы D_i расположены на разных узлах, каждый будет хранить только m предвычисленных таблиц.

3.5 Решение системы $A' \cdot \vec{D}' = \vec{S}'$

Данная система обладает следующей спецификой: её требуется решить множество раз с фиксированной матрицей A', но с разными правыми частями $\vec{S}'$. Таким образом, единственный эффективный способ — построение обратной матрицы ${A'}^{-1}$ и умножение $\vec{D}' = {A'}^{-1} \cdot \vec{S}'$. При этом обратную необходимо построить всего один раз. Количество строк, в которых матрица ${A'}^{-1}$ отлична от единичной, совпадает с количеством потерянных элементов, чьи индексы меньше n. Пусть это число равно m'. Тогда, как и в случае с созданием избыточной информации, в целях уменьшения вычислительных затрат возможно построение предвычисленных таблиц умножения, и на каждый узел сети будет приходиться лишь m' таких таблиц.

3.6 Матрица с единичной строкой

Пусть матрица A является результатом преобразования матрицы Вандермонда рассмотренным выше, а её строка с номером $n+1$ имеет вид $(f_{1,1}, \ldots, f_{1,n})$, причём $f_{1,i} \neq 0$, $1 \leqslant i \leqslant N$.

Определим A_m соотношением (1), где $\mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ — диагональная матрица с элементами поля λ_1, …, λ_n на диагонали.

$$A_m = (\mathrm{diag}(f_{1,1}, \ldots, f_{1,n}, 1, \ldots, 1))A(\mathrm{diag}(f_{1,1}^{-1}, \ldots, f_{1,n}^{-1})) \tag{1}$$

$$A_m = \begin{pmatrix} \mathrm{diag}(f_{1,1}, \ldots, f_{1,n}) \\ F \end{pmatrix} (\mathrm{diag}(f_{1,1}^{-1}, \ldots, f_{1,n}^{-1})) \tag{2}$$

$$A_m = \begin{pmatrix} I \\ F(\mathrm{diag}(f_{1,1}^{-1}, \ldots, f_{1,n}^{-1})) \end{pmatrix} = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \\ f_{1,1}f_{1,1}^{-1} & \cdots & f_{1,n}f_{1,n}^{-1} \\ \vdots & \ddots & \vdots \\ f_{m,1}f_{1,1}^{-1} & \cdots & f_{m,n}f_{1,n}^{-1} \end{pmatrix} \tag{3}$$

Получившаяся в результате матрица A_m также обладает свойством линейной независимости любого набора из n строк (умножения на невырожденные диагональные матрицы справа и слева не меняют этого свойства), $A_m \in Q$, а её $(n+1)$–я строка равна $(1, \ldots, 1)$.

4 Прикладные средства

4.1 Введение

В ходе работ был создан тестовый макет реализующий некоторые из рассмотренных выше алгоритмов. В частности, механизмы вычисления избыточной информации по набору файлов. Кроме этого был разработан механизм оповещения системы о создании контрольной точки. Ниже приводится описание прикладных средств, тестовые замеры производительности и краткая документация по пользовательскому интерфейсу.

4.2 Копирование файлов контрольных точек

Рассмотрим некоторые алгоритмы широковещательной рассылки по однородному вычислительному кластеру. Сетевая инфраструктура считается построенной так, что не имеет значения, с какого узла и на какой узел передавать — издержки буду одинаковыми. Ниже будет проанализирована производительность различных схем.

4.2.1 Постановка и моделирование

Требуется передать файл с исходного узла на n принимающих, причём размер файла может быть весьма велик и доходить до десятков гигабайт. Передача может вестись не только с исходного узла (*source*), возможна ретрансляция узлами-приёмниками. Таким образом, строится некоторый ориентированный граф, охватывающий все узлы-приёмники. Ниже будет представлено несколько различных типов передающих сетей. Общие для всех моделей обозначения:

N_{dest} — количество принимающих узлов.

B_{disk} — скорость чтения единицы данных из локальной дисковой памяти.

B_{net} — скорость чтения единицы данных по сетевому соединению.

B_{total} — скорость передачи данных в построенной коммуникационной сети. Можно взять равной средней скорости выхода данных с исходного узла по каждому соединению.

Для простейшей ситуации, когда $N_{dest} = 1$, скорость передачи можно считать равной $B_{total} = \min(B_{disk}, B_{net})$.

4.2.2 Передача от одного всем

Простейшая модель, когда исходный узел передаёт каждому принимающему непосредственно, без ретрансляций (рис. 1). Эта схема отличается простотой, однако для неоднородных коммуникационных сетей она не применима. Её отличает простота реализации, однако в случае, если $B_{net} < B_{disk} \cdot N_{dest}$, начнутся значительные потери производительности. Скорость B_{total} в такой ситуации можно считать равной $B_{total} = \min(B_{net}/N_{dest}, B_{disk})$.

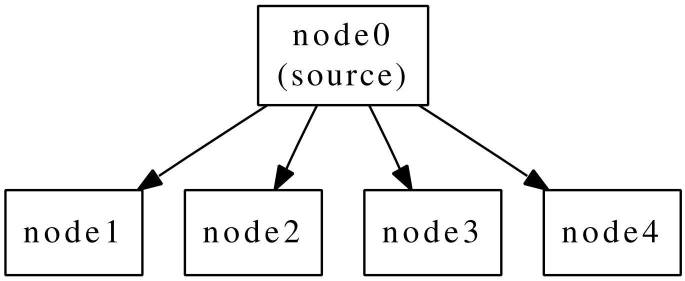

Рис. 1: Передача от одного всем. $N_{dest} = 4$

4.2.3 Последовательная передача

В этой модели блоки данных передаются по цепочке узлов (рис. 2). Каждый ретранслирующий узел записывает полученный блок на локальный диск и передаёт следующему в цепочке. Эта схема значительно разгружает сетевую инфраструктуру, так как ни один узел не имеет более одного входящего и одного исходящего потока. Таким образом, на сетях с коммутируемыми каналами данные передачи будут идти практически независимо. Кроме этого, в случае гетерогенной коммуникационной сети возможно так расположить узлы в последовательность, что по наиболее узким каналам, коллективно используемым группой узлов, произойдёт только одна передача. Время, затраченное на передачу блока, равно времени, затраченному на запись блока на диск и пересылку по сети. Таким образом, для линейной передачи пропускная способность $B_{total} = \frac{B_{net} \cdot B_{disk}}{B_{net} + B_{disk}}$.

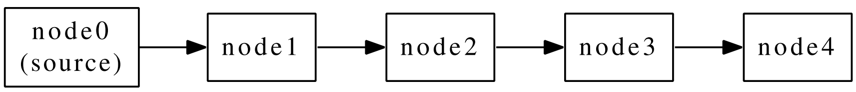

Рис. 2: Последовательная передача. $N_{dest} = 4$

4.2.4 Передача по двоичному дереву

Эта модель построена аналогично последовательной передаче, однако некоторые узлы дополнительно передают блоки данных двум другим (рис. 3). Занумеруем узлы каким-либо образом, присвоив исходному узлу индекс 0. Далее по данным индексам построим двоичное дерево с корнем в узле с нулевым индексом. Построение происходит по двоичной записи индекса — у $node_k$ источником является $node_{k_R}$, а передача идёт на $node_{k_L}$ и $node_{k_L|1}$, где k_R — побитовый сдвиг k вправо, k_L — влево, а $a|b$ — операция побитового ИЛИ. Таким образом, для любого N_{dest} мы построим дерево глубины не более $[\log N_{dest}] + 1$. Вычисление скорости передачи аналогично последовательному случаю $B_{total} = \frac{B_{net} \cdot B_{disk}}{B_{net} + B_{disk}}$. Данная схема может быть адаптирована под гетерогенные коммуникационные сети, если при нумерации узлов учитывать информацию о топологии вычислительной системы.

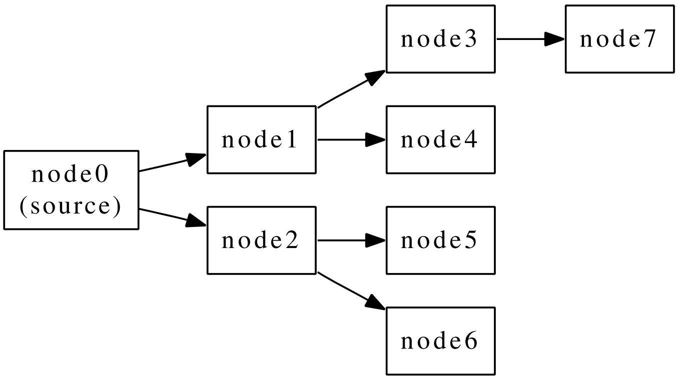

Рис. 3: Передача по двоичному дереву. $N_{dest} = 7$

4.2.5 Сравнительные результаты

В ходе тестов передача данных шла с использованием коммуникационной среды SCI с пропускной способностью около 240Mb/s и жёстких дисков со скоростью чтения и записи около 40Mb/s. Передача данных по сетевой среде производилась при помощи средств MPI, что объясняет некоторые странности в полученных результатах (при увеличении принимающих узлов с 1 до 2 скорость возрастает). Для модели передачи по дереву были достигнуты скорости $B_{total} = 34.5$Mb/s при $N_{dest} = 7$. При аналогичных условиях для последовательной пересылки и для передачи от одного всем были получены скорости $B_{total} = 28.5$Mb/s и $B_{total} = 19.5$Mb/s соответственно. На графике (рис. 4) представлены усреднённые результаты тестов на файлах размером от 2 до 16 Gb по трём рассмотренным выше моделям коммуникационных сетей.

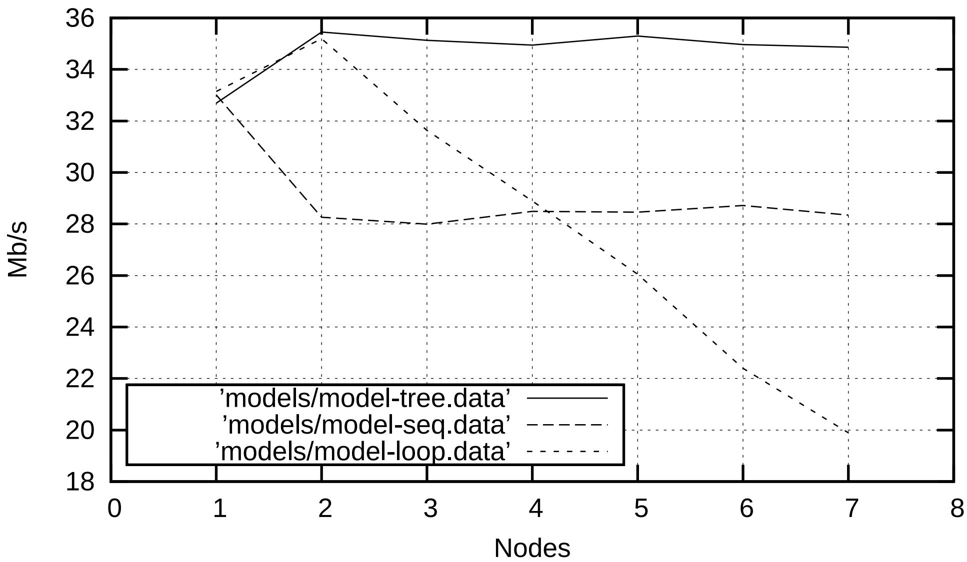

Рис. 4: Скорость передачи B_{total} в зависимости от количества узлов. $N_{dest} = 1 - 7$

4.3 Оптимизация рассылок в неоднородных средах

Рассмотренные выше модели распределения данных подходят только для однородных коммуникационных сред, так как не учитывают сложность передачи между компонентами связности метакластера. Ниже будет предложен простой алгоритм построения графа пересылок и доказательство его оптимальности для случая неоднородных коммуникационных сред.

4.3.1 Формализация задачи

Дан полный граф Γ с вершинами $n(\Gamma) = \{n_0, \ldots, n_N\}$, вершина n_0 и весовая функция на ребрах $f : n(\Gamma) \times n(\Gamma) \to \mathbb{R}$. Для ребра $e = (n_i, n_j)$ положим $f(e) = f(n_i, n_j)$. Далее f везде предполагается симметричной, то есть $f(x, y) = f(y, x)\, \forall x, y \in n(\Gamma)$.

Определение 1. Весом графа Γ' с ребрами $e(\Gamma') = \{e_1, \ldots, e_M\}$ называется число, равное $\mu(\Gamma') = \sum_{i=1}^{M} f(e_i)$.

Определение 2. Граф Γ' называется связным, если для любого $n_i \in n(\Gamma')$ существует путь по ребрам графа Γ' $e_1^i, \ldots, e_k^i$ из n_0 в n_i

Определение 3. Связный граф Γ' называется оптимальным, если его вес минимален среди всех связных графов Γ'' с теми же вершинами.

Требуется построить оптимальный граф Γ' с вершинами $n(\Gamma') = n(\Gamma) = \{n_0, \ldots, n_N\}$. Заметим, что оптимальных графов может быть много, например, в случае, если $f(n_i, n_j) = \mathrm{const}$, оптимальным будет любой связный граф.

4.3.2 Построение

Будем строить Γ' по индукции. База: $\Gamma'_0 = (\{n'_0 = n_0\}, \varnothing)$ — единственный граф из одной вершины, и поэтому оптимальный.

Шаг: пусть имеется оптимальный граф Γ'_k , построим Γ'_{k+1} с теми же свойствами. Пусть $n(\Gamma'_k) = \{n'_0 = n_0, n'_1, \ldots, n'_k\}$ — множество вершин Γ'_k, найдём вершины $n'_{k+1} \in n(\Gamma) \setminus n(\Gamma'_k)$ и $n'_j \in n(\Gamma'_k)$, такие что $\forall \hat{n}' \in n(\Gamma'_k)\, \forall \hat{n} \in n(\Gamma) \setminus n(\Gamma'_k)\ f(n'_j, n'_{k+1}) \leqslant f(\hat{n}', \hat{n})$. Другими словами, n'_{k+1} — вершина из $\Gamma \setminus \Gamma'_k$, такая что «расстояние» от неё до Γ'_k минимально среди всех $n(\Gamma) \setminus n(\Gamma'_k)$. Будем нумеровать ребра в порядке добавления в Γ'_{k+1}, то есть $\Gamma'_{m+1} = \Gamma'_m \cup \{e_m\} \cup \{n'_m\}$.

Утверждение. Γ'_{k+1} *оптимален.*

Доказательство. Рассмотрим произвольный связный граф без циклов $\Gamma'' = \Gamma''_{k+1}$ на вершинах $n(\Gamma'_{k+1})$. Покажем что $\mu(\Gamma'_{k+1}) \leqslant \mu(\Gamma'')$.

Определение 4. Положим характеристику $\chi_{n'_l}(\hat{\Gamma})$ равной количеству ребер в графе $\hat{\Gamma}$ имеющих одним из концов вершину n'_l.

Пусть $n'_{l_1}, n'_{l_2}, \ldots, n'_{l_q}$, $q = \chi_{n'_{k+1}}(\Gamma'')$ — вершины, соединённые с n'_{k+1} ребром в графе Γ'', выписанные в порядке добавления их в граф Γ'_{k+1}, то есть $\forall j \in \{1, \ldots, q - 1\}\ \exists i\ n'_{l_j} \in n(\Gamma'_i),\ n'_{l_{j+1}} \notin n(\Gamma'_i)$. Далее рассмотрим два случая.

$\chi_{n'_{k+1}}(\Gamma'') = 1$. Тогда $f(n'_{l_1}, n'_{k+1}) \geqslant f(n'_j, n'_{k+1})$, где n'_j — вершина, соединённая с n'_{k+1} ребром в графе Γ'_{k+1}, а граф Γ'_k оптимален по предположению индукции.

$\chi_{n'_{k+1}}(\Gamma'') > 1$. Построим граф $\hat{\Gamma}''$ с весом $\mu(\hat{\Gamma}'') \leqslant \mu(\Gamma'')$, такой что $\chi_{n'_{k+1}}(\hat{\Gamma}'') = \chi_{n'_{k+1}}(\Gamma'') - 1$ Положим $i_1 = l_q$ и $i_2 = l_{q-1}$. Рассмотрим ребро $\hat{e}_2 = (n'_{k+1}, n'_{i_2})$. Так как Γ'' не имеет циклов, после удаления $\hat{e}_2$ он распадётся на две компоненты связности. Обозначим Γ''_+ компоненту, содержащую n'_{i_2}, и Γ''_- — оставшуюся часть.

Вспомогательное утверждение 1. Для любой вершины $n'_l \in n(\Gamma'_{k+1})$, ребра $e_m \in e(\Gamma'_{k+1})$, выходящего из неё, и любого ребра $e_i \in e(\Gamma'_{k+1})$, $l < i \leqslant m$, верно, что $f(e_i) \leqslant f(e_m)$.

Вспомогательное утверждение 2. Для любой вершины $n'_l \in n(\Gamma'_{k+1})$, ребра $e \notin e(\Gamma'_{k+1})$ или $e = e_{k+1}$, выходящего из неё, и любого ребра $e_i \in e(\Gamma'_{k+1})$, $l < i \leqslant k+1$, верно, что $f(e_i) \leqslant f(e)$.

Определение 5. Назовем ребро $e = (n_i, n_k)$ *соединяющим*, если $e \in e(\Gamma'_k)$ и $n_i \in n(\Gamma''_+)$, $n_j \in n(\Gamma''_-)$ или $n_j \in n(\Gamma''_+)$, $n_i \in n(\Gamma''_-)$.

Предположим что найдётся, соединяющее ребро $e_{j'_2}$, такое что $i_2 < j'_2 \leqslant i_1$; тогда $f(e_{j'_2}) \leqslant f(e)$ по вспомогательному утверждению 2. Рассмотрим граф $\hat{\Gamma}'' = e_{j'_2} \cup \Gamma'' \setminus e$; для него выполнено $\mu(\hat{\Gamma}'') \leqslant \mu(\Gamma'')$ и $\chi_{n'_{k+1}}(\hat{\Gamma}'') = \chi_{n'_{k+1}}(\Gamma'') - 1$, таким образом построен граф с искомыми свойствами. Если же такого $e_{j'_2}$ нет, то рассмотрим вершину $n'_{i_3} \in \Gamma'_-$, такую что из неё выходит ребро e_{j_3}, $j_3 > i_2$. Тогда $f(e_{j_3}) \leqslant f(e)$ по вспомогательному утверждению 1. Если найдётся соединяющее ребро $e_{j'_3}$, такое что $i_3 < j'_3 \leqslant i_2$, то заменой e на $e_{j'_3}$ получаем граф $\hat{\Gamma}''$ с искомыми свойствами, иначе — найдём $n'_{i_4} \in \Gamma''_+$ и проведём с ней аналогичные рассуждения. В результате получится цепочка неравенств $f(e) \geqslant f(e_{j_3}) \geqslant f(e_{j_4}) \geqslant \dots$. Поскольку j_m ограничены снизу, и существует хотя бы одно соединяющее ребро, таковое будет найдено, и заменой e на него будет построен искомый граф $\hat{\Gamma}''$, такой что $\mu(\hat{\Gamma}'') \leqslant \mu(\Gamma'')$ и $\chi_{n'_{k+1}}(\hat{\Gamma}'') = \chi_{n'_{k+1}}(\Gamma'') - 1$.

Таким образом, цепочкой переходов $\Gamma'' \to \hat{\Gamma}'' \to \dots$ можно придти к случаю графа $\tilde{\Gamma}''$, где $\chi_{n'_{k+1}}(\tilde{\Gamma}'') = 1$. □

4.3.3 Метакластера

Предложенный алгоритм может быть использован для частичной оптимизации схем широковещательных рассылок в метакластерах. В описанном методе никак не учитываются более или менее эффективные способы передачи данных в однородной коммуникационной среде. Однако при непосредственной реализации алгоритма возможно учитывать различные подходы к оптимизации гомогенных широковещательных рассылок. Также не рассмотрена возможность параллельного исполнения некоторых операций, эта задача должна быть возложена на иные средства.

Задачи оптимизации для метакластера можно свести к рассмотрению предыдущей задачи с дополнительным условием при выборе ребра. В указанном алгоритме не наложено никаких ограничений на выбор ребра в случае, если есть несколько рёбер с равным весом. Положим, что в такой ситуации выбирается наиболее близкое к корню ребро. Для простоты рассмотрим модель метакластера, состоящего из двух тесносвязанных компонент, соединённых между собой медленным каналом. Пусть вес ребра внутри каждого кластера равен 1, а между компонентами, например, 5.

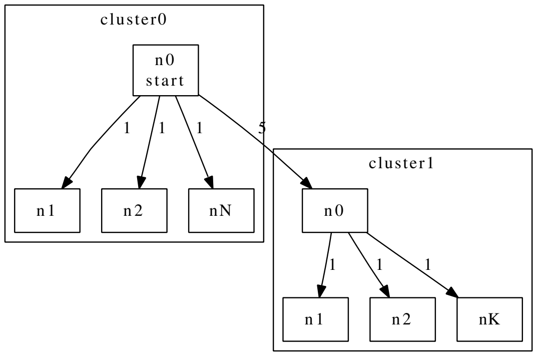

Рис. 5: Граф передач для неоднородного кластера

Тогда начиная с одного из узлов кластера будет построен граф (рис. 5), имеющий несколько рёбер с одинаковым весом выходящих из одной вершины.

Если предположить, что компоненты связности метакластера однородны, то для каждой такой группы рёбер возможно применить оптимизации, рассмотренные в разделе 4.2. После изменения схемы широковещательной рассылки на двоичное дерево результирующий граф будет выглядеть иначе (рис. 6). При этом увеличивается глубина графа, однако при большом числе принимающих узлов внутри одной однородной компоненты этот способ может быть эффективен. Каждый конкретный случай требует тщательного анализа, а также зависит от текущей загрузки магистрального канала между компонентами.

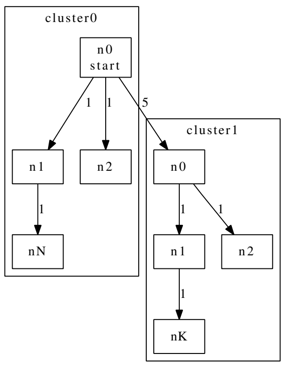

Рис. 6: Оптимизированный граф передач для неоднородного кластера

Эти результаты верны не только для широковещательной рассылки одного файла, но и для иных задач, требующих передачи определенного объёма данных по вычислительному кластеру, например — вычисление избыточной информации по имеющемуся набору контрольных точек.

4.4 Реализация

Алгоритмы кодирования Рида–Соломона описанные выше были реализованы в прикладных средствах создания избыточной информации. Кроме этого, созданное приложение позволяет производить рассылки данных на несколько узлов или копировать между не видящими друг-друга частями системы используя промежуточные точки. Приложение состоит из библиотеки и небольшой интерфейсной части.

Библиотека основана на подходе абстрактного ввода-вывода (bio). Внутри установлен некоторый интерфейс, используя который реализованы различные способы передачи, чтения или записи данных. Это позволяет добавлять использование новых сред передачи информации без вмешательства в уже имеющиеся части. Также это позволяет вводить сложные механизмы, например, кодирование Рида–Соломона, без учета транспортного уровня или источника данных. Возможно объединение нескольких bio в параллельные цепочки, так что одни и те же данные передаются нескольким получателям. Некоторые позволяют объединять в последовательности, так что полученные данные тем или иным образом модифицируются, например для получения избыточной информации. Реализованы следующие абстракции.

Транспортные:

mpi: Передача через асинхронные вызовы библиотеки MPI (MPI_Irecv/MPI_Isend). Использование Irecv/Isend вместо обычных позволяет полнее использовать возможности MPI. Применение MPI, в свою очередь, позволяет использовать оптимизированную под коммуникационную среду реализацию. Большинство тестов проводились именно с этим транспортом.

mpi-bc: Исключительно тестовый механизм, использующий вызовы MPI_Bcast для передачи данных. Был создан для сравнения эффективности с построенными коммуникационными деревьями.

tcp: Передача по протоколу TCP. Позволяет использовать библиотеку вне тесносвязанных кластеров.

Ввод-вывод:

null: Аналогично /dev/null в UNIX-подобных системах, отбрасывает всю полученную информацию, например, для терминирования ненужных потоков. В основном используется в тестовых целях или для проведения замеров производительности, что бы не оказывала влияния дисковая подсистема.

file: Простое считывание или запись в файл, является стандартной начальной или конечной вершиной дерева.

Модифицирующие:

mmap: Надстройка над file, которая позволяет повысить производительность чтения и записи используя механизмы UNIX-подобных операционных систем отображать дисковое пространство в память (mmap). При этом становятся доступны механизмы управления упреждающим чтением и синхронизацией.

xor: Считает XOR-сумму полученных данных из нескольких источников и передает дальше. Используется для построения избыточной информации в алгоритмах повышения отказоустойчивости.

rs: Считает избыточную информацию по алгоритму Рида–Соломона над полем $GF(8)$ полученных данных из нескольких источников и передаёт дальше. Используется, как и xor, для построения избыточной информации в алгоритмах повышения отказоустойчивости.

4.4.1 Построение

Из перечисленных выше абстракций возможно строить древовидные графы; циклы недопустимы, так как приведут к блокировке. Возможно два типа графов, копирование одного файла на несколько узлов, либо создание по набору файлов одинакового размера контрольных сумм. Во втором случае после построения графа необходимо поменять местами входной и выходной bio, это позволяет оставлять без изменений часть программы непосредственно производящей копирование. Например, копирование файла с одного узла на другой будет состоять из абстракций, изображенных на рис. 7 (предполагается, что mmap не используется, а в качестве транспортного уровня выбран TCP/IP).

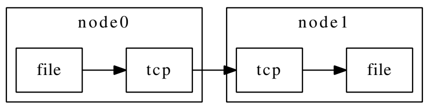

Рис. 7: Передача файла с между узлами

Более сложный случай, с использованием mmap и последовательной передачей через промежуточный узел (рис. 8).

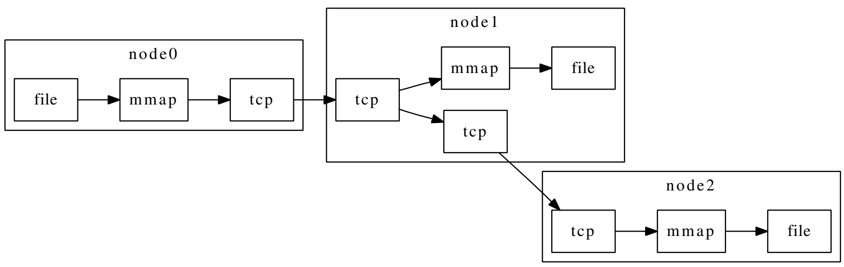

Рис. 8: Передача файла с использованием mmap на два узла

4.4.2 Передача

Передача осуществляется следующим образом. Из принимающего bio считывается блок данных который, возможно пройдя ту или иную обработку, подсчет и протоколирование, передается на выходной bio. Вся остальная работа по передаче скрыта от пользователя и осуществляется внутри bio тех или иных типов. Внутри bio, в случае если требуется передать данные более чем одному получателю, блок последовательно записывается в каждую из принимающих абстракций. В случае асинхронных bio, запись происходит не в момент передачи им данных, а в момент вызова метода синхронизации. Для различных bio он реализован по-разному; так, для mmap, это вызов fsync, для mpi — вызов MPI_wait.

4.4.3 Прикладное приложение

С использованием библиотеки описанной выше собрано приложение, позволяющее копировать данные с узла на узел или создавать избыточную информацию, при этом вычисления производятся распределенно, что, учитывая ненулевую вычислительную сложность кодирования Рида–Соломона, несколько снижает нагрузку. При старте приложению указывается список URL входных потоков, список выходных потоков и флаги модификаторов. При этом задача построения коммуникационной среды возложена на планировщик, запускающий это приложение. В качестве входного или выходного потока может быть указан URL одного из следующих типов:

- file://path: читать или писать данные в указанный файл
- rsfile://rank:path: читать и кодировать данные из указанного файла, ранк файла равен rank
- fcp://host:port: читать или писать данные по сети, в случае, если поток выходной, создается серверный сокет, иначе, открывается соединение с указанным узлом по указанному порту.

Флаги-модификаторы могут быть следующими:

- -m: использовать при чтении и записи в файл метод mmap.
- -c: распределять данные среди выходных потоков по циклу — первый блок первому, второй второму и так далее.
- -x: производить кодирование входных данных считая исключающее ИЛИ (XOR) блоков.
- -s число: установить количество столбцов в матрице кодирования Рида–Соломона как указано в аргументе.
- -S число: установить количество дополнительных строк в матрице Рида–Соломона как указано в аргументе.

4.5 Библиотека оповещения о создании контрольных точек

Данная библиотека позволяет вычислительному приложению оповещать систему обеспечения отказоустойчивости о создании координированной или отдельной локальной контрольной точки. При этом на выделенный демон ccpdпередается пакет по протоколу TCP/IP содержащий всю необходимую информацию. В случае координированной точки демону передается следующая информация:

- uuid задачи, считывается из переменной окружения TASK_UUID при старте вычислительного процесса.

- числовой идентификатор отправившего процесса, каждый имеет уникальный номер начиная с 0, посылка информации о координированной точки обычно происходит одним выделенным процессом.

- время создания контрольной точки

- количество файлов в наборе

- для каждого файла передается

  – абсолютное имя файла
  – uuid узла или файловой системы, по которому можно определить расположение файла
  – числовой идентификатор процесса, аналогичный описанному выше
  – последовательный номер файла контрольной точки
  – контрольная MD5 сумма файла, в данный момент не используется, нужна для обеспечения целостности данных контрольной точки
  – размер файла контрольной точки, несет в общем то информационный характер, в данный момент заменяет контрольную сумму

В случае создания локальной контрольной точки информация передающаяся на сервер отличается в следующем:

- передается только путь к одному файлу, его размер и uuid узла на котором данный файл создан

- для остальных передаются лишь последовательные номера точек, существовавших на момент создания и образующих с локальной координированный набор

Это позволяет серверу собирать координированные наборы по приходящим сообщениям, удалять уже ненужные или создавать избыточную информацию по актуальным файлам.

4.5.1 Пользовательский интерфейс

Пользовательский интерфейс библиотеки состоит из вызовов инициализации, непосредственно передачи сообщений и вызова завершения работы. Ниже они описаны более подробно

`ccpc_flags_set(int flags)`

Устанавливает различные флаги, так или иначе изменяющие поведение библиотечных вызовов. Возможны различные логические ИЛИ следующих значений:

`CCPC_FLAGS_DEBUG`

Дополнительная отладочная распечатка о посылаемой на сервер информации о контрольной точке.

`CCPC_FLAGS_NODENAME`

Брать в качестве UUID файловой системы первые 16 символов имени узла, получаемого через uname

`CCPC_FLAGS_ENVUUID`

Брать в качестве UUID приложения значение из переменной окружения с именем TASK_UUID. В случае если она содержит некорректно отформатированное значение будет выдано сообщение об ошибке и идентификатор приложения будет взят нулевым.

`CCPC_FLAGS_SEQ_ID`

В случае, если выставлен этот флаг, то на сервер будут отсылаться не имена файлов а номера контрольных точек для соответствующих узлов. Имя файла будет создано только для локальной точки. Подробнее об этом в описании функции ccpc_register

`ccpc_flags_unset(int flags)`

Аналогично функции ccpc_flags_set, только снимает флаги указанные в качестве аргумента.

`ccpc_init(int numproc, int templ_size, const char * templ)`

Заполняет внутренние структуры информацией о файловой системе и идентификаторе приложения, а также инициализирует шаблоны имен файлов контрольных точек. Шаблон может содержать две подстановки целочисленных переменных в формате описанном для библиотечной функции printf. Первая будет заменена на номер узла для которого создается файл контрольной точки, вторая — на номер точки. Пример: "e2d.%02d-%04d.ccp". Данный вызов должен быть произведен до любого применения ccpc_register. Перед ним возможно вызывать только ccpc_flags_set или ccpc_flags_unset

`ccpc_init_serv(const char * serv_addr)`

Устанавливает IP адрес сервера, получающего сообщения. Возможно любое корректное значение в стандартном формате.

`ccpc_uuid_get_fs ( char * uuid )`

Возвращает значение полученное для UUID локальной файловой системы. В данный момент это либо нули либо первые 16 символов имени узла полученного через вызов uname. Это зависит от того, выставлен ли флаг CCPC_FLAGS_NODENAME. Аргумент должен ссылаться на область памяти больше 16 байт, значение UUID будет возвращено в двоичном виде а не в печатном формате.

`ccpc_uuid_set( int rank, const char * uuid )`

Устанавливает значение UUID файловой системы для узла с идентификатором rank. Используется впоследствии при заполнении данных о файле контрольной точке, не используется при установленном флаге CCPC_FLAGS_SEQ_ID. Значение uuid — 16 байт двоичной информации, а не UUID в стандартном формате. Может быть использован с вызовом ccpc_uuid_get_fs для обмена информацией о UUID файловых систем на различных узлах.

`ccpc_rank_local(int rank)`

Устанавливает идентификатор локального узла, используется только при выставленном флаге CCPC_FLAGS_SEQ_ID.

`ccpc_register(const int * seq_id, int size)`

Отправить на сервер информацию о наборе файлов для контрольной точки. В параметре seq_id передается массив последовательных номеров файлов локальных точек на момент, когда был создан файл на данном узле. При этом будут произведены подстановки в шаблон, заданный вызову ccpc_init и полученные имена отправлены на сервер. В случае, если выставлен флаг CCPC_FLAGS_SEQ_ID, подстановка будет проведена только для локального узла, номер которого должен быть выставлен вызовом ccpc_rank_local. Для остальных будет отправлено не имя файла, а только номер точки.

`ccpc_terminate()`

Отсылает на сервер данные о последней точке, прошедшей вызов ccpc_register. При этом, устанавливаются флаги, указывающие о том, что это последнее сообщение от данного процесса. После этого не должно быть вызовов ccpc_register, только ccpc_fini. Данная функция имеет смысл лишь при отсутствии флага CCPC_FLAGS_SEQ_ID и если вся информация отсылается одним выделенным процессом задания.

`ccpc_fini()`

Освобождает все ресурсы, занятые вызовами данной библиотеки.

4.5.2 Серверный процесс

Серверный процесс принимает пакеты от пользовательских приложений, разбирает их и вызывает внешнюю команду, указанную ему при старте. В случае координированной контрольной точки, внешней команде в качестве аргументов передается имя файла, UUID узла или файловой системы и контрольная MD5 сумма файла. Логика создания избыточной информации и построения коммуникационной сети возложена именно на внешнее приложение. Это сделано что бы упростить серверный процесс, но также такой подход дал возможность использовать язык отличный от C, на котором написан демон.

Скрипт для циклического сдвига файлов контрольной точки написан на языке shell (sh/bash) с использованием стандартного приложения make. По списку точек определяются узлы на которые требуется произвести копирование, создается соответствующий Makefile и исполняется в параллельном режиме, так как передача ведется по частично независимым каналам.

Заключение

В ходе работы были рассмотрены различные алгоритмы обеспечения отказоустойчивости распределенных приложений в вычислительных системах с локальной дисковой памятью. Часть из них, признанная эффективными, была реализована в качестве средств обеспечения отказоустойчивости. Созданные средства были собраны в тестовый макет с использованием вычислительной задачи расчёта течений с тепловыми неоднородностями в потоке.

В качестве перспектив развития стоит отметить возможность интегрировать эти инструментальные средства в системы управления распределенными вычислительными средами, например в GLOBUS.

Список литературы

[1] James S. Plank. Improving the Performance of Coordinated Checkpointers on Networks of Workstations using RAID Techniques.
(http://www.cs.utk.edu/~ plank/plank/papers/SRDS-15.ps)

[2] James S. Plank. A Tutorial on Reed–Solomon Coding for Fault-Tolerance in RAID-like Systems.
http://www.cs.utk.edu/~ plank/plank/papers/CS-96-332.ps

[3] James S. Plank, Ying Ding. Correction to the 1997 Tutorial on Reed–Solomon Coding.
http://www.cs.utk.edu/~ plank/plank/papers/CS-03-504.ps

[4] Nitin H. Vaidya. A Case for Two-Level Distributed Recovery Systems.
http://www.crhc.uiuc.edu/~ nhv/old.papers/fault-tolerance/SIGMETRICS-95.ps

[5] D. Manivannan, Mukesh Singhal.
Quasi-Syncronous Checkpointing: Models, Characterization, and Classification.
TR No. OSU-CISRC-5/96-TR33, Dept. of Computer and Information Science, The Ohio State University

[6] Отчет о научно-исследовательской работе по проекту «Модели распределенной общей памяти (DSM) и файловые системы для многопроцессорных вычислительных комплексов» в рамках программы фундаментальных исследований Президиума РАН на 2004 год №21 «Разработка фундаментальных основ создания распределенной информационно-вычислительной среды на основе технологии GRID» (годовой научный отчет)