Сальников А. Н.

Вычислительная система на примере кластера UNIX серверов. Взгляд пользователя.

Московский Государственный Университет имени
М. В. Ломоносова,
факультет вычислительной математики и кибернетики.

конспект лекций

2007 г.

Оглавление

1. Введение

1.1. Задача курса

Основная задача курса — познакомить слушателей с современными технологиями, популярными подходами, а также некоторыми известными алгоритмами, на основе которых создаются кластерные вычислительные системы. Предполагается, что данный курс будет интересен как пользователям кластерных вычислительных систем, так и администраторам вычислительных кластеров построенных на базе UNIX серверов. Курс может быть полезен в том числе для лиц, принимающих решения об аппаратной и программной составляющей создаваемого кластера. Изложенные в курсе подходы и алгоритмы могут быть использованы как отправная точка в научных исследованиях студентов в области сетевых технологий, операционных систем, систем поддержки параллельных и распределённых вычислений, распределённых файловых систем.

1.2. Структура курса

Курс состоит из восьми глав:

1. Введение
2. Общая структура вычислительного комплекса
3. Вычислительная система с точки зрения пользователя
4. Программное обеспечение для разработки программ на вычислительных системах

5. Организация системы очередей задач пользователей

В курсе будут рассмотрены некоторые вопросы связанные с администрированием вычислительных систем. Будет показано устройство вычислительных систем как набора аппаратных компонент, приведены примеры некоторых вычислительных систем. В курсе будет обсуждено программное обеспечение, используемое для организации работы вычислительных систем, построенных как кластер серверов, работающих под управлением UNIX-подобных операционных систем. Обсуждаются программные средства, используемые для их администрирования. В курсе будут обсуждены некоторые ситуации, в которых требуется вмешательство системного администратора вычислительной системы.

В курсе не будут рассмотрены задачи, связанные с написанием параллельных программ и их оптимизацией для многопроцессорных вычислительных систем.

Кроме того, некоторые разделы курса актуальны не только для кластерных вычислительных систем, но и для систем работающих под управлением UNIX-подобных операционных систем вообще.

Первая глава — введение.

Во **второй главе** будут рассмотрены общие принципы построения вычислительного комплекса. Будут рассмотрены виды и некоторые известные примеры архитектур вычислительных комплексов. Будут рассмотрены некоторые архитектуры процессоров, памяти и подсистем коммуникаций, используемые в современных вычислительных системах, предназначенных для решения ресурсоёмких

научно-практических задач. Также будет рассказано о аппаратных принципах построения файловых хранилищ и обсуждены некоторые современные технологии их создания. Кроме того, будет сделан обзор классов программного обеспечения, специфических для многопроцессорных вычислительных комплексов.

Третья глава посвящена взгляду пользователя на вычислительную систему. Обсуждаются вопросы организации удалённого доступа к вычислительной системе: интерактивное взаимодействие с интерфейсной машиной вычислительного комплекса; просмотр состояния очередей и загруженности отдельных компонент; вопросы передачи файлов между пользовательским рабочим местом и вычислительным комплексом. Производится обзор некоторых основных команд shell и наиболее типичных редакторов, присутствующих на вычислительном комплексе. Делается акцент на рассмотрении пользовательского интерфейса системы очередей (в отличие от администраторского интерфейса).

Четвёртая глава посвящена обзору основных компиляторов устанавливаемых на кластерных вычислительных системах, их основным параметрам и способам оптимизации создаваемого ими кода. Будут рассмотрены основные функции отладчика на примере gdb. Будут обсуждены вопросы переноса данных с одной многопроцессорной системы на другую на примере библиотеки NetCDF.

В **пятой главе** рассматриваются некоторые стратегии планирования вычислений на многопроцессорных системах, в том числе алгоритмы «справедливого» распределения ресурсов, такие как Backfill и Gang; наиболее известные системы ведения

очередей задач пользователей; конфигурация некоторых систем ведения очередей задач пользователей (maui, LoadLeveler) с точки зрения администратора; способы разделения задач пользователей по приоритетам.

1.3. Актуальность данного курса

Существует некоторое количество научно-практических задач, решение которых требует большого количества ресурсов при решении их на вычислительных системах. Такие задачи могут быть требовательны к ресурсу процессорного времени, но так же могут быть чрезвычайно требовательны к ресурсу памяти.

Практика показывает, что использование однопроцессорной системы даже очень высокой производительности как правило не позволяет эффективно решать данные задачи за приемлемое время. В случае, когда задача оперирует с огромным количеством данных, как например при обработке экспериментальных данных полученных с ускорителя элементарных частиц, бывает очень трудно использовать одну какую-то многопроцессорную систему, поскольку данные на неё не будут помещаться.

Объединить процессоры в единый вычислительный комплекс можно различными способами. Наиболее широкое распространение получило создание вычислительных систем на основе кластеров UNIX-серверов. Это связано с тем, что кластеры обладают рядом преимуществ:

1. Они хорошо масштабируются по числу процессоров

2. Распределённость по памяти и дисковому пространству позволяет решать задачи, которые требуют большого количества ресурсов и не могут быть размещены в памяти отдельного узла вычислительной системы
3. На основе кластеров можно создавать системы с такой производительностью, которая не может быть достигнута другими способами или достигается неоправданно большими затратами на стоимость оборудования и поддержания эффективности (отношение реальной производительности к теоретической)

В связи с этим становится актуальной проблема создания и сопровождения кластерных систем, в частности, их администрирования.

2. Общая структура вычислительного комплекса

2.1. Что такое вычислительный комплекс

Вычислительный комплекс — комплекс, служащий для выполнения различных вычислительных задач. Типичный вычислительный комплекс состоит из набора компонент (см. Рисунок 2.1.1):

1. Вычислительные системы (одна или несколько)
2. Файловые хранилища
3. Интерфейсная машина
4. Сервера, предоставляющие публичные сервисы (web, почта, …)
5. Рабочие станции
6. Средства контроля взаимодействия со внешним миром
7. Управляющий сервер

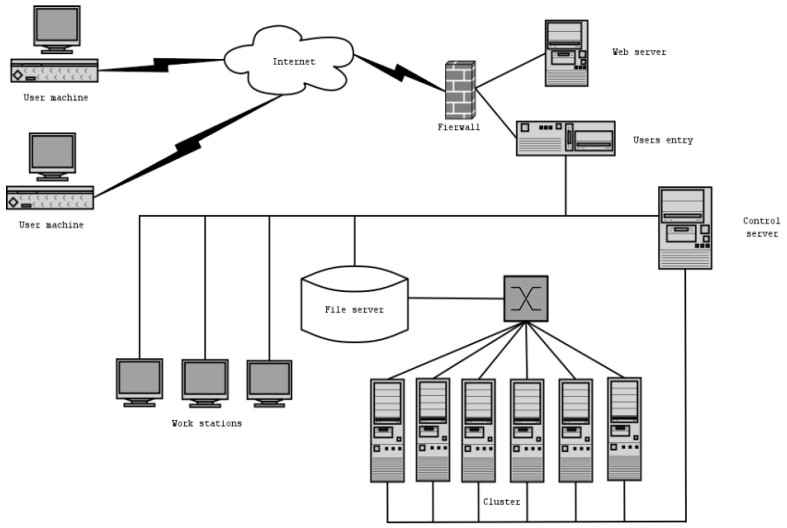

Рисунок 2.1.1: Структура вычислительного комплекса

Кратко опишем систему и её отдельные компоненты. Подробнее они будут рассмотрены далее по курсу.

Узел вычислительной системы — отдельный компьютер, который работает под управлением своей собственной операционной системы и который объединён сетью с другими узлами.

Вычислительный система — объединённый в сеть набор узлов, выполняющий вычисления и представляемый как единая система.

Файловое хранилище — система, предназначенная для хранения данных (результаты экспериментов, вычислений, резервные копии)

Интерфейсная машина — машина, предназначенная для взаимодействия системы и удалённых пользователей.

Рабочие станции — (в данном контексте) набор машин, имеющих локальный доступ к вычислительной системе.

Управляющий сервер — сервер, контролирующий работу вычислительной системы.

2.2. Вычислительная система

После краткого обзора отдельных компонент ВК перейдём к более подробному их рассмотрению. Начнём с наиболее значимой части ВК, составляющей её основу — вычислительной системы. Ниже рассмотрим архитектуры существующих ВС, некоторые наиболее яркие примеры и компоненты, из которых узлы ВС состоят: процессоры, память, коммуникационную подсистему.

2.2.1. Классификация вычислительных систем

Обсудим классификацию многопроцессорных систем по принципу их организации. Наиболее популярным способом классификации многопроцессорных систем является классификация по способу доступа в память. Можно выделить три класса: симметричные многопроцессорные системы (Symmetric Multiprocessing, SMP), в которых доступ память равноправен для каждого процессора; системы с неоднородным доступом в память (Non-Uniform Memory Access, NUMA), системы с рапределённой памятью (Massively parallel processing, MPP). Кластерные системы являются вариантом MPP.

SMP и NUMA системы относятся к классу так называемых систем с общей памятью. Каждый процессор имеет доступ к произвольной ячейке оперативной памяти присутствующей в многопроцессорной системе. В случае с SMP системами время доступа не зависит от того, какой номер у процессора и по какому адресу происходит обращение. Для NUMA систем время доступа зависит от номера процессора и адреса в памяти.

В MPP системах с процессором связывается область памяти. Каждый процессор может напрямую обращаться к своей собственной области памяти и не может напрямую обратиться не к своей области памяти. Для обращения к чужой памяти используется какой либо специальный механизм передачи данных — обычно механизм передачи сообщений.

При поддержке команд обработки векторных данных говорят о векторно-конвейерных процессорах, которые, могут объединяться в PVP-системы (Parallel Vector Processor) с использованием общей или распределенной памяти.

Тем не менее, вне зависимости от архитектуры, необходимо решать один и тот же набор задач: управления очередями, организация пользовательского доступа, и так далее.

Ниже мы рассмотрим каждый упомянутый выше класс архитектур подробнее.

2.2.1.1. Симметричные мультипроцессорные системы (SMP)

SMP-системы состоят из некоторого количества однородных процессоров и массива памяти,

разделяемой между процессорами. Самое простое решение — это когда процессоры обращаются к памяти через общую системную шину. Обнаружилось, что такая архитектура плохо масштабируется по отношению к числу процессоров. Проблема заключается в росте числа коллизий с увеличением числа процессоров при обращении к общей шине. Эту проблему удалось частично решить с помощью следующего приёма. Оперативная память делится на несколько блоков, где каждый блок подключен к своей общей шине, а общие шины по числу блоков связываются с каждым из процессоров через полносвязанный коммутатор. Таким образом удаётся снизить число конфликтов на шине и сделать доступ в память для процессоров параллельным. При этом обращение в память для любого процессора всё-равно происходит за одинаковое время. К сожалению, стоимость оборудования за счёт коммутатора значительно вырастает и построить такую систему с большим числом процессоров довольно сложно (больше чем 32).

Современные системы SMP архитектуры состоят, как правило, из нескольких однородных серийно выпускаемых микропроцессоров и массива общей памяти, подключение к которой производится с помощью либо общей шины, либо коммутатора, либо комбинированным способом, как это было показано ранее. (см. Рисунок 2.2.1.1).

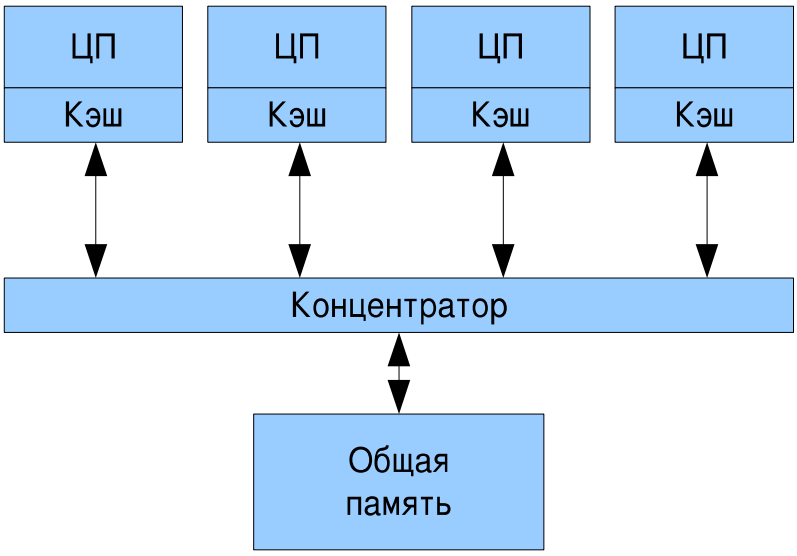

Рисунок 2.2.1.1. Схема архитектуры класса SMP

Наличие общей памяти значительно упрощает организацию взаимодействия процессоров между собой и упрощает программирование, поскольку параллельная программа работает в едином адресном пространстве. Однако существует ряд проблем, присущих системам этого типа. Все они, так или иначе, связаны с оперативной памятью. Помимо конфликтов при обращении к общей шине памяти возникла проблема, связанная с наличием кэш-памяти. В многопроцессорных системах, построенных на базе микропроцессоров со встроенной кэш-памятью, нарушается принцип равноправного доступа к любой точке памяти. Данные, находящиеся в кэш-памяти некоторого процессора, недоступны для других процессоров. Это означает, что после каждой модификации копии некоторой переменной, находящейся в кэш-памяти какого-либо процессора,

необходимо производить синхронную модификацию самой этой переменной, расположенной в основной памяти, что порождает большие накладные расходы и, как следствие, падение производительности.

Примерами SMP систем являются: HP 9000 (Exemplar), Sun Fujitsu PRIMEPOWER 850.

2.2.1.2. Системы с неоднородным доступом к памяти (NUMA)

Для решения проблем, возникающих в симметричных многопроцессорных системах, было решено пожертвовать однородностью и организовать неоднородный доступ к памяти (Non-Unifom Memory Access, NUMA) при сохранении единого для всех процессоров адресного пространства.

Система состоит из однородных базовых модулей (плат), состоящих из небольшого числа процессоров. С каждым модулем обычно связан один или несколько блоков памяти. Модули объединены с помощью высокоскоростного коммутатора. Поддерживается единое адресное пространство для всех модулей. Аппаратно поддерживается доступ к удаленной памяти, т. е. к блокам памяти приписанным другим модулям. Скорость доступа из модуля к блокам памяти приписанным данному модулю в несколько раз быстрее, чем к остальным блокам памяти.

В случае, если аппаратно поддерживается когерентность кэшей во всей системе (обычно это так), говорят об архитектуре cc-NUMA (cache-coherent NUMA).

Масштабируемость NUMA-систем ограничивается объемом адресного пространства, возможностями

аппаратуры поддежки когерентности кэшей и возможностями операционной системы по управлению большим числом процессоров. На настоящий момент, максимальное число процессоров в NUMA-системах составляет 256 (Origin2000). Другим представителем данного класса многопроцессорных систем является IBM pSeries 690.

2.2.1.3. Массивно-параллельные системы (MPP)

Следующим шагом стало решение отказаться от единого адресного пространства для всех процессоров многопроцессорной системы. В MPP системах каждому процессору приписана его собственная оперативная память. Другие процессоры не имеют возможности обратиться к не своей памяти, однако процессоры связаны коммуникационной средой. Каждый процессор имеет возможность передать другому процессору сообщение через коммуникационную среду. Обычно коммуникационная среда устроена таким образом, что сообщения для других процессоров доставляются транзитным образом через третьи процессоры. О таких системах говорят как о системах с распределённой памятью. (см. Рисунок 2.2.1.2).

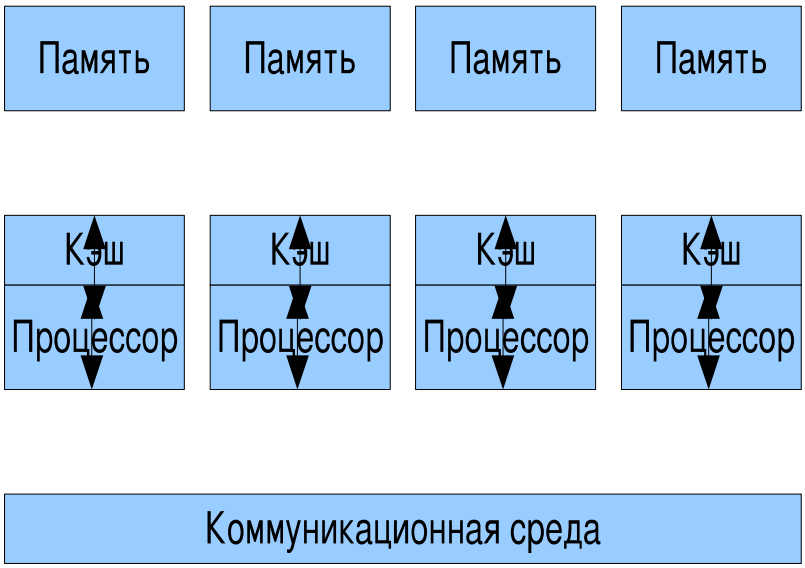

Рисунок 2.2.1.2: Структура системы класса MPP

Тройку процессор, память, оборудование для коммуникаций обычно выносят на отдельную печатную плату и в дальнейшем называют узлом MPP системы. При этом на каждом узле может функционировать либо полноценная операционная система (как в системе RS/6000 SP2), либо урезанный вариант, поддерживающий только базовые функции ядра, а полноценная ОС работает на специальном управляющем компьютере (как в системах Cray T3E, nCUBE2). Сейчас на узел MPP системы довольно часто устанавливают несколько процессоров или многоядерные процессоры. В результате узел MPP системой может быть SMP системой. В узлах аппаратные средства для организации коммуникаций устроены так, что освобождают центральный процессор от участия в трансляции данных.

Такая архитектура вычислительной системы устраняет одновременно как проблему конфликтов при обращении к памяти, так и проблему когерентности кэш-памяти. Это дает возможность практически неограниченного наращивания числа процессоров в системе. Успешно функционируют MPP системы с сотнями и тысячами процессоров (ASCI White - 8192, Blue Mountain - 6144). Производительность наиболее мощных систем достигает 10 Tflops. Важным свойством MPP систем является их высокая степень масштабируемости. В зависимости от вычислительных потребностей для достижения необходимой производительности требуется просто собрать систему с нужным числом узлов.

Тем не менее, у данного класса так же имеется ряд своих особенностей. Для MPP систем на первый план выходит проблема эффективности коммуникационной среды. Для MPP систем чрезвычайно важно как узлы связаны между собой. От количества транзитов (передач через промежуточные узлы), при передаче сообщения, зависит эффективность работы коммуникаций в MPP системах. Способ объединения узлов для передачи сообщений через другие узлы MPP системы задаётся так называемой топологией коммуникационной среды. Топологии существующих MPP систем весьма разнообразны. В Intel Paragon процессоры образовывают прямоугольную двумерную сетку. Для этого в каждом узле достаточно четырех коммуникационных каналов. В компьютерах Cray T3D/T3E применяется топология трехмерного тора. Соответственно, в узлах этого компьютера имеется шесть коммуникационных каналов. Фирма nCUBE использует в своих компьютерах топологию n-мерного гиперкуба. Иногда для соединения вычислительных узлов используется иерархическая система высокоскоростных коммутаторов. Это впервые было

реализовано в компьютерах IBM SP2. Такая топология дает возможность прямого обмена данными между любыми узлами, без участия в этом промежуточных узлов.

Системы с распределенной памятью подходят для параллельного выполнения независимых программ, поскольку при этом каждая программа выполняется на своем узле и никаким образом не влияет на выполнение других программ. При этом отсутствует возможность прямого доступа к данным, расположенным в других узлах. Перед использованием эти данные должны быть предварительно переданы на тот узел, который в данный момент в них нуждается. Кроме того, обмены данными между узлами выполняются значительно медленнее, чем обработка данных в локальной оперативной памяти узлов. Для написания эффективных параллельных программ для MPP систем необходимо учитывать все эти особенности. Написание программ для MPP систем сильно отличается от написания программ для SMP систем.

2.2.1.4. Кластерные системы

Кластерные системы отличаются от MPP систем прежде всего тем, что узлы в кластера — полноценные компьютеры каждый из которых управляется своей собственной копией операционной системы. Развитие коммуникационных технологий, а именно, появление высокоскоростного сетевого оборудования и специальных библиотек, таких как MPI, реализующих механизм передачи сообщений над стандартными сетевыми протоколами, сделали кластерные технологии более доступными.

Кластер — связанный набор полноценных компьютеров, используемый в качестве единого вычислительного ресурса.

С точки зрения простоты сборки и простоты использования кластер обладает большим числом преимуществ перед MPP системами.

- Кластер также легко масштабируем по числу процессорных узлов как и MPP системы. (За исключением проблем связанных с объёмом выделяемой и потребляемой энергии.)
- Для управления кластером не нужно ставить специальную операционную систему.
- Коммуникационная среда обычно реализована поверх известных и достаточно популярных сетевых технологий.

Для создания кластеров обычно используются как «простые» однопроцессорные персональные компьютеры, так и многопроцессорные SMP(NUMA) серверы. При этом не накладывается никаких ограничений на состав и архитектуру узлов. Каждый из узлов функционирует под управлением своей собственной операционной системы. Чаще всего используются распростанённые ОС: Linux, FreeBSD, Solaris, AIX, Windows NT. В тех случаях, когда узлы кластера неоднородны, то говорят о *гетерогенных* кластерах.

При создании кластеров можно выделить два подхода. Первый подход применяется при создании небольших кластерных систем. В кластер объединяются полнофункциональные компьютеры, которые продолжают работать и как самостоятельные единицы,

например, компьютеры учебного класса или рабочие станции лаборатории. Второй подход применяется в тех случаях, когда целенаправленно создается мощный вычислительный ресурс. Тогда системные блоки компьютеров компактно размещаются в специальных стойках, а для управления системой и для запуска задач выделяется один или несколько полнофункциональных управляющих сервера.

Существует множество сетевых технологий для организации коммуникационной среды в кластерных системах. Наиболее доступная в данный момент времени технология Gigabit Ethernet.

Существуют также технологии разрабатывающиеся специально для организации коммуникаций в вычислительном кластере. Например технология SCI фирмы Scali Computer (~100 MB/s) и Mirynet (~120 MB/s). Активно включились в поддержку кластерных технологий и фирмы-производители высокопроизводительных рабочих станций (SUN, HP, Silicon Graphics).

Как правило программирование для подобных систем, осуществляется в рамках модели передачи сообщений. В качестве реализации модели передачи сообщений используется одна из доступных реализаций MPI. Например: OpenMPI, mpich, Intel-MPI, Scali MPI, poe для оборудования от IBM.

2.2.1.5. Параллельные векторные системы (PVP)

Вариантом симметричных многопроцессорных систем являются векторные параллельные системы, основным признаком которых является наличие специальных векторно-конвейерных процессоров, в которых предусмотрены команды однотипной обработки векторов независимых данных.

Как правило, несколько таких процессоров (1—16) работают одновременно над общей памятью (аналогично SMP) в рамках многопроцессорных конфигураций. Несколько таких узлов могут быть объединены с помощью коммутатора (аналогично MPP).

2.2.2. Примеры вычислительных систем

2.2.2.1. Sun Fujitsu PRIMEPOWER 850

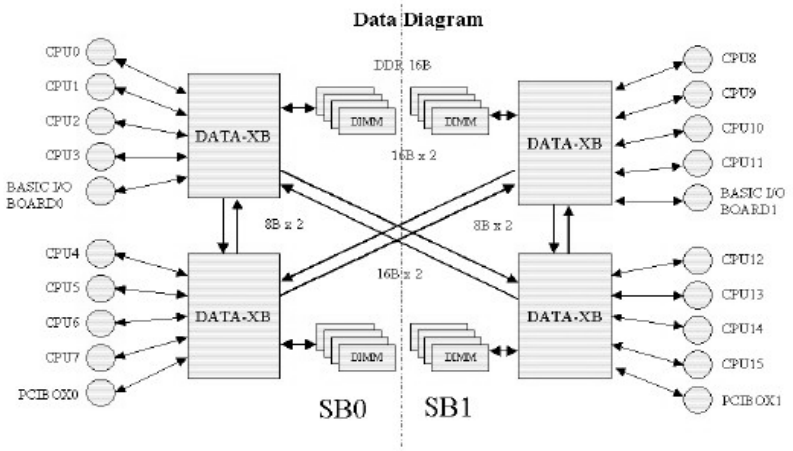

Рисунок 2.2.2.1. Подсистема доступа к памяти Primepower 850

Fujitsu Sun PRIMEPOWER 850 сервер (в дальнейшем pp850), установлен в Научно-исследовательском институте физико-химической биологии им. А. Н. Белозерского. Используется в первую очередь для расчётов, связанных с биологией и химией. В pp850 установлено 16 процессоров SPARC64-V 1,89 GHz, кэш первого уровня (L1) 256 KiB на процессор, 3 MiB кэш второго уровня на процессор, 128 GB оперативной памяти в максимальной конфигурации.

Как показано на рисунке 2.2.2.1, подсистема доступа в оперативную память разделена на 4 блока. Каждый блок соединяет 4 процессора и один контролер периферийных устройств. Каждый блок ответственен за когерентность кэшей второго уровня. Адресное пространство оперативной памяти распределено между блоками таким образом, что сперва приложение обращается к паре блоков, а затем в случае обращения по адресу с большим номером происходит обращение к остальным 2 блокам и в этом случае адреса начинают распределяться уже по четырём блокам. Каждый блок управляет четырьмя модулями памяти, и адреса, по аналогии с блоками, также распределены. Блоки соединены между собой каналом связи с частотой работы 540 MHz, что в конечном случае для всей системы целиком позволяет передавать 41,8 GiB в секунду. Вся система в целом по доступу в память является SMP системой.

2.2.2.2. IBM pSeries 690

Один из серверов IBM eServer pSeries 690 установлен факультете ВМиК МГУ и имеет имя Regatta. Данная вычислительная система работает под управлением операционной системы AIX5L. На IBM pSeries 690 можно установить до 16 2-х ядерных

процессоров на базе кристалла IBM Power4. С точки зрения классификации архитектур вычислительных систем по доступу в память pSeries 690 является системой типа ccNUMA.

Система состоит из процессорной подсистемы и до восьми корпусов ввода-вывода. Процессорная подсистема состоит из 1—4 многокристальных модулей (MCM), каждый из которых содержит четыре двухядерных кристалла Power4, образующих конструктивный блок процессорной подсистемы с 8 ядрами. Каждый корпус ввода-вывода содержит 20 слотов ввода-вывода PCI и до 16 отсеков для дисковых накопителей.

Ключевым аспектом конструкции процессорной подсистемы является упаковка полупроводниковых кристаллов Power4 на одну керамическую подложку. На многокристальном модуле (MCM) смонтировано четыре процессорных кристалла Power4, составляющие вместе 8-ядерный конструктивный блок SMP. Каждый модуль реализует полносвязанный граф коммуникаций между чипами Power4. Это показано на рисунке 2.2.2.2.

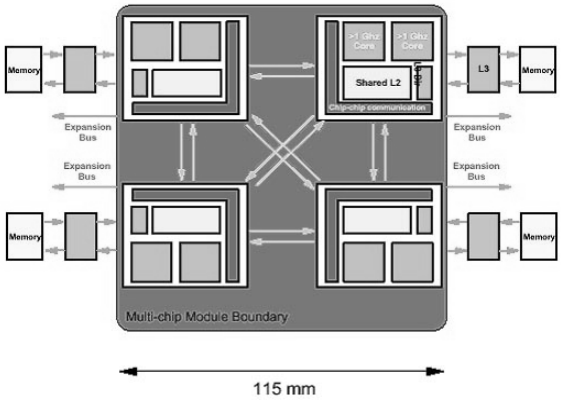

Рисунок 2.2.2.2. модуль MCM

В максимальной 32-ядерной конфигурации четыре модуля MCM, каждый из которых насчитывает четыре кристалла Power4, то есть 8 ядер, соединены вместе с помощью каналов связи модуль-модуль. Каждый модуль MCM имеет доступ к кэшу L3 объемом до 128 MiB. Поддерживается синхронная динамическая оперативная память с коррекцией ошибок ECC SDRAM объемом до 256 GiB. Окончательный вариант сборки системы представлен на рисунке 2.2.2.3.

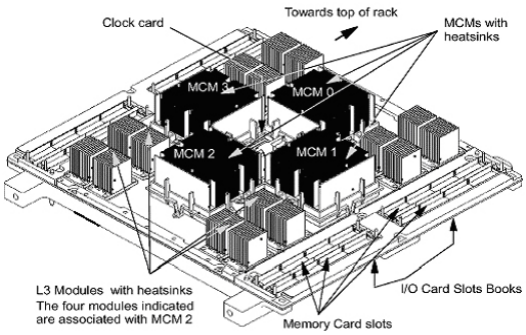

Рисунок 2.2.2.3. Системная плата pSeries 690

2.2.2.3. IBM Blue Gene

Архитектура Blue Gene разрабатывалась как архитектура следующего поколения для обеспечения скачка в производительности от терафлопов к петафлопам. В текущий момент достигнута производительность в 360 терафлоп. Архитектура разрабатывается в США несколькими организациями: IBM, Ливерморская Национальная Лаборатория и Министерство энергетики. Архитектура Blue Gene относится к MPP классу архитектур.

На рисунке 2.2.2.4 изображена общая структура IBM Blue Gene. Выделено несколько структурных единиц.

- **Процессор** (chip). В Blue Gene/L используется духядерный процессор на 700 Mhz. PowerPC 440, который способен производить 4 операции с плавающей точкой на ядре одновременно. Пиковая

производительность процессора 2,8 гигафлопа на оба процессорных ядра.

- **Вычислительная плата** (compute card). На печатную плату небольшого размера устанавливается два двухядерных процессора и модули памяти. Память устанавливается в объёме 1Gb, по 512 на каждый процессор.
- **Плата ввода-вывода**. Плата содержит процессор, память и контроллер ввода-вывода. При помощи них например может быть осуществлён доступ к удалённому файловому серверу через сеть Ethernet.
- **Плата узлов** (node card). Печатная плата содержащая оборудование для организации коммуникаций и подвода питания для вычислительных плат и плат ввода-вывода. На плату узлов может быть установлено 16 вычислительных плат и 2 или 4 платы ввода-вывода.
- Затем платы узлов собираются в **стойку** и набор стоек образует многопроцессорную систему.

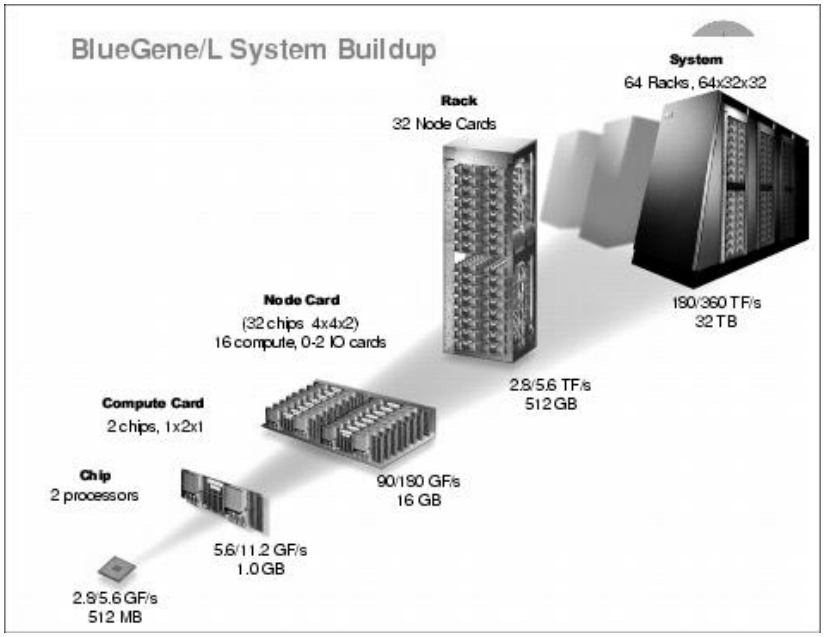

Рисунок 2.2.2.4. Компоновка узлов IBM Blue Gene.

В целом можно считать, что роль вычислительного узла в Blue Gene выполняет вычислительная плата, а сам вычислительный узел является SMP системой. Вычислительные узлы в Blue Gene не могут напрямую иметь доступ к подсистеме ввода-вывода. Доступ к внешнему миру происходит транзитным образом, с использованием памяти промежуточных узлов, через платы ввода вывода, которые ассоциированы с одним узлом.

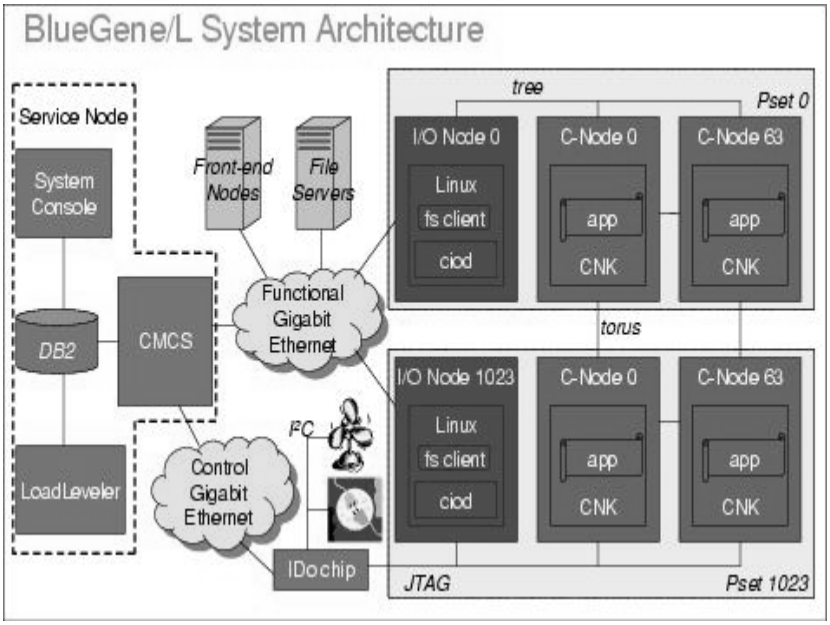

Рисунок 2.2.2.5. Общая структура IBM Blue Gene

2.2.3. Процессоры

Процессор является основной составляющей любой многопроцессорной системы. На данный момент существуют различные архитектуры процессоров, предлагаемые различными фирмами-производителями, в том числе Ниже будут рассмотрены те архитектуры, которые наиболее часто используются в многопроцессорных системах, в частности, в кластерах.

2.2.3.1. Intel Itanium

Первые образцы 64-разрядного процессора Intel представляют собой картридж размером примерно 10x6 см, который включает в себя кэш-память третьего уровня емкостью 2 либо 4 Мбайт и радиатор. Картридж

монтируется в разъем типа Slot и имеет 418 выводов. Процессор имеет трехуровневую иерархию кэш-памяти. Если кэш-память первого и второго уровней интегрирована на кристалле процессора, то микросхемы кэш-памяти третьего уровня расположены на самой плате картриджа. На реализацию процессора с соблюдением проектных норм 0,18 мкм требуется около 320 млн. транзисторов, из которых только 25 млн. пришлось на реализацию самого ядра, а остальные — на кэш-память. Самый большой модуль процессора — это блок вычислений с плавающей запятой, он занимает около 10% площади кристалла. Производительность Itanium составляет до 6,4 млрд. операций с плавающей запятой в секунду. Благодаря архитектуре EPIC и 15 исполнительным устройствам процессор может выполнять до 20 операций одновременно. При этом он может непосредственно адресовать до 16 Тбайт памяти при пропускной способности до 2,1 Гбайт/с. В процессоре реализована поддержка всех расширений Intel (технологий MMX, SIMD и симметричной мультипроцессорной обработки), за исключением SSE2.

Одна из самых интересных деталей в плане размещения узлов процессора — это система синхронизации работы узлов. Одновременная передача тактовых импульсов при большой площади процессора представляет сложную задачу для разработчиков, поскольку задержки в распространении импульсов тактового генератора могут вызывать рассинхронизацию узлов. Для этой цели по всей площади кристалла разместили большое число точек распространения тактовых импульсов.

Для двух- и четырехпроцессорных систем Intel выпустила специальный набор микросхем Intel 460GX, которые могут включаться каскадно, увеличивая число

одновременно используемых процессоров. Поскольку конфигурация таких систем изначально предусматривает объемы оперативной памяти в несколько гигабайт, то в системах Itanium применяются сравнительно недорогие микросхемы памяти типа SDRAM. При этом для увеличения производительности, по словам представителей Intel, используются такие методы, как буферирование, чередование и деление памяти на несколько банков. Набор микросхем реально поддерживает работу с 64 Гбайт памяти при максимальной пропускной способности 4,2 Гбайт/с, хотя 64-разрядная адресация памяти теоретически позволяет обращаться к гораздо большему количеству адресов.

Процессор содержит по 128 целочисленных регистров и регистров с плавающей запятой, 64 однобитных предикатных регистра, 8 регистров перехода. 96 регистров из 128 видны в качестве «регистрового окна» и могут быть использованы при программной конвейеризации. Регистры перехода служат для предсказания адреса перехода и хранят адрес, используемый в косвенном переходе. Первые регистры содержат широко употребительные константы — целый 0, плавающие 0.0, плавающие 1.0.

Поддержка спекулятивных команд в архитектуре EPIC реализуется МП Itanium с помощью дополнительного 1-битового поля, называемого NaT. Все регистры МП имеют на самом деле длину 65 бит. Спекуляция состоит из двух частей: 1) команда, начинающая спекулятивную загрузку; 2) команда, делающая проверку на наличие исключений. Команда, начинающая загрузку, очищает бит NaT. В случае, если происходит интерференция обращений к памяти, генерируется исключения, проявляющееся в установке этого бита в 1. Команда, проверяющая исключения,

проверяет установлен ли бит NaT, и если он установлен, происходит вызов восстанавливающего кода, а бит очищается.

2.2.3.2. AMD Opteron

Второй по величине производитель микропроцессоров с архитектурой x86 — корпорация AMD — избрала альтернативный подход: добавила 32 разряда к уже имеющимся 32. В результате регистры стали иметь 64 разряда, появились команды манипуляции с 64-разрядными данными, и шина адреса увеличилась до 64 разрядов — появилась архитектура x86-64. Новые команды отличаются от команд процессоров x86 только наличием префикса, указывающего на их разрядность.

Кроме шестнадцати регистров общего назначения, имеются восемь 64-разрядных регистров для операций вещественной арифметики. Первые восемь регистров обозначаются названиями, отражающими их x86-происхождение: RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI. Так, восемь младших разрядов RAX фактически эквивалентны регистру A (аккумулятору) процессора i8080 и регистру AL i8086. Разряды 8—15 эквивалентны регистру AH i8086. Объединение этих двух полей представляет регистр AX i8086. Битовое поле 0—31 — полный эквивалент регистра EAX в 32-разрядных 80x86. Дополняют архитектуру процессора шестнадцать 128-разрядных регистров для хранения операндов SIMD-инструкций.

Обеспечена полная аппаратная поддержка выполнения инструкций x86-32 на уровне ядра. В отличие от процессора Itanium, здесь обеспечивается полноценная реализация 8-, 16- и 32-разрядных приложений без потери производительности. Таким

образом, на одном процессоре могут одновременно и независимо работать 16- и 32-разрядные приложения. Это делает переход пользователей на новую платформу безболезненным. Процессоры могут работать в двух режимах — Long и Legacy Mode. В первом кристалл будет работать как x86-64, а во втором — как x86-процессор, совместимый с 16- и 32-разрядными приложениями и поддерживающий расширение SSE.

Процессоры содержат интегрированный контроллер памяти, совместимый с технологией HyperTransport. Это позволяет напрямую работать с системной памятью, минуя системную шину и набор микросхем. Для возможности обращения к одному и тому же сегменту памяти в мультипроцессорных системах используется архитектура NUMA (Non-Uniform Memory Access). Каждому процессору отведен отдельный сегмент памяти, но при необходимости будет доступен и сегмент памяти другого процессора.

2.2.4. Коммуникационная подсистема

Производительность коммуникационной подсистемы оказывает большое влияние на общую производительность системы. Зачастую она становится тем самым узким местом, из-за которого она сильно падает. Поэтому очень важно тщательно спроектировать топологию коммуникаций и обеспечить их достаточную пропускную способность. Ниже рассмотрен некоторые распространённые топологии построения многопроцессорных систем, а так же некоторые современные коммуникационные технологии, в ним применяемые. Информация о конкретных топологиях взята из книги [60]

2.2.4.1. Существующие топологии

Простейшая топология - это ***линейный массив***, описываемый графом, последовательно соединяющим узлы (рис. 2.2.4.1.).

Рисунок 2.2.4.1. Топология в виде линейного массива

Однонаправленный граф такого вида очевидно представляет собой схему соединений в конвейере с числом стадий, равным числу узлов. Число дуг в таком графе равно числу узлов без единицы. При числе узлов *n* максимальное число транзитов для передачи данных от первого к последнему узлу равно *n* - 1. Как правило, при конвейерной обработке необходимость в транзитах отпадает.

Следующая топология - ***кольцо***, которое можно рассматривать как частный случай линейного массива, свернутого в цикл (рис. 2.2.4.2).

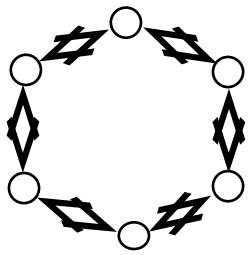

Рисунок 2.2.4.2. Топология в виде кольца.

Максимальное число дуг, которое надо пройти, чтобы от одного элемента добраться до любого другого равно $n/2$ при четном n и $(n-1)/2$ при нечетном n в кольце, если дуги двунаправлены.

Следующая классическая топология называется звездой (рис. 2.2.4.3.)

Казалось бы, по максимальному числу дуг (транзитов), которые необходимо пройти, чтобы передать данные от одного узла к другому, звезда наиболее оптимальна: вне зависимости от числа узлов число транзитов не превышает двух. Однако в этом случае требуется наличие в центре достаточно сложной коммутирующей схемы с $(n-1)$-м входом и $(n-1)$-м выходом, обеспечивающей переключение каналов.

Топология типа звезды обладает тем недостатком, что выход из строя центрального узла влечет за собой нарушение работоспособности всей системы.

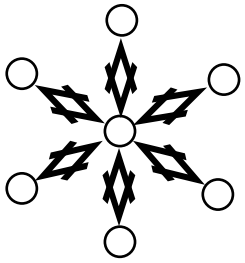

Рисунок 2.2.4.3. Топология звезда

В мультипроцессорных системах широко используется топология ***решетки*.** Решетки характеризуются своей размерностью.

Двухмерную решетку можно представить графом, изображенным на рис. 2.2.4.4.

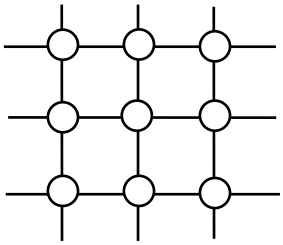

Рисунок 2.2.4.4. Топология решётки

Дуги соединяют между собой ближайших соседей. Если рассматривать многомерные решетки, то каждый узел можно помечать набором индексов. Как правило, ближайшими соседями к узлу $U_{i,j,\ldots,k}$ считаются узлы с наборами индексов, отличающихся от исходного набора на 1 только в одной позиции набора индексов. Например, соседом узла $U_{i,j,..,l,\ldots,k}$ будет узел $U_{i,j,\ldots,l+1\ldots,k}$.

Поскольку мы предполагаем всегда, что число узлов в рассматриваемых графах конечно, можно перенумеровать все узлы последовательно (без индексов). В этом случае формула определения ближайших соседей, естественно, оказывается более сложной.

Число узлов в k-мерном пространстве в прямоугольной решетке равно $N = n * n * \ldots * n$, где n — размерность решетки. Общее число связей (дуг) равно $k * N$ или в нашем случае $k * n^k$.

Топология двухмерных решеток с успехом используется во многих SIMD-архитектурах. Иногда их

называют решетками типа иллиак по имени классической вычислительной системы ILLIAC-IV.

Решетки удобно использовать для решения матричных задач и задач сортировки с использованием параллелизма. Известна оценка числа шагов параллельных вычислений для получения произведения матриц размерности $n \times n$, которое равно $0.35n$. Речь идет о двухмерной решетке с $n \times n$ компонентами. Задача сортировки при параллельном ее выполнении требует числа тактов, пропорционального n.

Следующий важный тип топологии называют ***n-кубом,*** или ***гиперкубом*** размерности (ранга) n. Число связей (дуг) в нем равно числу ребер гиперкуба, т.е. 2^n **.** Максимальный транзит равен n**.** Гиперкуб — весьма удачная топология со сравнительно приемлемыми значениями характеристик числа связей и «самого длинного» пути.

Обычно все типы топологий сравниваются с предельным типом – полносвязанным графом, в котором каждый узел соединен дугой с каждым другим. Число дуг в полносвязанном графе равно $N(N-1)/2$, где N - число узлов. Самый длинный путь в топологии «каждый с каждым» равен 1. С ростом числа узлов число дуг растет как N^2. Отношение числа узлов к числу дуг в полносвязанном графе равно $\frac{2}{N-1}$.

Для гиперкуба степени n числом дуг равно $n2^{n-1}$. Отношение числа узлов гиперкуба к числу дуг равно

$$\frac{2^n}{n2^{n-1}} = \frac{2}{n}, \text{ или } \frac{2}{\log_2 N}$$

Это отношение убывает с ростом *N* значительно медленнее чем $\frac{2}{N-1}$ и. преимущества, связанные с меньшим числом связей в гиперкубе по сравнению с полным графом, становятся более значительными. С точки зрения аппаратных затрат уменьшение общего числа связей в мультипроцессорных системах весьма желательно из соображений простоты реализации и надежности.

Длина максимального пути в гиперкубе размерности *n* равна *n* при этом растет относительно медленно со скоростью логарифма.

Большое семейство мультипроцессоров SIMD- так и MIMD-архитектуры имеют топологии гиперкуба, в частности PARSITEC, WARP, iPSC, CM и др.

Используется также топология, называемая ***циклами, соединенными в гиперкуб*** (рис. 2.2.4.5)**.**

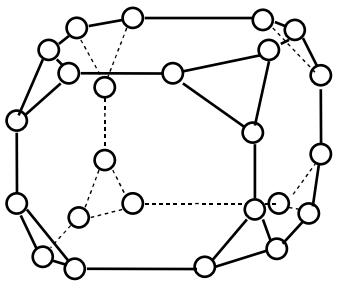

Рисунок 2.2.4.5. Топология гиперкубического соединения циклов

Число узлов в графе с такой топологией равно $n*2^n$. Строится он заменой каждого узла гиперкуба на кольцо, состоящее из n узлов. От каждого узла кольца отходит по одной связи по граням изначального гиперкуба.

Максимальный путь равен:

$n+n/2=1.5n$ для четных n, и

$n + (n+1)/2 = 1.5n + 0.5$ - для нечетных n.

Отметим важное свойство топологии гиперкубических циклов: число ребер, исходящих из одного узла, всегда равно трем вне зависимости от размерности самого гиперкуба.

Топология ***пирамиды***. Такая топология используется в некоторых мультипроцессорных системах специального назначения, главным образом для анализа изображений.

Пирамида представляет собой иерархическую многоэтажную структуру, каждый этаж которой представляет собой решетку. В частном случае четырехарной пирамиды каждый узел промежуточного этажа связан с четырьмя узлами нижестоящего этажа и с одним узлом вышестоящего узла. На вершине пирамиды располагается только один корневой узел. На рис. 2.2.4.6. представлена четырехарная пирамида.

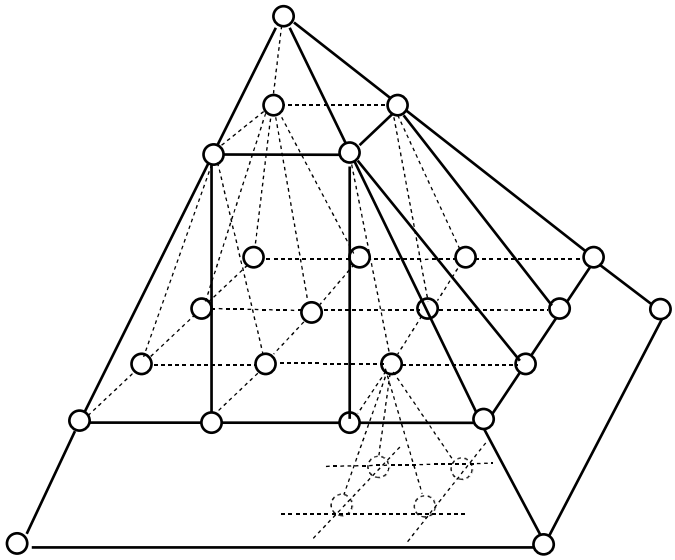

Рисунок 2.2.4.6. Топология пирамиды

Тем самым, четырехарная пирамида с числом узлов в основании $P = k^2$ имеет высоту $log_4 P$. Общее число узлов в ней равно $(4P - 1)/3$**.** Каждый внутренний узел, не принадлежащий основанию и вершине, связан с девятью другими: связь с верхним этажом, четыре связи на своем этаже и четыре связи с нижним этажом.

Максимальный транзит для пирамиды можно подсчитать, однако характер ее функционирования не требует обычно передачи данных, минуя несколько этажей. Данные, "собранные" на каком либо этаже, передаются только на соседний этаж. Обработка при этом ведется параллельно на всех этажах.

2.3. Дисковая подсистема

Дисковая подсистема — с точки зрения кластеров — система хранения данных, отвечающая ряду требований:

- Надёжность —
  - Для пользователя неисправность была максимально незаметна
  - Автоматическое устранение неисправностей логического уровня
  - Защищённость от несанкционированного доступа
  - Использование надёжной ФС (поддержка журналирования, разграничение прав пользователей)
- Производительность —
  - Возможность использования нескольких носителей одновременно для чтения/записи
  - Возможность параллельных операций чтения/записи
- Достаточный объём —
- Ёмкость системы должна быть достаточной для хранения результатов вычислений всех пользователей системы

Далее рассмотрены две основные технологии, применяемые в дисковых хранилищах: SCSI и RAID. Первая служит для подключения устройств хранения данных, вторая — для повышения степени надёжности хранения информации и ускорения скорости доступа к ней.

2.3.1. SCSI

SCSI (*Small Computer Systems Interface*) — интерфейс, разработанный для объединения на одной шине различных по своему назначению устройств, таких как жёсткие диски, накопители на магнитооптических дисках, приводы CD, DVD, стриммеры, сканеры, принтеры и т. д. Раньше имел неофициальное название Shugart Computer Systems Interface в честь создателя Алана Ф. Шугарта

Любое SCSI-устройство должно поддерживать обязательные команды общего набора и своего класса, чем обеспечивается высокий уровень совместимости. Теоретически возможен выпуск устройства любого типа на шине SCSI.

После стандартизации в 1986 году, SCSI начал широко применяться в компьютерах Apple Macintosh, Sun Microsystems. В компьютерах совместимых с IBM PC SCSI не пользуется такой популярностью в связи со своей сложностью и сравнительно высокой стоимостью.

В настоящее время SCSI широко применяется на серверах, высокопроизводительных рабочих станциях; RAID-массивы на серверах часто строятся на жёстких дисках со SCSI-интерфейсом (хотя в настоящее время на серверах нижнего ценового диапазона всё чаще применяются RAID-массивы на основе SATA).

Существует три стандарта SCSI (SE — single-ended, LVD — интерфейс дифференциальной шины низкого напряжения, HVD — интерфейс дифференциальной шины высокого напряжения), каждый из которых имеет множество дополнительных и необязательных возможностей. Некоторые комбинации возможностей имеют собственные наименования.

Пропускная способность и ширина интерфейса SCSI менялась со временем. Первые реализации интерфейса предусматривали ширину канала 8 бит и пропускную способность 5 MiB/s. На данный момент интерфейс SCSI предоставляет канал шириной 16 бит и пропускной способностью 320 MiB/s. При этом по нему возможно подключить до 16 устройств.

2.3.2. RAID

RAID (redundant array of independent/inexpensive disks) — дисковый массив независимых дисков. Служат для повышения надёжности хранения данных и/или для повышения скорости чтения/записи информации. Беркли представил следующие уровни RAID, которые были приняты как стандарт де-факто:

- **RAID 0** представлен как неотказоустойчивый дисковый массив
- **RAID 1** определён как зеркальный дисковый массив
- **RAID 2** зарезервирован для массивов, которые применяют код Хемминга
- **RAID 3, 4, 5, 6** используют чётность для защиты данных от одиночных неисправностей

Кроме того, существуют комбинации этих уровней (RAID 0+1, RAID 10, RAID 50, RAID 100 и другие), которые объединя.т в себе отдельные их достоинства.

2.3.2.1. RAID уровня 0

Технология RAID 0 также известна как распределение данных (data striping). С применение этой технологии, информация разбивается на куски

(фиксированные объемы данных, обычно именуемых блоками), и эти куски записываются на диски и считываются с них параллельно. С точки зрения производительности это означает два основных преимущества:

- повышается пропускная способность последовательного ввода/вывода за счет одновременной загрузки нескольких интерфейсов
- снижается латентность случайного доступа

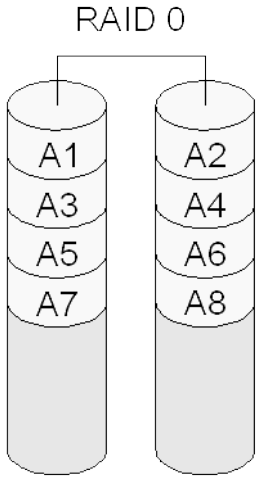

Рисунок 2.3.2.1
Последовательность записи блоков A_i на диски при использовании RAID 0

Недостаток: уровень RAID 0 предназначен исключительно для повышения производительности, и не обеспечивает избыточности данных. Поэтому любые дисковые сбои приводят к потере всей информации на

носителях (восстановить которую при отсутствии резервной копии очень трудно).

2.3.2.2. RAID уровня 1

Технология RAID 1 также известна как зеркалирование (disk mirroring). В этом случае копии каждого куска информации хранятся на отдельном диске; или каждый (используемый) диск имеет «двойника», который хранит точную копию этого диска. Если происходит сбой одного из основных дисков, этот замещается своим «двойником».

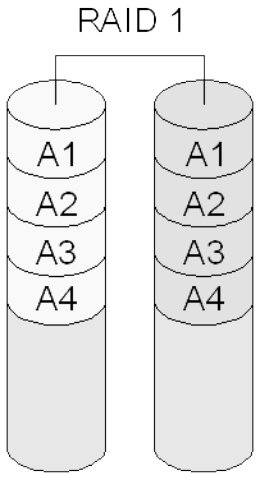

Рисунок 2.3.2.2.
Последовательность записи блоков A_i на диски при использовании RAID 1

Время записи может оказаться несколько больше, чем для одного диска. Время чтения может быть увеличено теоретически двукратно.

Уровень RAID 1 хорошо подходит для приложений, которые требуют высокой надежности и низкой латентности при чтении. RAID 1 обеспечивает избыточность хранения информации, но в любом случае следует поддерживать резервную копию данных, т. к. это единственный способ восстановить случайно удаленные файлы или директории.

2.3.2.3. RAID уровней 2 и 3

Технология RAID уровней 2 и 3 предусматривает параллельную работу всех дисков. Эта архитектура требует хранения битов четности для каждого элемента информации, распределяемого по дискам. Отличие RAID 3 от RAID 2 состоит только в том, что RAID 2 использует для хранения битов четности несколько дисков, тогда как RAID 3 использует только один. RAID 2 используется крайне редко.

Если происходит сбой одного диска с данными, то система может восстановить его содержимое по содержимому остальных дисков с данными и диска с информацией четности.

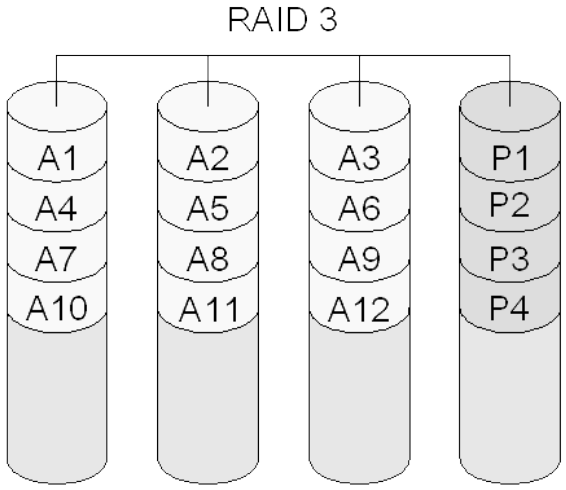

Рисунок 2.3.2.3. Последовательность записи блоков A_i и контроля чётности P_i на диски при использовании RAID 3

Производительность в этом случае очень велика для больших объемов информации, но может быть весьма скромной для малых объемов, поскольку невозможно перекрывающееся чтение нескольких небольших сегментов информации.

2.3.2.4. RAID уровней 4 и 5

RAID 4 исправляет некоторые недостатки технологии RAID 3 за счет использования больших сегментов информации, распределяемых по всем дискам, за исключением диска с информацией четности. При этом для небольших объемов информации используется только диск, на котором находится нужная информация. Это означает, что возможно одновременное исполнение нескольких

запросов на чтение. Однако запросы на запись порождают блокировки при записи информации четности. RAID 4 используется крайне редко.

Технология RAID 5 очень похожа на RAID 4, но устраняет связанные с ней блокировки. Различие состоит в том, что информация четности распределяется по всем дискам массива. В данном случае возможны как одновременные операции чтения, так и записи. Данная технология хорошо подходит для приложений, которые работают с небольшими объемами данных, например, для систем обработки транзакций.

2.3.2.5. RAID 6

RAID 6 — это отказоустойчивый массив независимых дисков с распределением контрольных сумм, вычисленных двумя независимыми способами. Этот уровень во многом схож с RAID 5. Только в нем используется не одна, а две независимые схемы контроля четности, что позволяет сохранять работоспособность системы при одновременном выходе из строя двух накопителей. Для вычисления контрольных сумм в RAID 6 используется алгоритм, построенный на основе кода Рида-Соломона (Reed-Solomon).

Этот уровень имеет очень высокую отказоустойчивость, большую скорость считывания (данные хранятся блоками, нет выделенных дисков для хранения контрольных сумм). В то же время из-за большого объема контрольной информации RAID 6 имеет низкую скорость записи. Он очень сложен в реализации, характеризуется низким коэффициентом использования дискового пространства: для массива из

пяти дисков он составляет всего 60%, но с ростом числа дисков ситуация исправляется.

RAID 6 по многим характеристикам проигрывает другим уровням, поэтому на сегодня не получил коммерческого применения.

3. Пользовательская среда кластера

Обычно, для взаимодействия с пользователями выделяют специальный компьютер, который чаще всего имеет ту же архитектуру процессора, что и узел кластера, и чаще всего работает под управлением той же операционной системы, что и узел кластера. Эта машина во введении была названа как интерфейсная машина (front-end машина). Она предоставляет интерфейс пользователю вычислительной системы к самой вычислительной системе. Как правило здесь происходит редактирование файлов; запись исходных данных для обсчёта на вычислительную систему, сохранение результатов обработки данных многопроцессорной системы. Довольно часто на интерфейсной машине происходит компиляция пользовательских программ; именно вследствие компиляции архитектура интерфейсной машины обычно совпадает с архитектурой узла кластера. В качестве интерфейса к вычислительной системе может быть дополнительно представлен web-сервер, который предоставляет некий web-интерфейс для мониторинга и управления очередью пользовательских заданий. Однако чаще всего постановка и удаление задач пользователя в очередь и из очереди происходит на интерфейсной машине.

3.1. Удалённый доступ к вычислительной системе на основе SSH протокола.

Получить доступ к удалённой вычислительной системе можно довольно большим количеством способов, но из всего многообразия методов можно выделить несколько основных и наиболее часто используемых. Это доступ по SSH, по FTP или через Web-интерфейс. Не все из них (например, FTP) может разрешить администратор вычислительной системы, могут также быть предусмотрены альтернативные способы, но доступ по SSH есть почти всегда. К его рассмотрению мы и переходим.

SSH (Secure Shell) — это программа для входа в другие компьютеры доступные по сети, для выполнения команд или программ на удаленных компьютерах и для передачи файлов с одного компьютера на другой. Она обеспечивает проверку подлинности и безопасности соединений по незащищенным каналам. Используется как замена rlogin, rsh, и rcp. Дополнительно, ssh обеспечивает безопасность соединений с X-сервером и безопасное перенаправление иных необходимых пользователю TCP соединений.

Традиционные BSD 'r'-команды (rsh, rlogin, rcp) уязвимы для различных видов хакерских атак. Кроме того, если кто-то имеет «привелигерованный» (root) доступ к машинам сети или возможность физического подключения к сетевым коммуникациям, то это позволит собирать весь проходящий трафик, как входной, так и выходной включая пароли, поскольку TCP/IP является «чистым» протоколом (ssh никогда не посылает пароли в чистом виде). X Window System

имеет тоже достаточное количество «узких» мест в плане безопасности. Но с помощью ssh можно создавать безопасные удаленные сеансы X-сервера.

В SSH-протоколе декларируется, что данные будут зашифрованы. Однако, конкретный способ шифрования Намерено не указывается. Предполагается, что в реализации будет предложен один или несколько способов шифрования данных. В текущий момент наиболее популярно шифрование на основе открытого ключа DSA и RSA. Ещё довольно распространена поддержка протокола kerberos.

3.1.1. Клиенты SSH

У пользователя, зарегистрированного в вычислительном центре, на начальном этапе возникает вопрос: какой ssh клиент использовать для доступа к интерфейсной машине многопроцессорного комплекса в его операционной системе использовать. Замечательно, что в текущий момент существует огромное число различных ssh-клиентов практически под все существующие операционные системы.

3.1.1.1. Клиент в пакете OpenSSH.

В большинстве UNIX-подобных операционных системах в качестве ssh клиента принято использовать пакет OpenSSH (http://www.openssh.org/). OpenSSH разрабатывался как часть проекта OpenBSD, однако при его разработке делался упор на переносимость и в текущий момент его реализация есть практически на всех UNIX-подобных операционных системах, включая Cygwin под Windows, для довольно большого числа аппаратных платформ. Разработчики распространяют данный пакет бесплатно, а также предоставляют его исходный код.

В OpenSSH авторизация и передача данных устроена на основе пар открытый, закрытый ключ. В момент установки SSH сервера для машины генерируется пара открытый, закрытый ключ. Затем, при первом обращении к серверу со стороны удалённой машины (клиента) открытый ключ копируется и сохраняется (обычно в $HOME/.ssh/known_hosts). Далее, в случае не соответствия пар ключей на клиенте и сервере будет выдано предупреждение и отказано в соединении. Данные при передаче шифруются в соответствии с открытым ключём и в таком виде передаются на сервер. Предполагается, что вычисление закрытого ключа по открытому является чрезвычайно вычислительно сложной задачей, а для расшифровки данных требуется знание именно закрытого ключа.

Такой способ шифрования предполагает, что передача открытого ключа со стороны сервера происходит весьма редко. В момент передачи публичного ключа злоумышленник может подсмотреть ключ и дальше ему нет необходимости вычислять закрытый ключ для расшифровки данных.

OpenSSH позволяет организовать авторизацию без указания пользователем пароля, на основе открытого и приватного ключей. Пользователь сперва локально создаёт пару публичный ключ и приватный ключ, (для этого используется программа ssh-keygen) затем указывая свой пароль заходит на удалённую машину, записывает публичный ключ в специальный файл (по умолчанию $HOME/.ssh/authorized_keys). После выполнения этой процедуры с локальной машины на удалённую можно заходить по SSH протоколу не указывая пароль. Такой способ авторизации позволяет значительно облегчить работу по написанию специальных программ и скриптов для не

интерактивной удалённой работы на машине через SSH протокол.

Терминальный доступ к удалённой машине, а также доступ на выполнение удалённой команды в OpenSSH предоставляется с помощью программы ssh. Для работы необходимо указать имя или IP-адрес сервера, к которому производится подключение, а также имя учётной записи на удалённой машине (при опредлённых настройках, это может делаться автоматически). Затем, ещё до установления соединения с удалённой машиной клиент пытается сперва выяснить есть ли в наличии соответствующий публичный ключ для авторизации на удалённой машине, если есть, то производится авторизация на основе ключа, в случае провала авторизации или отсутствия ключа или не пустой строчки с паролем в ключе клиент запрашивает пароль у пользователя. Ниже приведён пример команды для установки соединения.

```
$ssh salnikov@my-cluster.ru 'omp_submit -w 04:30:00 my_task'
```

Если идентификация пользователя принята сервером, то сервер либо выполняет указанную команду, либо регистрирует пользователя в машине и дает ему нормальную оболочку на удаленной машине. Вся связь с удаленной командой или оболочкой будет автоматически зашифрована. Если был назначен псевдо-терминал (нормальный сеанс регистрации в системе), то пользователь может использовать управляющие последовательности символов, указанные ниже. Если псевдо-терминал не был назначен, то сеанс с удалённой машиной происходит

без реакции на управляющие последовательности символов и может быть использован для передачи бинарных файлов. Сеанс прерывается, когда команда или оболочка на удаленной машине завершает работу и все **x11** и TCP/IP соединения завершены. Статус выхода исполняемой удалённой команды возвращается как статус выхода для локально запущенной программы **ssh**.

Довольно важной функциональностью, которую предоставляет OpenSSH, является возможность туннелирования TCP соединений. Наиболее востребовано туннелирование протокола общения с X сервером. Если со стороны OpenSSH сервера туннелирование разрешено, то клиент может указать опцию -X в результате на стороне сервера будет выставлена переменная окружения DISPLAY, а весь графический вывод от приложений запущенных на удалённой машине будет перенаправлен через ssh туннель на локальную машину, где должен быть запущен X сервер для отображения получаемых графических данных.

При помощи кючей -L и -R можно организовывать туннели на определённые TCP порты на локальной и удалённой машине. По ключу -L происходит следующее: все пакеты, которые должны быть отправлены через соединение с определённым TCP портом на локальной машине перенаправляются на определённый порт на удалённой машине. Ниже приведён пример обращения на ftp сервер третьей машины через вторую машину.

```
ant7# ssh -L 21:ftp.debian.org:21 hill.seminar
lftp localhost
```

По ключу -R при соединении с определённым портом на удалённой машине все данные перенаправляются на определённый порт третьей машины таким образом, как будто весь трафик исходил от удалённой машины, когда в действительности трафик инициировался локальной машиной. Такое действие требует, чтобы на удалённую машину пользователь заходил как пользователь обладающий достаточными правами (в случае UNIX как root).

Поскольку SSH предоставляет терминальный доступ на удалённую машину на удалённую машину в том числе передаются и управляющие последовательности символов. Например при нажатии клавиш <ctrl> и <z> (обозначается как ^z) одновременно во всех Unix командных интерпретаторах происходит преостановка запущенного процесса. В случае с SSH будет приостановлен процесс на удалённой машине, а не сама программа ssh на локальной машине. Для управления самой программой ssh с локальной стороны вводится специальный набор управляющих символов, которые не передаются на удалённую машину, а вместо этого меняют состояние самой программы ssh, запущенной на локальной машине. Далее приведён список основных управляющих последовательностей символов для OpenSSH клиента.

Управляющая последовательность	**Описание действия**
~.	Отсоединяет OpenSSH клиента от удалённой машины. Программы с удалённой стороны теряют стандартный поток вывода.
~^Z	Приостанавливает ssh. Действие аналогичное нажатию ^z в командном

Управляющая последовательность	Описание действия
	интерпретаторе.
~#	Показывает список затунелированных соединений
~B	Посылает сообщение, чтобы OpenSSH сервер завершил соединение со своей стороны, а не со стороны клиента.
~C	Исполняет команду по управлению самой программой ssh
~?	Показывает список управляющих последовательностей.

3.1.1.2. Клиент PuTTY

Для Windows в текущий момент наиболее популярен пакет PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/). Он предоставляет собой набор программ для захода на удалённую машину по SSH протоколу. Поддерживает туннелирование, в первую очередь протокола X сервера. Поддерживает авторизацию по ключам. В нём имеется как консольная программа для захода на удалённую машину, так и графический интерфейс. Сочетание PuTTY и X сервера в Cygwin значительно облегчает взаимодействие с удалённой Unix машиной с машины под управлением Windows. Также как и OpenSSH предоставляются исходные коды от всех утилит входящих в PuTTY и предоставляется возможность с сайта выкачать уже откомпилированную бинарную версию под разные версии Windows, в том числе 32-х битные и 64-х битные.

3.1.1.3. Прочие клиенты

На ряду со свободным программным кодом существуют коммерческие версии. Например существует клиент SecureCRT. Существуют клиенты SSH для мобильных устройств Например pocketPuTTY (http://www.pocketputty.net/). Для Palm OS существует реализация pssh (http://www.sealiesoftware.com/pssh/). Есть также клиент и для Symbian OS — s2PuTTY (http://s2putty.sourceforge.net/).

3.1.2. Безопасное копирование файлов SCP

Применяя те же методы шифрования, что и для терминального доступа можно передавать файлы в сети безопасным с точки зрения несанкционированного чтения способом. Для этой цели в SSH есть механизм **SCP** — **Secure Copy**. **SCP** обеспечивает защищённую передачу файлов между локальной и удалённой машиной с использованием протокола ssh. Протокол SCP в основном идентичен протоколу BSD rcp. Но, в отличие от rcp, информация шифруется во время передачи. Протокол SCP реализует только передачу файлов. В OpenSSH это реализовано путём создания со стороны программы scp SSH-соединения с SSH сервером (sshd) и исполнения на удалённой машине программы scp. При этом одна копия читает файл и шифрует его, а другая пишет и расшифровывает. Авторизация идёт через общение с OpenSSH сервером (sshd).

Для передачи данных на удалённую машину клиент передаёт серверу файлы для передачи, опционально включая их базовые атрибуты (разрешения, временные пометки). Для скачивания, клиент посылает запрос на файлы или каталоги для скачивания. При скачивании каталога сервер передаёт клиенту его вместе с

поддиректориями и файлами. Таким образом, скачивание файлов контролируется сервером.

К сожалению, SCP становится опасным протоколом, если в начальный момент обмена ключей при установлении первого соединения злоумышленник смог сделать так, что вместо удалённой машины локальная машина в сети видит машину злоумышленника. В данной ситуации злоумышленник может подсунуть вместо того файла, который хочет получить пользователь локальный машины тот файл который хочет злоумышленник или сохранить файл пользователя вместо удалённой машины на свою собственную. Кроме порчи информации это опасно ещё подсовыванием троянских программ. Выход из данной ситуации носить блокнотик с ключами всех удалённых машин, к которым предполагается подключение.

Ниже в таблице приведён список ключей специфичных для программы scp из пакета OpenSSH:

-p	Сохраняет время модификации, время доступа и режимы как у файла-оригинала.
-r	Рекурсивное копирование дерева каталогов с подкаталогами.
-B	Пакетный режим. Авторизация исключительно на основе ключей без указания пароля или ключевой фразы. предполагается, что scp в данном случае ничего не спрашивает у пользователя.
-q	Отключает индикатор прогресса.

3.1.3. Продвинутый интерфейс для манипуляции с файлами — SFTP

SCP — довольно простой протокол для манипуляции с файлами. Бывает, что пользователю

необходим доступ к удалённой файловой системе по похожему на FTP протоколу, но через защищённый канал. Для этих целей в SSH-2 присутствует защищённый протокол передачи данных — SFTP (Secure File Transfer Protocol). Изначально SFTP задумывался просто как FTP протокол запущенный поверх SSH. Однако фактически SFTP получился новым протоколом отличающимся по своим возможностям от FTP.

По сравнению с протоколом SCP, который позволяет совершать только передачу данных, SFTP поддерживает набор операций над удалёнными файлами — скорее, он ближе к протоколу удалённой файловой системы. По сравнению с клиентом SCP, клиент SFTP позволяет возобновлять прерванные передачи данных, получать список файлов в директории и удаление удалённых файлов.

В пакете OpenSSH программа sftp**.** Это интерактивная программа для перемещения файлов, похожая на ftp. Программа sftp позволяет использовать много возможностей ssh, таких как аутентификация с открытым ключем и сжатие.

После установления соединения и авторизации запускается интерактивный режим в котором можно указывать команды для манипуляции с файлами и каталогами. Команды являются нечувствительными к регистру букв, которыми они набраны, и имена путей, если они содержат пробелы, могут быть заключены в кавычки.

Далее в таблице приведены некоторые команды sftp.

Команда	Описание
cd	Изменяет текущий каталог на удалённой машине на тот, который указан как параметр команде cd.
lcd	Тоже самое, что cd, но на локальной машине.
chgrp	Изменяет группу для файла или каталога на удалённой машине.
chmod	Меняет права доступа к файлу или каталогу на удалённой машине.
chown	Меняет владельца у файла или каталога на удалённой машине.
help,?	Справка по командам
get	Получает файл или каталог с удалённой машины и сохраняет его на локальной машине. Если имя на локальной машине не указано, то назначается имя аналогичное имени на удаленной машине. Если указан флаг **-P**, тогда будут скопированы полные права доступа и время доступа к файлам.
put	Записывает локальный файл или каталог на удалённую машину. Действует аналогично get, только в другую сторону.
lls	Отображает содержимое каталога указанного в параметрах на локальной машине или текущего каталога на локальной машине.
ls	Тоже самое, что и lls только на удалённой машине.
lmkdir	Создание каталога на локальной машине

Команда	Описание
mkdir	Создание каталога на удалённой машине
ln,symlink	Создаёт символическую ссылку
lpwd,pwd	показывает текущий каталог
quit,exit	выход из sftp
rename	переименовывает файл или каталог.
rm	удаляет файл на удалённой машине
!	исполняет команду указанную в параметрах командным интерпретатором на удалённой машине

3.2. Постановка задач в систему очередей

Здесь, мы подходим к самому главному, к тому ради чего создаётся вычислительный кластер. Кластер создаётся ради того, чтобы пользователи могли запускать на нём свои задачи. Как правило, для запуска задач на вычислительной системе используется специализированное программное обеспечение, которое стремиться «справедливо» распределить ресурсы вычислительной системы между её пользователями. Соответствующее программное средство в дальнейшем будем называть планировщиком задач пользователя. Вторая задача, которую решает планировщик — это обеспечение наиболее оптимального использования ресурсов вычислительной системы задачами пользователей.

К настоящему моменту времени различными компаниями и сообществами разработчиков создано

довольно большое число систем управления задачами пользователей. Существуют как свободно распространяемые системы, такие как Sun Grid Engine, так и коммерческие, как Maui, OpenPBS, LoadLeveler.

Обычно механизм взаимодействия пользователя с вычислительной системой выглядит следующим образом.

1. Пользователь заходит на интерфейсную машину с которой доступен набор клиентских программ по управлению системой очередей. Он пользуется соответствующим набором программ и передаёт тем самым свою команду на сервер управляющий очередями.
2. Команда, после своего поступления на сервер очередей и обрабатывается им. В зависимости от ситуации сервер очередей создаёт свою внутреннюю команду, которую передаёт на один из серверов запущенных на одном из вычислительных узлов кластера.
3. Работа сервера на узле заключается в слежении за пользовательскими процессами на узле кластера, а так же в создании процесса по команде поступившей от сервера очередей и уничтожении процесса соответственно. В функции сервера, запущенного на узле кластера, входит функция информирования сервера очередей о состоянии куска пользовательской задачи предназначенной этому узлу кластера. Он должен информировать о невозможности запустить пользовательский процесс о досрочном

завершении пользовательского процесса и т. п..

В целом, любая система управления очередями пользователю вычислительной системы интересна прежде всего командами постановки задания в очередь, просмотра состояния очереди и удаления задачи из очереди. Рассмотрим эти команды на примере системы *Portable Batch System — PBS.*

3.2.1. Система очередей PBS

Система управления заданиями (Portable Batch System - PBS) - это пакет программ по управлению системными компьютерными ресурсами и группами заданий. PBS принимает группы заданий (атрибуты контролируются при помощи shell скриптов), сохраняет и защищает задание до запуска, запускает задание, возвращает результаты работы приложения пользователю. Пакет PBS может быть инсталлирован и сконфигурирован для работы как на одной многопроцессорной вычислительной системе, так и на нескольких. В PBS допускается разнообразное объединение различных вариантов конфигурации так, что пользователь может ставить своё задание в одну из очередей, где кажная очередь сопоставлена своей многопроцессорной системе. В состав PBS входят следующие основные компоненты: набор программ реализующих передачу команд серверу заданий и исполнительному серверу (часть работающая на узле кластера), реализация алгоритмов планирования заданий.

3.2.1.1. Команды PBS

PBS поддерживает два интерфейса вызова команд: интерфейс командной строки и графический интерфейс. Существуют три класса команд:

- пользовательские команды: qsub, qstat, qdel, qselect, qrerun, qorder, qmove, qhold, qalter, qmsg, qrls
- команды оператора: qenable, qdisable, qrun, qstart, qstop, qterm
- команды администратора: qmgr, pbsnodes

Более подробно рассмотрим некоторые из команд. Для добавления пользовательской задачи в очередь используется команда **qsub**. Она требует создания специального текстового файла, где указаны параметры запускаемой пользователем задачи. Простейший пример такого текстового файла приведён ниже.

```
#!/bin/sh

#PBS -l walltime=1:00:00
#PBS -l mem=400mb
#PBS -l ncpus=4

./my_work
```

Здесь ключём -l указывается список ограничений на всё множество ресурсов многопроцессорной системы, которое необходимо пользователю для

исполнения своей задачи. В примере указаны ограничения соответственно на время исполнения задачи на многопроцессорной системе, объём памяти и число процессоров, которое необходимо задаче пользователя для своего исполнения. Далее в таблице приведён список ограничений на ресурсы которые можно указывать в PBS.

ресурс	**расшифровка**
arch	Тип архитектуры необходимый задаче пользователя. Задача будет поставлена в очередь на ту многопроцессорную системы где спецификация архитектур совпадает.
cput	Общий объём процессорного времени необходимый задаче.
file	Максимальный объём дискового пространства необходимый задаче для своей работы.
mem	Суммарный объём памяти необходимый задаче.
ncpus	Число процессоров, которое необходимо задаче для своего исполнения.
nice	UNIX приоритет создаваемых задачей процессов.
nodes	Число и тип узлов кластера необходимых задаче.
pcput	Максимальный объём процессорного времени затрачиваемый каждым процессом задачи.
pmem	Ограничение на объём оперативной памяти для каждого процесса задачи.

ресурс	расшифровка
pvmem	Ограничение на объём виртуальной памяти для каждого процесса задачи.
resc	Логическое выражение, которое задаёт ограничение на ресурсы. В том смысле насколько важно невыполнение одного или нескольких ограничений.
software	Требование на наличие программного обеспечения. Указанное программное обеспечение необходимо для возможности исполнения задачи пользователя.
vmem	Общий объём виртуальной памяти необходимый задаче.
walltime	Отрезок астрономического времени в течении которого задача может быть в состоянии исполняющейся на многопроцессорной системе.

Кроме ограничений на ресурсы у команды **qsub** есть свой набор параметров. Укажем наиболее значимые из них в следующей таблице.

опция	расшифровка
-c временной_интервал	Указывает через какой интервал времени для задачи пользователя будет устанавливаться контрольная точка. В случае сбоя задачу можно будет продолжить с ближайшей контрольной точки.
-e путь	Указывает каталог, который будет выставлен как

опция	расшифровка
	текущий каталог для задачи. В него будут складываться файлы со стандартным входом, выходом и ошибкой.
-h	приостанавливает задачу сразу после запуска
-I	указывает, что задача интерактивная. То-есть её ввод и вывод будут с и на терминал пользователя. Однако запуск будет произведён в тот момент, когда задача дождётся своей очереди.
-j	объединяет поток вывода и поток ошибок в один поток
-l список_ресурсов	Ограничения на ресурсы приведённые в предыдущей таблице.
-M список_email	Список почтовых адресов на которые будет пересылаться информация о изменении в состоянии задачи пользователя.
-N имя_задачи	Задаёт имя задачи. В дальнейшем вся манипуляция с задачей со стороны пользователя будет производится по этому имени. Если имя не указано система очередей сама назначит имя задаче.

опция	расшифровка
-o путь_до_файла	Задаёт путь до файла куда будут перенаправлены стандартный поток вывода и поток ошибок.
-p приоритет	Указывает пользовательский приоритет для задачи в системе очередей.
-q имя_очереди	Указывает в какую очередь следует помещать задачу пользователя. Так можно указывать различные вычислительные системы.
-S путь_до_шела	Указывает из под какого командного интерпретатора следует запускать процессы задачи пользователя.
-v список_переменных	Список экспортируемых процессу задачи пользователя переменных (-V — экспортировать все переменные.)

Указанные параметры и список ресурсов в целом схожи для всех планировщиков задач пользователя. Отличия могут быть в названиях параметров, и некоторых других деталей. Например часть параметров может отсутствовать и наоборот присутствовать что-то уникальное.

Следующий важный момент просмотр состояния очереди. В PBS это делается при помощи команды **qstat**. Команда **qstat** распечатывает состояние очереди в стандартный поток вывода. Пример вывода команды приведён ниже.

```
% qstat
Job id    Name         User         Time Use S Queue
--------- ---------- ---------- -------- - -----
16.south  aims14      james             0 H workq
18.south  aims14      james             0 W workq
26.south  airfoil     barry       00:21:03 R workq
27.south  airfoil     barry       21:09:12 R workq
28.south  subrun      james             0 Q workq
29.south  tns3d       susan             0 Q workq
30.south  airfoil     barry             0 Q workq
31.south  seq_35_3    bayuca            0 Q workq
```

Здесь можно видеть несколько полей. Поле «Job id» задаёт идентификатор задачи пользователя в очереди. Поле «Name» указывает имя задачи пользователя. Поле «User» указывает от имени какого пользователя запущена задача. «Time Use» показывает как долго исполняется задача. «S» обозначает статус задачи в очереди (статус будет обсуждён позднее). Поле «Queue» показывает имя очереди. Указывая различные опции команде **qstat** можно добиться выдачи более детальной информации по состоянию очереди.

Задача пользователя может получить одно из возможных значений статуса:

- **E** — задача находится в состоянии, когда она получила ресурсы вычислительной системы в своё распоряжение, но ещё не запущена.
- **H** — задача приостановлена.
- **Q** — задача принята в очередь и в соответствии с правилами перемещения

задач в очереди ожидает освобождения ресурсов для исполнения себя.

- **R** — задача запущена.
- **T** — задача находится в состоянии миграции с одной группы процессоров на другую.
- **W** — задача ожидает когда наступит указанный в ограничениях момент времени запуска.
- **S** — задача присутствует в очереди, но не учитывается при выборе очередной исполняемой задачи.

В целом для разных планировщиков выводы от соответствующих команд просмотра состояния очередей схожи и статусы задач пользователя в очереди схожи. Ниже приведён вывод команды просмотра состояния очереди для машины regatta.

```
bash-2.05a$ llq

Id Owner Submitted ST Class Time_Limit Time_Left CPU

--------------- -------- --- - ----- - ----- -------- -------- --

regatta.34705.0 kovalev  Jun 5 17:44 R long_ 24:00:00 20:27:10  8

regatta.34707.0 borisova Jun 6 10:38 R long_ 20:00:00 13:03:55  8
```

По имени задачи в очереди, или по её идентификатору с помощью команды **qdel** можно удалить задачу из очереди. Есть также команды: **qhold**, которая приостанавливает задание в очереди; **qmove**, которая перемещает задание из одной очереди в другую.

С точностью до названий в разных планировщиках задач пользователя команды в целом совпадают. Есть отдельные нюансы. Например не все системы могут поддерживать миграцию задач с одной вычислительной системы на другую и из одной очереди в другую.

3.2.2. Политика использования ресурсов многопроцессорной вычислительной системы

На вычислительной системе перед пользователем встаёт проблема, выбора объёма ресурсов и запрашиваемого интервала времени для исполнения своей задачи. Он может заметить, что время, проведённое в очереди, нелинейным образом зависит от объёма запрашиваемых ресурсов. Это связано с тем, что администраторы вычислительной системы проводят определённую политику по отношению к пользователям и задачам пользователей.

Как правило, задача проводит в очереди тем больше времени, чем на большее астрономическое время предполагается занять многопроцессорную систему. Короткие по времени задачи обычно легко проскакивают через очередь, несмотря даже на то, что они могут требовать большого числа процессоров. В то же время администраторы вычислительной системы не очень любят отдавать всё множество процессоров какой-то одной задаче по причине больших накладных расходов на вытеснение задач с большого числа процессоров. Кроме того, если задача заняла всё множество процессоров, будет невозможно запускать маленькие по числу процессоров и по времени отладочные задачи, что приведёт к серьёзному недовольству среди пользователей.

Проблема выбора правильной политики использования вычислительной системы является

достаточно сложной и многосторонней задачей. Частично она решается при помощи программного обеспечения планировщиков задач пользователя, однако в основном это забота администратора вычислительной системы.

3.3. Работа с консолью и командный интерпретатор в UNIX

Основной интерфейс с многопроцессорными вычислительными системами осуществляется через виртуальный терминал (консоль) и командный интерпретатор. Основная идея заключается в том, что пользователь как-бы получает себе экран монитора и клавиатуру на интерфейсной машине вычислительной системы в то время как физически и клавиатура, и монитор принадлежат той машине, за которой сидит пользователь. Предполагается, что соответствующие виртуальная клавиатура и виртуальный дисплей подключены через виртуальную среду передачи символьных данных (терминальную линию) к интерфейсной машине вычислительной системы. Терминальная линия может быть реализована как канал связи в сети, например через SSH соединение, а может быть реализован как переписывание кусочка оперативной памяти ядром операционной системы в случае если терминал непосредственно подключён к интерфейсной машине. Предполагается, что терминальная линия может реагировать на специальные последовательности символов (Escape — последовательности). Например последовательность символов соответствующая одновременному нажатию на клавиши <ctrl> и <d> приводит к закрытию терминальной линии. В случае реализации терминальной линии через сетевой протокол с установкой соединения соединение будет разорвано.

Механизм взаимодействия с пользователем через консоль реализован следующим образом. Предполагается, что на противоположном от пользователя конце терминальной линии запущено специальное приложение (login), которое выводит на виртуальный экран монитора приглашение указать имя своей учётной записи и набрать пароль на виртуальной клавиатуре. Эта информация считывается программой login и проверяется. Если имя и пароль зарегистрированы на интерфейсной машине, то программа login завершается и запускается специальная программа командный интерпретатор. Всё дальнейшее взаимодействие с интерфейсной машиной ведётся через этот командный интерпретатор.

В UNIX системах существует огромное множество таких командных интерпретаторов. Вот список наиболее популярных из них:

- bash
- csh
- ksh
- tcsh
- zsh

Выяснить каким именно вы пользуетесь в текущий момент можно при помощи команды 'echo $SHELL'. Далее будет рассмотрен список команд интерпретатора. Часть из этих команд встроено непосредственно в интерпретатор, а часть является отдельными внешними по отношению к интерпретатору программами (внешними командами).

Взаимодействие с командным интерпретатором производится через командную строку. Пользователь в строке вводит команду с параметрами и получает на

экран текстовый вывод команды. Сигналом к началу выполнения команды служит ввод с виртуальной клавиатуры символа перевода строки.

Существует возможность исполнить группу команд как одну команду. Команды между собой могут быть объединены несколькими способами: через символ ';' и через символ '|'. В случае с ';' команды исполняются последовательно слева направо. В случае указания '|' команды также выполняются по очереди слева направо, но поток вывода левой команды является потоком ввода правой команды; таким образом реализуется конвейер команд.

Указание символа '&' в конце команды означает, что команда будет выполнена в фоновом режиме. То есть несмотря на то, что работа команды ещё не закончена интерпретатор либо показывает приглашение для ввода следующей командной строки, либо исполняет следующую команду по порядку в командной строке.

При помощи символов '>' и '<' можно перенаправлять поток вывода и ввода команды в файл и из файла соответственно. С помощью '>>' файл можно дописывать. А при помощи комбинации символов '2>&1' в bash можно поток ошибок можно присоединить к потоку вывода.

3.3.1. Работа с файловой системой

Наиболее популярное действие производимое с файловой системой — это просмотр её состояния. Для этой цели существуют 2 команды: встроенная с минималистическим интерфейсом **dir** и внешняя с богатым интерфейсом **ls**. Рассмотрим некоторые наиболее важные параметры команды **ls**.

параметр (ключ)	описание
-a	показывает информацию о всех файлах и каталогах. По умолчанию файлы и каталоги начинающиеся с точки не отображаются.
-b	показывает информацию со специальными защитными символами. Позволяет непосредственно указать этот файл в командной строке интерпретатору.
-d	показывать только каталог.
-t	Сортировать файлы по времени изменения файла.
-s	Печатать размер для каждого файла.
-S	Сортировать файлы по размеру.
-l	показывать информацию об одном файле или каталоге в одной строке.
-r	сортировать файлы в обратном порядке.
-U	Не сортировать вообще.
-R	Выдавать информацию рекурсивно по всем подкаталогам.

Следующее действие, которое необходимо совершать — это перемещение по каталогом файловой системы. В UNIX существует две встроенные команды **cd** и **pwd**. При помощи команды **pwd** можно выяснить какой каталоге каталог файловой системы в данный момент выбран в качестве текущего. С помощью

команды **cd** в качестве текущего можно установить другой каталог, путь до каталога указывается в параметре команды **cd**, например «`cd /tmp`». Если никакой параметр не указан, то в качестве текущего каталога будет выбран домашний каталог пользователя.

Копирование файлов производится при помощи внешней команды **cp**. Для команды указывается что копируется и куда, например «`cp file1 file2`» скопирует `file1` в `file2`, перезаписав `file2` если он уже существовал. У команды **cp** есть свой набор параметров. Например с ключём `-r` произвести копирование каталога вместе со всеми его подкаталогами. Если не указано куда производить копирование, то копирование будет произведено в текущий каталог. У **cp** есть специальный ключ `-f`, при указании этого ключа **cp** ничего не будет спрашивать у пользователя и будет продолжать копирование несмотря на возможное наличие ошибок и небезопасных для пользователя ситуаций (например с переписыванием файлов).

С помощью **rmdir** можно удалять каталоги, удаляемый каталог не должен содержать файлов, включая скрытые файлы, а с помощью **rm** удалять файлы. По аналогии с **cp** для **rm** возможно указать ключ `-f`. В результате все запросы подтверждения удаления будут подавлены. Можно также указать ключ `-r` для рекурсивного удаления файлов и каталогов. Сочетание ключей `-r` и `-f` нужно использовать с осторожностью. Случайный запуск «`rm -rf`» в каталоге «`/`» от имени пользователя root приведёт к весьма плачевным последствиям для операционной системы.

Команда **mkdir** создает новый каталог. Например после исполнения команды «`mkdir -p project/programs/December`» будет создан каталог с заданным именем в текущем каталоге. Ключ `-p` позволяет создавать промежуточные родительские каталоги.

В UNIX файлы перемещаются командой **mv**. Она эквивалентна комбинации команд **cp** и **rm**. Может использоваться как для перемещения файлов и каталогов, так и для их переименования. Если в качестве каталога назначения указан существующий каталог, то перемещаемый каталог становится подкаталогом указанного каталога.

Важным понятием в файловых системах UNIX является понятие ссылки на файл. В файловой системе каждому файлу и каталогу сопоставлены 2 понятия набор блоков данных файла или каталога и индексный дескриптор. В блоках данных хранятся собственно сами данные файла, а индексный дескриптор служит для хранения информации о самом файле: атрибуты доступа, является ли этот файл файлом или каталогом, где находятся блоки данных, и т. п.. Оказалось, что в некоторых случаях удобно для одного и того-же набора блоков данных иметь несколько индексных дескрипторов. По сути это означает доступ к одному и тому-же набору данных под разными именами. Для реализации такого «двойного» именования данных в командном интерпретаторе создана команда **ln**.

Команда **ln** создает нове имя (ссылку) на уже существующий файл. Таким образом у файла появляется несколько имен. Однако у ссылки есть один недостаток. Она должна указывать на файл в той же файловой системе (точки монтирования файла и ссылки на файл должны совпадать). Для решения

проблемы несовпадения точек монтирования была создана так называемая символическая ссылка. Символическая ссылка это специальный файл в котором написан путь до файла или каталога на который он ссылается. Поскольку символическая ссылка это обычный файл, содержимое которого ядро операционной системы специальным образом интерпретирует, её можно положить в произвольную файловую систему. Символическая ссылка создаётся с помощью ключа `-s`.

Приведём следующую команду интерпретатору как иллюстрацию работы **ln**: «`ln -s /usr/local/mpich/bin/mpicc /usr/local/bin`». По этой команде при запуске `mpicc` из каталога `/usr/local/bin` реально будет запускаться `mpicc` из каталога `/usr/local/mpich/bin`.

Довольно часто в UNIX приходится менять права доступа на файл/каталог и иногда приходится менять владельца на файл/каталог. Это производится при помощи внешних команд **chmod** и **chown** соответственно.

В файловых системах UNIX с каждым файлом и каталогом сопоставлен набор атрибутов: у файла есть владелец и группа, а также набор прав доступа. У файла/каталога есть 3 различных объединения прав доступа: объединение для владельца, объединение для группы пользователей, которая приписана файлу, и объединение для остальных пользователей. Для каждой категории прописаны права на чтение, запись/ изменение и исполнение. Для каталога право на исполнение означает возможность указания данного каталога в качестве текущего.

Права доступа задаются набором битов. С файлом сопоставляется дополнительный атрибут прав доступа: бит замещения (suid bit). В случае выставления такого бита процесс при исполнении такого файла получит права владельца/группы файла, а не того, кто запускает этот файл (в нормальной ситуации файл запускается с правами запускающего). С каталогом сопоставляется закрепляющий бит (sticky bit); он означает, что файлы одного из владельцев в этом каталоге могут менять только пользователь root и владелец файла.

Команда **chmod** предназначена для смены прав доступа к файлам. Для данной команды нужно указывать форматную строку. Форматная строка устроена следующим образом:

`[ugoa...][[+-=][rwxXstugo...]...][,...]'.

Каждый аргумент — это список символьных команд, разделенных запятыми, изменяющих права доступа к каталогу.

Каждая символьная команда форматной строки начинается с нуля (ноль означает указание прав доступа в восьмеричном виде) или набора букв 'ugoa'; эта комбинация указывает, чьи права доступа к файлу будут изменены: пользователя, владеющего файлом (u); группы, владеющей файлом (g); других пользователей (o) или же всех пользователей (a). Символ 'a' эквивалентен 'ugo' (маска на все три поля сразу). Если не задан ни один символ, то автоматически будет использоваться символ 'a'.

Оператор '+' добавляет выбранные права доступа к уже имеющимся правам доступа файлов; '-' удаляет эти

права; а '=' присваивает в то эти права каждому указанному файлу.

Символы 'rwxXstugo' указывают на новые права доступа того пользователя, который задан одним из символов 'ugoa'. Здесь символы обозначают следующее:

r — чтение;

w — запись и изменение;

x — исполнение;

X — бит исполнения, но срабатывающий только для каталогов;

s — бит замещения, возможно указание для владельца и группы;

t — закрепляющий бит;

У команды **chmod** есть набор параметров. Есть параметр `-f`, означающий «молчаливое» изменение прав. В противоположность «молчаливому» изменению есть «громкое» изменение `-v`, здесь выдаётся отчёт на каждый файл. И есть ещё возможность рекурсивного изменения прав доступа, которое указывается параметром `-R`.

Приведём несколько примеров: «`chmod g-s file`» снимает бит замещения для группы. «`chmod -R g +r $HOME`» открывает домашний каталог пользователя для чтения группой.

Смена владельца у файла/каталога производится с помощью команды **chown**. В отличии **chmod** форматная строка устроена много проще «`user:group`». Также, как и для **chmod,** доступны параметры `-f`, `-v`, `-

`R`, которые обозначают в точности тоже самое что и в команде **chmod**.

Для вывода содержимого одного или нескольких файлов, и записи файлов из стандартного потока ввода в UNIX существует команда **cat** (акроним от *concatenate*). Для объединения файлов в один файл может использоваться в комбинации с операциями перенаправления (> или >>).

```
# Порядок работы с 'cat'

cat filename      # Вывод содержимого файла.

#Объединение содержимого 3-х файлов в одном.
cat file.1 file.2 file.3 > file.123
```

Рассмотрим наиболее интересные параметры **cat**, ключ `-n` вставляет порядковые номера строк в выходном файле. Ключ `-b` нумерует только не пустые строки. Ключ `-v` выводит непечатаемые символы в нотации с символом ^. Ключ `-s` заменяет несколько пустых строк, идущих подряд, одной пустой строкой.

Поиск файлов в UNIX осуществляется с помощью внешней команды **find**. Формат команды следующий: «`find path expression`». **find** ищет все файлы подходящие под определяемое логическое выражение начиная от указанного первым параметром каталога. Наиболее популярен поиск файла по имени например «`find /usr/src/include -name map`». Типы поиска, указанные в логическом выражении можно комбинировать при помощи '`(`', '`)`', '`!`', '`-and`', '`-or`'. Покажем некоторые параметры которые можно

указывать при поиске файлов в логическом выражении:

тип поиска	описание
-amin *n*	файлы к которым был осуществлён доступ не больше чем n минут назад.
-anewer *n*	Файл был модифицирован не позже чем n минут назад.
-fstype *type*	Выбирает все файлы находящиеся на файловой системе указанного типа.
-name *pattern*	выбирает файлы где имя соответствует образцу.
-path *pattern*	Выбирает файлы путь к которым соответствует образцу.
-perm *mode*	выбирает файлы у которых права соответствуют указанным.
-size *size*	размер файла не меньше чем указанный. Можно указывать в байтах, килобайтах, блоках и словах (обычно 2 байта).
-type *type*	Выбирает только файлы определённого типа: b, c, l, d, f, p, s. Например можно указать искать только файлы блочных устройств: «`find /dev -type b`».
-user name	искать файлы принадлежащие пользователю.

3.3.2. Работа с текстовым выводом

Типичная проблема, возникающая перед пользователем заключается в том, что зачастую программы выдают на виртуальный терминал довольно много текста. Текст как минимум может не помещаться на экран. Для решения этой проблемы существует внешняя команда **more** и внешняя команда **less**. Команда **less** отличается от команды **more** богатством возможностей. Команда **more** позволяет поэкранно отображать содержимое файла. Основным преимуществом **more** перед **less** является простота реализации, благодаря чему, в установленном дистрибутиве UNIX **more** присутствует всегда.

Команда **less** позволяет перемещаться по файлу не только от начала к концу, но и в обратном направлении, а также перемещать по файлу не только страницами, как в **more**, но и отдельными строками. В **less** встроена возможность поиска по файлу, в целом поиск устроен так же как в текстовом редакторе **vi**, который будет рассмотрен позже.

Существуют 2 команды и **head** и **tail**, которые позволяют взглянуть на начало и конец файла соответственно. С помощью **tail** вы можете просмотреть некоторое количество последних строк файла, по умолчанию 10, не листая весь файл целиком. С **head** аналогично, но только для начала файла. Параметром `-n` можно указать число показываемых строчек.

3.3.3. Работа с процессами

Довольно часто пользователю приходится производить манипуляции со своими процессами. Наиболее популярные для этого внешние команды **ps**, **top**, **renice** и **kill**.

При помощи команды **ps** можно посмотреть список запущенных процессов, как своих собственных, так и вообще всех процессов. К сожалению параметры для команды **ps** не стандартизированы, в результате в разных UNIX операционных системах они отличаются. Рассмотрим наиболее важные параметры **ps** в LINUX.

параметр	**описание**
-A,-e	Показывает информацию обо всех процессах.
-a	Список всех процессов, которые присоединены к терминалу и которые были запущены пользователем вызвавшим **ps**.
-x	Снимает ограничение на присоединённость процесса к терминалу. «`ps -ax`» выдаёт все запущенные на машине процессы
-u	Позволяет выводить информацию о процессах пользователя указанных как спиток пользователей, разделённый запятыми. Команда «`ps -aux`» выведет информацию о всех процессах всех пользователей с указанием идентификаторов пользователя и группы (предполагается, что пользователя 'x' может не существовать).
-g	Выводить процессы согласно вхождению в группу.
-t	Задаёт ограничение на терминал. Терминалы указываются в списке разделённом запятыми.
-o	Формирует строку вывода у команды.

Если пользователю интересно, как процессы используют ресурсы компьютера, то он может воспользоваться командой **top**.

Команда **top** выдаёт список процессов отсортированный по степени использования процессора или памяти. К сожалению не во всех UNIX системах такая команда называется **top**, например в AIX её роль выполняет команда **topas**. Из интересных параметров для LINUX можно выделить следующие:

параметр	описание
-b	Не запрашивает внутренние команды и исполняет число запросов к операционно системе указанное с ключём `-n`.
-d	Задежка в секундах и долях секунды через которую будет обновлён вывод команды **top**.
-u,-U	С помощью соответствующих опций можно показывать информацию только для указанного пользователя. для `-u` по эффективному идентификатору пользователя и просто по идентификатору пользователя для `-U`.
-p	показывать только для процесса с фиксированным идентификатором.

Если, после просмотра состояния с помощью команды **top** пользователю хочется изменить приоритет процесса, то возможно изменить приоритет процесса с помощью команды **renice**.

Наконец, завершить процесс или выставить для него сигнал можно при помощи команды **kill**. Синтаксис команды **kill** следующий: «`kill -signal_number pid`». Если указать в качестве идентификатора процесса -1,

то сигнал будет выставлен для всех процессов. Например при помощи «`kill -9 -1`» можно принудительно завершить все процессы, которым пользователь имеет право выставлять сигнал. По умолчанию, если номер сигнала не указан, выставляется сигнал с именем SIGTERM, который должен приводить к завершению процесса, однако реакция на сигнал может быть переопределена процессом и процесс не завершится. Для сигнала с номером 9 переопределить реакцию нельзя, поэтому для гарантированного завершения процесса целесообразно выставлять сигнал с номером 9.

3.3.4. Работа с учётными записями

Типична ситуация, когда пользователь хочет знать, кто кроме него в текущий момент работает на машине. Для этих целей существуют две внешние команды **who** и **finger**. Обе команды выдают список пользователей зарегистрированных в текущий момент на машине. **finger** показывает дополнительную информацию о пользователе, в том числе, даже если он не работает на машине в текущий момент.

Другая типичная задача, которую решает пользователь — это задача смены пароля. Для этой цели существует команда **passwd**. С помощью **passwd** можно сменить не только собственный пароль, но и пароь пользователя указанного в качестве аргумента (конечно если достаточно прав доступа). И ещё можно срок действия пароля, заблокировать/разблокировать пользователя.

С помощью команды **chsh** можно изменить свой интерпретатор командной строки на какой либо другой из списка разрешённых интерпретаторов или вообще

на произвольную программу, если **chsh** запускается от имени пользователя root.

3.3.5. Справочная информация в UNIX

Наиболее стандартизованный формат представления справочной информации в UNIX — это представление в виде справочных страниц (man pages). Доступ к ним осуществляется с помощью команды **man**. Например «`man ls`» позволяет вывести справочную информацию для команды **ls**. В идеале работа пользователя в UNIX должна начинаться с набирания команды «`man man`».

Все справочные страницы распределены по нескольким секциям.

секция	описание
1	В этой секции содержится информация о запускаемых программах и командах интерпретатора, например о **passwd**, **ls**.
2	Информация о системных вызовах ядра.
3	Информация о библиотечных функциях, например описание функции `printf`.
4	Информация о специальных файлах. Обычно о содержимом каталога `/dev`.
5	информация об устройстве конфигурационных файлов, например `/etc/passwd`.
6	Руководство по играм
7	Глобальная сборная информация. Например здесь содержится информация о системе *Xwindows* в целом. «`man X`»

секция	описание
8	Информация о демонах и командах системного администрирования, которые обычно может запускать только пользователь root.

В случае, когда существуют два объекта одинаково называющиеся, например как команда **passwd** и конфигурационный файл `passwd`, для отображения пользователю будет выбрана та страница с информацией, которая находится в секции с меньшим номером. С целью разрешения конфликтов такого рода секцию для команды **man** можно указать явно, например: «`man 3 printf`».

Для облегчения навигации в пространстве команд и документации существуют ещё 2 команды: **whatis** и **apropos**. Команда **whatis** ищет справочную страницу по слову, аналогично команде **man**, но выдаёт только заголовок страницы. Команда **apropos** предназначена для поиска информации по теме. Данной команде можно указать фразу, после чего будут найдены те страницы в заголовках который встречается указанная фраза. Ниже приведён приведена иллюстрация работы команды **apropos**.

```
salnikov@comp:~$ apropos power
acpid (8)            - Advanced Configuration and Power Interface event
daemon
cpow (3)             - complex power function
cpowf (3)            - complex power function
cpowl (3)            - complex power function
DPMSForceLevel (3x)  - forces a DPMS capable display into the
specified power level
ldexp (3)            - multiply floating-point number by integral power of
2
ldexpf (3)           - multiply floating-point number by integral power of
2
ldexpl (3)           - multiply floating-point number by integral power of 2
on_ac_power (1)      - test whether computer is running on AC power
pow (3)              - power functions
pow10 (3)            - base-10 power functions
pow10f (3)           - base-10 power functions
pow10l (3)           - base-10 power functions
poweroff (8)         - stop the system.
powf (3)             - power functions
powl (3)             - power functions
```

Кроме **man** в UNIX присутствует и другая система документации: **info**. Если идиология **man** приписывает всю информацию помещать на одну страницу, то в **info** информация об одном объекте помещается сразу в несколько страниц, между которыми осуществляется навигация. Таким образом решается проблема не структурированности и избыточности информации на одну страницу. Однако несмотря на достоинства **info**, **man** остаётся более популярным средством документирования.

3.3.6. Высокоуровневые оболочки

Для облегчения навигации в файловой системе, а также для предоставления интуитивно понятного интерфейса к командам интерпретатора была создана специальная оболочка называемая Midnight Commander по аналогии с широко известным пользователям

операционной системы MS DOS оболочкой Norton Commander. Запускается с помощью команды **mc**.

mc содержит в себе текстовый редактор (**mcedit**), список реакций в случае открытия файла определённого типа (MIME types), содержит простой интерфейс к команде поиска файла (**find**), встроенная возможность «на лету» распаковывать архивы и отображать их внутренность как файловую систему с возможностью копировать нужные файлы и каталоги в реальную файловую систему, включена возможность предоставления доступа через ftp, ssh и samba доступ к удалённой файловой системе как к своей собственной (без указания точки монтирования). Перечисленные возможности наиболее яркие, есть и другие возможности и особенности данной консольной оболочки.

3.4. Консольные текстовые редакторы

Существует огромное число консольных текстовых редакторов. Вот наиболее часто встречаемые из них: **vi**, **vim**, **nano**, **emacs**, **joe**, **mcedit**. Они служат для разных целей.

Есть довольно простые текстовые редакторы: такие как **nano**. У них очень простой интуитивно понятный интерфейс, однако они практически ничего не умеют. Эти редакторы нужны для того, чтобы быстро что-то поправить в конфигурационном файле когда никакого более продвинутого редактора запустить нельзя.

Встроенные редакторы, такие как **mcedit** предназначены в первою очередь для использования их через оболочку, но в принципе их можно использовать и самостоятельно. Могут быть как достаточно сложными, так и простыми.

Сложные редакторы предназначены для полноценной обработки текста. Например редактор **vim** включает в себя весьма богатые возможности по манипуляции с текстом. Редактор **emacs** написан на lisp, в следствии чего является легко расширяемым и в него легко встроить практически всё что угодно. Например в **emacs** встроен почтовый клиент.

К сожалению нормальная работоспособность консольных редакторов сильно зависит от функциональности того виртуального терминала на котором они запущены. Редактор **vi** устроен таким образом, чтобы работать на любом типе виртуального терминала. Он не использует ни стрелочки, ни функциональные клавиши. Это приводит к некоторой не удобности его использования с одной стороны, но к возможности установить на любую UNIX машину с произвольным типом терминала.

3.4.1. Редактор vi

Редактор **vi** работает в 3-х режимах: режим команд, визуальный режим и режим вставки. В момент запуска редактор переключается в режим команд. В режиме команд по сути происходит управление самим редактором: есть команды чтения и записи файла, команда выхода, а также набор команд для работы **vi** как текстового процессора. В визуальном режиме можно выделить какую-то область текста и работать с ней затем в командном режиме. В режиме вставки осуществляется перемещение по тексту и его посимвольное изменение.

3.4.1.1. Команды перемещения курсора

Переход в командный режим из остальных режимов происходит по нажатию клавиши <esc>. Переход в

визуальный режим осуществляется нажатием клавиши с символом 'v'. Переход в режим вставки и замены происходит из командного режима по нажатию 'i'.

В командном режиме символы 'h' ,'j', 'k', 'l' работают как команды перемещения курсора на одну позицию влево, вниз, вверх, вправо соответственно. Символ 0 (клавиша «ноль», не «о») перемещает к началу строки. Символы '^' и '$' перемещают курсор к началу текста и к концу текста соответственно. Символ 'w' перемещает курсор к началу следующего слова, а символ 'e' перемещает к концу текущего слова.

Для поиска нужного фрагмента в тексте предусмотрена операция перемещения курсора в позицию текст после которой соответствует, шаблону задаваемому регулярным выражением. Эта операция производится по указанию '/' и регулярного выражения после него, например «/main\{». Команда начинает выполняться при введении символа перевода строки. При повторном указании символа '/' произойдёт поиск следующего фрагмента текста удовлетворяющего шаблону в тексте; такого же эффекта можно добиться указанием символа 'n' и вводом символа перевода строки после него.

Многие из приведенных выше команд перемещения курсора могут быть предварены коэффициентом повторения; в таком случае команда просто повторяется несколько раз. Например: **3w** — передвигает курсор вперед на три слова. Для некоторых команд коэффициент повторения имеет особое значение.

4H — переходит на четвертую строку видимую на экране,

8L — переходит к восьмой строке от края(низа) экрана,

3$ — переходит к концу третьей строки (вниз).

Для некоторых команд (например, '^') коэффициент повторения игнорируется; для других (например., '/' и '?') он вообще не применим.

Для программистов очень полезна команда '%' по этой команде, если курсор стоит на скобке ищется парная к текущей скобка после чего курсор переводится в соответствующую позицию.

3.4.1.2. команды изменения текста

Все команды манипуляции с текстом в vi производят эту самую манипуляцию в том месте куда подведён курсор. Например с помощью команды 'd' можно удалить один символ непосредственно под курсором, а с помощью команды «dd» удалить строку на которой стоит курсор. Команду «dd» можно использовать совместно с коэффициентом повторения: «3dd» удаляет три строки, текущую и две следующих. Команда 'd' может быть использована совместно с практически любой командой перемещения по тексту.

Изменение текста производится либо в режиме вставки, в который попадаешь по нажатию 'i', либо как результат операции поиска с заменой. Поиск с заменой будет обсуждён позже.

Можно вносить изменения в текст также с помощью команд копирования в буфер и вставки из буфера. Команда 'y' копирует фрагмент текста в буфер. Из буфера скопированный текст может быть вставлен в позицию после курсора при помощи команды 'p' и

перед курсором при помощи команды 'P'. Указав команду «yy» можно скопировать в буфер целую строчку и затем её вставить в нужное место файла.

Результаты манипуляций с текстом можно отменить введя команду 'u'. Эта команда отменяет последнее изменение в тексте.

3.4.1.3. Командная строка редактора vi

Когда вы вводите двоеточие в командном режиме, курсор автоматически перемещается на командную строку и ждет ввода команды. В режиме командной строки собраны команды требующие подтверждения своего исполнения. Команда указанная после двоеточия не начнёт исполнятся до тех пор, пока не будет введён символ перевода строки. Ниже приведены некоторые из команд режима командной строки.

команда	описание
w	Записывает редактируемый текст в файл. Например «w hello.txt» сохранит в файл hello.txt текущего каталога. «w!» означает попытаться записать файл всё-равно, например в случае если на файле нет прав доступа на запись.
e	Открывает файл для редактирования.
q	Выход из редактора vi. «q!» осуществит выход даже случае, когда редактируемый файл не сохранён.
s	Поиск с заменой в тексте.
r	Считывает файл и размещает его в конце, после текущего редактируемого текста.
!	исполняет команду командного

команда	описание
	интерпретатора UNIX, а вывод команды печатает на экран. Например «!ls» отобразит список файлов и подкаталогов текущего каталога.
число	Происходит перемещение курсора на строку с номером, заданным числом.

3.4.1.4. Поиск с заменой

Поиск с заменой производится в режиме командной строки редактора. Чтобы заместить один фрагмент текста другим существует команда 's'.

Её синтаксис с начальным ':' следующий:

```
:[область действия]s/{образец}/{строка}/[флаги]
[число применений]
```

Область действия задаёт диапазон строк в файле. Например «1,25» задают диапазон от первой до 25-ой строки, а «26,$» задаёт диапазон от 26-ой строки до конца файла. Далее идёт собственно сама команда 's', а затем, после символа '/' указывается образец который будет искаться в тексте. Образец задаётся регулярным выражением. В тех местах, где текст соответствует образцу будет произведена замена на строку, указанную после символа '/'. После строки можно указать флаги:

флаг	действие
&	Оставляет флаг таким, как он был в предыдущей команде поиска с заменой.
c	Требуется подтверждение каждой производимой подстановки строки в текст.
g	По умолчанию производится замена только в месте первого совпадения с шаблоном в строке. В случае указания данного флага подстановка будет произведена во всех местах строки, которые удовлетворяют шаблону.
n	Показывать число соответствий в которых не была произведена подстановка строки.
p	Печатать номер строки в которой была произведена подстановка.

приведём несколько примеров:

- **:1,$s/the/THE/g** Начиная с первой строки до последней (строки $), заместить все встреченные the на THE (g - означает глобальную замену)
- **:'a,.s/.*/ha ha/** От строки помеченной как a до текущей (строки .), заменить любой текст на строку "ha ha".

3.4.2. Редактор vim

Редактор **vim** — усовершенствованный редактор **vi**; усовершенствованный довольно сильно. Из наиболее бросающегося в глаза — это то, что для позиционирования курсора работают стрелочки. Сообщения **vim** значительно более информативны, чем

сообщения **vi**. В **vim** есть возможность создавать макрокоманды которые сопоставляются клавишам, в том числе функциональным. **vim** является многооконным консольным редактором. Кроме всего прочего в **vim** есть такая «мелочь» как подсветка синтаксиса и автоформатирование текста.

3.4.3. Редактор mcedit

Встроенный редактор обеспечивает выполнение большинства функций редактирования, присущих полноэкранным редакторам текста. Он вызывается из оболочки Midnight Commander нажатием клавиши '<F4>' при выбранном файле и '<SHIFT>+<F4>' для создания нового файла. его можно запустить отдельно от оболочки из командной строки указав **mcedit**.

Поддерживаются следующие возможности: копирование,перемещение, удаление, вырезание и вставка блоков текста; отмена предыдущих операций '<ctrl>+<T>'; '<ctrl>+<K>' выпадающие меню; вставка файлов; поиск и замена по регулярным выражениям, как в **vim** но через оконный интерфейс (а также собственный вариант операций поиска и замены, основанный на функциях scanf-printf); выделение текста по комбинации клавиш shift-стрелки в стиле MSW-MAC (только для linux-консоли); переключение между режимами вставки-замены символа; а также операция обработки блоков текста командами оболочки. Есть встроенная возможность подсветки синтаксиса и автоформатирования текста.

Для того, чтобы узнать, какие клавиши вызывают выполнение определенных действий, достаточно просмотреть выпадающие меню, которые вызываются нажатием клавиши '<F9>' в окне редактора. В меню не перечислены комбинации клавиш: '<Shift>

+<клавиши стрелок>' — выделение блока текста. Выделенный блок можно скопировать в файл ~.mc/cedit/cooledit.cli по нажатию клавиш '<Ctrl> +<Ins>'. По нажатию клавиш '<Shift>+<Ins>' производится вставка последнего скопированного в cooledit.clip блока в позицию курсора. '<Shift> +<Del>' удаляет выделенный блок текста, запоминая его в файле cooledit.clip. Работает выделение текста с помощью мыши, причем если удерживать клавишу '<Shift>', то управление мышью осуществляется терминальным драйвером мыши.

В целом редактор похож на редактор встроенный в *Norton Commander* для MS-DOS или на *Far Manager*, а сочетания клавиш на стандартные сочетания клавиш в MS Windows. Это делает UNIX среду более привычной для пользователя MS Windows.

4. Программное обеспечение для разработки программ на вычислительных системах

4.1. Компиляторы

Для пользователей вычислительных систем, в связи с тем, что их программы имеют тенденцию требовать огромного процессорного времени, вопрос оптимизации программного кода выходит на первый план. В связи с этим пользователям чрезвычайно полезно знать о методах оптимизации кода применяемыми современными компиляторами и какие из этих оптимизаций применимы, а какие нет, на пример по причине увеличивающегося числа ошибок округления.

В современном UNIX мире существует масса компиляторов. Наиболее известен из всех компилятор gcc по причине наличия его практически для всех UNIXподобных операционных систем и аппаратных платформ. Почти всегда разработчики аппаратных платформ являются ещё и разработчиками компилятора для них. По причине того, что никто как разработчики не знает особенности архитектуры своей аппаратной платформы компиляторы от разработчиков аппаратуры обычно создают более эффективный двоичный код чем gcc. Однако неприхотливость gcc и «либеральность» к исходному коду и диалектам языков программирования делают его весьма популярным компилятором.

Перед рассмотрением опций оптимизации современных компилятором остановимся на опциях

общих для всех C компиляторов в UNIX-подобных операционных системах:

Опция	описание
-o *file_name*	Задаёт имя файла, который получится на выходе.
-c	Остановиться на стадии компиляции. Будет получен объектный файл, но линковка произведена не будет.
-I *path*	Указывает путь до каталога с файлами подключаемыми по директиве `#include`. Если указано несколько каталогов с опцией **-I**, то обход каталогов при поиске файлов будет производится в порядке их указания в опциях (слева направо).
-D *variable*, **-D** *variable=value*	Определяет макропеременную. Получается так, как буд-то такая переменная была определена по дерективе `#define` в исходном коде программы.
-U *variable*	Делает макропеременную неопределённой. Действие аналогичное директиве `#undef`.
-l *library_name*	Определяет подключаемую библиотеку например «`-lpthred -lm`» подключит

Опция	описание
	одновременно библиотеку POSIX реализующую нити и библиотеку с реализациями математических функций.
-L *path*	Задаёт путь до файлов с библиотеками. Файлы с библиотеками должны иметь вид `libимя_библиотеки.a` или `libимя_библиотеки.so`.
-S	Остановиться на стадии создания ассемблерного кода. На вывод будет выдан ассемблерный код.
-E	Остановиться на стадии макроподстановки.
-g	Включать в код отладочную информацию. Обычно несовместимо с уровнями оптимизации больше одного.
-O,-O*level*	Задаёт уровень оптимизации. Если параметр не указан, то будет выбран уровень по умолчанию (обычно второй). «*-O0*» означает не производить оптимизацию кода вообще. «-Os» означает оптимизировать код по объёму занимаемой им памяти.

Рассмотрим теперь отдельные компиляторы, их опции оптимизации кода программ, а также методы оптимизации кода.

4.1.1. Компиляторы IBM

Корпорация IBM предоставляет для компьютеров с архитектурой Power набор компилялоров: xlc — C компилятор, xlC — C++ компилятор и xlf — фортран компилятор. Подробно сведения о компиляторах Компиляторах IBM можно почерпнуть в [58].

Данные компиляторы поддерживают большое число методов оптимизации кода программ, предоставляют возможность по автоматическому распараллеливанию и поддерживают стандарт OpenMP.

Большинство современных вычислительных систем создаются на основе процессоров в 64-х битной системой команд. Как правило присутствует возможность создания как 32-х битного бинарного программного кода, так и 64-х битного. В компиляторах IBM это указывается при помощи ключей «`-q32`», «`-q64`», «`-q32_64`». Указание одного из ключей существенно влияет на размер инструкции в памяти и как следствие на то, какая часть программного кода окажется в КЭШ памяти. По этой причине, программа, которая не требует больших объёмов памяти скомпилированная с ключом «`-q32`» будет на 64-х битной машине исполняться быстрее чем она-же откомпилированная с ключом «`-q64`». Указание ключа «`-q32_64`» приведёт к тому, что в бинарный файл будет помещён сразу и 64-х битный код и 32-х битный.

Как и любой другой компилятор компиляторы IBM позволяют указать в семействе машин ту целевую архитектуру, для которой предполагается создавать

программный код. Это делается с помощью опции «-qarch» с указанием pwr3, pwr4, pwr5, ppc970, и т. д.. После указания соответствующей опции будут использованы команды именно того процессора, который был указан. Как следствие в общем случае создаваемый бинарный код будет непереносим между машинами внутри семейства, но по идее будет более оптимальным для целевой архитектуры. Вместо «-qarch» можно указать более мягкий параметр «-qtune». Этот параметр означает, что оставляя код исполняемым на всех машинах внутри семейства стремиться выбирать команды таким образом, чтобы код наиболее оптимально исполнялся именно на целевой архитектуре.

По аналогии с архитектурой процессора есть возможность указать тип и предполагаемый алгоритм работы КЭШ памяти. Это делается при помощи опции «-qcache». Подробнее в [58].

В компиляторах IBM есть встроенная возможность высокоуровневого анализа циклов в исходном коде и генерации оптимизированного кода. Эта возможность включается по указанию «-qhot». Однако, для сохранения порядка арифметических операций, что чрезвычайно важно для научных задач необходимо указывать ключ «-qstrict». Ключу «-qhot» сопоставлены некоторые дополнительные параметры:

параметр	**действие**
arraypad, noarraypad	Позволяет компилятору увеличивать размер массивов с целью оптимизации циклов, которые каким-то образом обрабатывают эти массивы. Для обработки массивов с

параметр	действие
	размерностью большей либо равной двум уменьшает промахи в КЭШ и минимизирует подкачки страниц оперативной памяти из свопа.
simd,nosimd	Смысл этой опции заключается в объединение нескольких скалярных команд в одну большую векторную команду, которая исполняется быстрее, чем изначальный набор команд процессору.
vector,novector	Используется в сочетании с «`-qnostrict`» и «`-qignerrno`». Оптимизация заключается в преобразовании операций в теле цикла таких как извлечение квадратного корня, в вызовы специальных библиотечных функций, которые реализованы специальным образом через векторные команды.

Компиляторы IBM поддерживают в том числе и межпроцедурную оптимизацию. Этот вид оптимизации включается с помощью ключа «`-qipa`». Межпроцедурная оптимизация производится в 2 этапа. На первом этапе производится компиляция исходного кода в объектный код и сбор информации для межпроцедурной оптимизации. На втором этапе, на стадии линковки, производится оптимизация кода не для отдельного модуля, а для всей программы целиком. (Сразу для всех функций из всех модулей.) Данный вид оптимизации стремиться выкинуть из конечного кода

участки с идентичными вычислениями. Также может быть принято решение о подстановке тела функции в место её вызова, если это ускорит работу программы. С целью оптимизации размещения процедур в памяти может быть произведено их переупорядочивание. О конкретных технология проведения межпроцедурной оптимизации сказано в [59].

В компиляторах IBM присутствует возможность автоматического распараллеливания исходного кода. Для этого нужно указать «`-qsmp`» или «`-qsmp=auto`». В данном случае будет создан многопоточный программный код (не путать с многопроцессным программным кодом).

Встроенная поддержка OpenMP включается указанием ключа «`-qsmp=omp`». После этого компилятор будет воспринимать прагмы OpenMP и затем на основе них создавать многопоточный код.

4.2. Перенос данных между ваычислительными системами различных архитектур

При переходе с одной архитектуры вычислительной системы на другую возникает проблема переноса данных. На разных архитектурах могут оказаться разные форматы представления чисел. Эта проблем решается с помощью специальных библиотек, таких как NetCDF и HDF. В качестве альтернативы можно представлять данные в текстовом виде, но данный подход влечёт за собой значительное увеличение размеров файлов, по сравнению с размером бинарного файла.

4.2.1. Библиотека NetCDF

Разработчиками NetCDF [61] (Network Common Data Form) разработаны библиотеки позволяющие получить доступ к бинарным файлам в NetCDF формате из языков C, Fortran 77, Fortran 90, и C++. Сообществом пользователей NetCDF созданы также привязки и к другим языкам, например Python. NetCDF является проектом с открытым исходным кодом.

Библиотека NetCDF — реализует концепцую, рассматривающую данные как набор самоописываемых, переносимых объектов, доступ к которым может быть осуществлен через простой интерфейс. К данным можно обращаться способом, не требующим знания того как они физически хранятся. Дополнительная информация, например единицы измерения, может хранится вместе с самими данными. Библиотека NetCDF предназначена в первую очередь для хранения данных организованных как множество массивов. Обычно это сеточные данные, получающиеся при численном решении задач математической физики, например задачи прогнозирования климата.

4.2.1.1. Модель данных NetCDF

Пространство данных NetCDF содержит измерения (*dimensions*), переменные (*variables*) и атрибуты (*attributes).* Все они обладают как именем, так и целочисленным идентификатором, по которым к ним можно обращаться. Все эти компоненты могут быть использованы для того, чтобы отобразить значения данных и связи между данными во множество объектов реализованных как массивы. Множество таких объектов образует пространство данных (data set). Библиотека NetCDF позволяет осуществлять одновременный доступ к нескольким пространствам

данных, которые идентифицируются по собственным ID, в дополнение к обычным именам файлов.

Имена всех компонентов это произвольная последовательность алфавитно-цифровых символов и символов нижнего подчеркивания '_' , тире '-' и точки '.', начинающаяся с буквы или нижнего подчеркивания. (Тем не менее, слова начинающиеся с нижнего подчеркивания зарезервированы для системного использования). Регистры различаются. Имена нулевой длины не допускаются.

На простом примере (Рис. 4.2.1.1.) покажем концепции модели netCDF. Там представлены три переменных трёх типов, измерения, атрибуты. Используемая форма записи называется CDL (network Common Data form Language), которая обеспечивает удобную форму описания пространств данных NetCDF в текстовом виде. Система netCDF включает в себя утилиты для преобразования бинарных файлов в текстовые CDL файлы и наоборот.

Запись в форме CDL может быть сгенрированна автоматически с помощью утилиты **ncdum**. Другая утилита, **ncgen**, создает пространство данных NetCDF (или опционально C или FORTRAN код, генерирующий пр-во) из исходных CDL-данных.

```
netcdf example_1 { // example of CDL notation for a netCDF file

dimensions: // dimension names and sizes are declared first
        lat = 5, lon = 10, level = 4, time = unlimited;

variables:           // variable types, names, shapes, attributes
        float   temp(time,level,lat,lon);
                    temp:long_name     = "temperature";
                    temp:units         = "celsius";
        float   rh(time,lat,lon);
                    rh:long_name = "relative humidity";
                    rh:valid_range = 0.0, 1.0; // min and max
        int     lat(lat), lon(lon), level(level);
                    lat:units       = "degrees_north";
                    lon:units       = "degrees_east";
                    level:units     = "millibars";
        short   time(time);
                    time:units      = "hours since 1996-1-1";
        // global attributes
                    :source = "Fictional Model Output";

data:               // optional data assignments
        level   = 1000, 850, 700, 500;
        lat     = 20, 30, 40, 50, 60;
        lon     = -160,-140,-118,-96,-84,-52,-45,-35,-25,-15;
        time    = 12;
        rh      =.5,.2,.4,.2,.3,.2,.4,.5,.6,.7,
                 .1,.3,.1,.1,.1,.1,.5,.7,.8,.8,
                 .1,.2,.2,.2,.2,.5,.7,.8,.9,.9,
                 .1,.2,.3,.3,.3,.3,.7,.8,.9,.9,
                  0,.1,.2,.4,.4,.4,.4,.7,.9,.9;
}
```

Рисунок 4.2.1.1. Пример файла в CDL формате.

Измерения обычно используются для задания физической величины, например времени, широты, долготы или высоты. Измерение в NetCDF имеет имя и размер. Размер — произвольное положительное целое число. Также, при задании переменной может присутствовать одно измерение с размером UNLIMITED. Такое измерение называется неограниченным измерением. Переменная с неограниченным измерением может иметь произвольное число точек вдоль этого измерения.

Если переменная имеет неограниченное измерение, то оно должно быть наиболее значимым для этой

переменной и наиболее медленно изменяемой. Поэтому любое неограниченное измерение должно идти первым в CDL-записи и в соответствующем объявлении массива в языке C.

Объявления измерений могут находится на одной или нескольких строках, после ключевого слова. Объявления находящиеся на одной строке разделяются запятыми. Каждое объявление имеет форму ИМЯ=ДЛИНА. В приведенном примере 4 измерения- lat, lon, level, и time. Первые три имеют ограниченную длину, время (time) — неограниченно.

Базовым элементом данных в NetCDF является переменная (variable). Переменная представляет массив значений одного типа. Скаляр является массивом с числом измерений 0. У переменной есть имя, тип данных и форма, описываемая списком измерений при создании переменной. Переменная также может обладать атрибутами, которые могут быть добавлены,удалены или изменены после создания переменной. Тип переменной как в 32-битном формате, так и в 64-битном формате может быть одним из 6 базовых типов: байт, символ, короткое целое, целое, число с плавающей точкой, и число с плавающей точкой двойной точности. В CDL-записи только эти 6 типов могут использоваться. Их имена соответственно : byte, char, short, int, float, double. real может быть использован как синоним float. long — синоним int.

Атрибуты NetCDF используются для хранения метаданных. Большинство атрибутов представляют информацию о конкретной переменной. Атрибуты идентифицируются по имени или ID переменной и имени атрибута.

Некоторые атрибуты предоставляют информацию о всем множестве данных в целом, т. н. глобальные атрибуты. В этом случае предполагается пустое имя переменной в записи CDL, а в C и Fortran используется специальный ID глобальной переменной.

Атрибут обладает, именем, типом данных и длиной. Все атрибуты рассматриваются как векторы, скаляр рассматривается как вектор длины 1.

Тип атрибута указывается при его создании. Разрешенные типы атрибутов те же самые, что и для переменных. Иногда атрибуты с одинаковым именем, но для разных переменных должны иметь различные типы. Например атрибуту valid_max указывающему максимальное допустимое значение переменной типа int лучше быть типа int, а для переменной типа float — лучше быть типа float. Атрибуты могут быть удалены, их тип, длина и значение могут быть измены после создания, в то время как библиотека netCDF не позволяет ни удалить, не изменить тип или форму переменной для созданного пространства данных.

CDL запись для объявления атрибута для переменной следующая:

```
variable_name:attribute_name=list_of_values;
```

Для глобального атрибута:

```
:attribute_name = list_of_values;
```

Тип и длина не указываются явно для каждого атрибута, а вычисляются исходя из присвоенных значений. Все значения атрибута должны быть одинакового типа.

В приведенном примере units — атрибут для переменной lat, которому присвоен вектор из 13 элементов типа char. A valid_range — атрибут переменной rh который имеет длину 2 и значения 0.0 и 1.1 Также определен один глобальный атрибут – source, который используется для документирования пространства данных.

В отличии от переменных, которые предназначены для хранения данных, атрибуты предназначены для вспомогательной информации. Общее количество дополнительной информации ассоциированной с объектом netCDF и хранимой в атрибутах достаточно мало, чтобы полностью помещаться в память. Переменные же обычно велики для этого и разбиваются на секции для обработки. Другое различие состоит в том что переменные могут быть многомерными, а атрибут является скаляром или вектором.

Переменные задаются именем, типом и формой до того как им будут присвоены значения, так что может существовать переменные без данных. Значения атрибутов задаются при создании, если только это не атрибут с нулевой длиной. Переменная может иметь атрибуты, а атрибут нет.

4.2.1.2. Интерфейс с языком C.

Ниже приведена типичная последовательность вызовов функций для создания нового пространства данных.

```
nc_create /* create netCDF dataset: enter define mode */
  ...
 nc_def_dim /* define dimensions: from name and length */
 ...
 nc_def_var /* define variables: from name, type, ... */
 ...
 nc_put_att /* put attribute: assign attribute values */
 ...
 nc_enddef /* end definitions: leave define mode */
 ...
 nc_put_var /* provide values for variables */
 ...
 nc_close   /* close: save new netCDF dataset */
```

Для создания пространства данных нужен лишь вызов `nc_create`. При работе с открытым пространством данных вожзможно нахождение либо в режиме определения (define mode), либо в режиме данных (data mode). При создании нового пространства данных в качестве первого режима всегда режим определения. В режиме определения вы можете создавать измерения, переменные и аттрибуты, но не можете записывать данные. В режиме данных вы можете записывать данные и изменять аттрибуты, но не можете создавать новые переменные, аттрибуты и измерения.

Для создания каждого измерения нужно вызвать функцию `nc_def_dim`. Аналогично для каждой переменной — `nc_def_var`. В случае создания атрибута используется одна из функций семейства `nc_put_att`. Чтобы перейти из режима определений в режим данных надо вызвать функцию `nc_enddef`.

Находясь в режиме данных вы можете добавлять новые данные, изменять старые и изменять значение аттрибутов (если новое значение не требует большего объёма памяти). NetCDF позволяет вносить данные в

переменную несколькими способами: как одиночное значение, как массив значений, как отображаемый на измерения фрагмент и как склейку фрагментов сразу же в нескольких переменных NetCDF.

Одиночное значение может быть записано в переменную с помощью одной из функций `nc_put_var`, в зависимости от того какой тип записывается. (Для всех 6 типов данных NetCDF существует хотя бы одна функция в семействе.) Массивы или подмассивы можно записать используя функции семейства `nc_put_vara`. с помощью функций nc_put_vars можно помещать склейку фрагментов массивов. Отображение на фрагмент по изменениям (окно в измерениях) записывается с помощью одной из функций семейства `nc_put_varm`.

Для окончания работы необходимо явно закрыть все открытые пространства данных, вызвав функцию `nc_close`. Это необходимо делать поскольку: по умолчанию, доступ к файловой системе буферизуется. Как следствие буферизации, аварийный выход из программы приведёт к потере изменений. Буферизацию можно отключить установив флаг NC_SHARE при открытии, но даже в этом случае изменения сделанные в режиме определения не записываются пока не будет вызван `nc_sync` или `nc_close`.

5. Организация системы очередей задач пользователей

5.1. Задача планирования вычислений

5.1.1. Постановка задачи планирования вычислений

Рассмотрим задачу планирования работ при следующих условиях. Предполагается наличие многопроцессорной системы с n процессорами, на которой необходимо исполнить m работ. Процессор может выполнять в каждый момент времени только одну работу. Работы обладают длительностью $\tau_{i,j}$ в случае назначения i работы на j процессор. Для каждой работы определён директивный интервал с $T_{start,i}$ и $T_{finish,i}$, которые определяют момент времени после которого работа должна быть запущена и момент времени к которому работа должна быть завершена. Каждая работа обладает приоритетом p_i, который показывает её важность относительно других работ.

Целей планирования последовательности назначений работ по процессорам может быть несколько. Далее будут рассмотрены некоторые стратегии планирования назначений. Обычно стремятся минимизировать общее времени исполнения всего множества работ на многопроцессорной системе, однако возможны вариации, когда стратегия построена таким образом, что минимизация времени исполнения происходит косвенно и не гарантируется стратегией.

Для алгоритма планирования, в случае если планирование производится во время запуска программы или постоянно планировщиком заданий всегда можно выделить время планирования процессов на многопроцессорной системе. Это время между получением планировщиком информации о освобождении процессора и непосредственным назначением работы на исполнение. Важной задачей является минимизировать это время. К сожалению здесь приходится искать компромисс. Обычно расписание близкое к оптимальному требует большого времени планирования и наоборот легко сделать какое-то расписание, которое будет далеко от оптимального. Время планирования обычно зависит от количества работ. Важной характеристикой алгоритма планирования является характер данной зависимости. Алгоритм может строить хорошее расписание, но быть плохо масштабируемым к числу планируемых работ.

В целом теория построения расписаний подробно обсуждается в книге [1]. В книге [2] показано, что в общем случае данная задача относится к классу NP-полных задач, что говорит о том, к данной задаче неприменимы переборные алгоритмы, однако можно использовать эвристические.

5.1.2. Списочные алгоритмы

Существует ряд довольно простых алгоритмом планирования, которые позволяют выбрать работу для назначения на освободившийся процессор из списка готовых работ. Критериев выбора может быть несколько. Например, назначается первая по списку работа (First Come First Served - FCFS) или работа с максимальным приоритетом (Higest Priority First - HPF). Возможно назначение в первую очередь работы, которая выполнится за кратчайшее время

$\text{job\_number}_j = argmin(\tau_{k,j})$, здесь k - номер одной из работ, директивный интервал которых допускает запуск к текущему моменту времени. Один из вариантов это когда на процессор назначается работа у которой верхняя граница директивного интервала наиболее близка к текущему моменту времени $t_{current}$, то есть $\text{job\_number} = argmin(t_{current} - T_{finish,k})$ (Earliest deadline first – EDF). Довольно популярен алгоритм, где процессоры образуют виртуальное кольцо с маркером и следующая готовая к исполнению работа назначается на процессор обладающий маркером, далее маркер передаётся следующему процессору (Round Robin RR).

Такого рода алгоритмы используются в системах реального времени. Примеры использования данных алгоритмов можно найти в [3,4]. Планирование процессов в операционной системе Linux в ядре начиная с версии 2.6.8.1 осуществляется на основе модификации RR алгоритма, где, по мнению создателей алгоритма время планирования не зависит от числа процессов за счёт особой системы подсчёта приоритетов [5].

5.1.3. Алгоритм основанный на множестве очередей

Алгоритм FB (feedback) очередей с обратной связью использует n очередей, каждая из которых обслуживается в порядке поступления. Новая работа поступает в очередь с номером 0, затем после получения кванта времени она переходит в очередь со следующим номером и так далее после очередного кванта времени. Время процессора планируется таким образом, что он обслуживает непустую очередь с наименьшим номером. В методе FB каждая, вновь

поступающая на обслуживание работа получает высокий приоритет и выполняется подряд в течение такого количества квантов времени, пока не появится новая работа. Если приход новой работы задерживается, то текущая работа не может проработать большее количество квантов времени, чем предыдущая работа.

Данный алгоритм наиболее эффективно работает с большим числом коротких по времени работ. Основное преимущество очередей FB и RR по сравнению с алгоритмом EDF и алгоритмом кратчайшая по времени работа вперёд в том, что FB и RR не требуют предварительной информации о временах выполнения работ. Пример использования алгоритмов такого типа можно найти в работе [6].

5.1.4. Алгоритм имитации отжига

Алгоритм имитации отжига основывается на имитации физического процесса, который происходит при кристаллизации вещества из жидкого состояния в твёрдое. Предполагается, что атомы уже выстроились в кристаллическую решётку, но ещё допустимы переходы отдельных атомов из одной ячейки в другую. Предполагается, что процесс протекает при постепенно понижающейся температуре. Переход атома из одной ячейки в другую происходит с некоторой вероятностью, причём вероятность уменьшается с понижением температуры. Устойчивая кристаллическая решётка соответствует минимуму энергии атомов, поэтому атом либо переходит в состояние с меньшим уровнем энергии, либо остаётся на месте. Атом может с некоторой вероятностью перейти и в состояние с большим уровнем энергии, однако глобально процесс протекает с понижением энергии. Впервые метод был

предложен в работе [7]. В общем виде алгоритм описан в книге [8].

Для задачи планирования вычислений вводится понятие расписания. Расписание – последовательность исполнения работ для каждого процессора многопроцессорной системы. С каждой работой сопоставляется пара $(proc_j, \text{order\_number})$, которая задаёт процессор на котором она должна быть выполнена и порядковый номер на процессоре. Работы на процессоре исполняются по очереди в соответствии с возрастанием порядковых номеров. Вводятся 2 операции: операция $\text{change\_proc}(job_i, proc_j, proc_k, \text{order\_number})$ и операция $\text{change\_order}(job_i, proc_j, \text{order\_number})$. Здесь первая операция переносит работу с одного процессора на другой и задаёт порядковый номер, а вторая операция оставляет работу на том же процессоре, но меняет её порядковый номер на процессоре.

Вводится последовательность расписаний: $S_0, S_1, ..., S_n$. По достижении последовательности S_n алгоритм останавливается. Задаётся каким-либо образом начальное расписание S_0. Далее, к начальному расписанию случайным образом применяется несколько раз одна из описанных выше операций. В результате получается новое временное расписание S'. Временное расписание может стать постоянным с некоторой вероятностью $P(S', S_1)$. Таким образом задаётся переход от одной итерации алгоритма к другой. Для расписания S_k определён переход к расписанию S_{k+1} через случайное

применение операций с вероятностью $P(S', S_{k+1})$. Здесь вероятность задаётся формулой:

$$P(S', S_{k+1}) = \begin{cases} 1, & time(S') - time(S_k) \le 0 \\ \exp\left(-\frac{time(S') - time(S_k)}{Q_k}\right), & time(S') - time(S_k) > 0 \end{cases}$$

Здесь $time(S)$ функция, определяющая как долго многопроцессорная система будет работать по соответствующему расписанию задаёт аналог энергии для атомов в кристаллической решётки. Q_k - убывающая последовательность значений температуры. Закон, по которому происходит убывание последовательности и скорость убывания могут задаваться произвольным образом по желанию создателя алгоритма. Также можно варьировать число применений операций по изменению расписания на одну итерацию алгоритма.

О использовании алгоритма имитации отжига для планирования вычислений в системах реального времени можно прочитать в статье [9].

5.1.5. Генетический алгоритм

Одним из популярных способов составления расписаний для многопроцессорных систем является способ составления при помощи генетического алгоритма. Генетический алгоритм – это алгоритм поиска максимума или минимума некоторой функции с помощью направленного перебора области определения функции. Перебор проводится методами схожими с законами естественного отбора. Генетические алгоритмы являются подмножеством

эвристических алгоритмов поиска. Генетические алгоритмы описаны в книге Джона Холланда [10].

На некотором n мерном множестве задаётся числовая функция $f(x_1, x_2, \dots, x_n)$. Предполагается, что мы ищем максимум данной функции, в дальнейшем она будет называться функцией качества. На множестве выбирается некоторое количество точек $X_1=(x_{1,1}, \dots, x_{1,n}), \dots, X_m=(x_{m,1}, \dots, x_{m,n})$, которые образуют начальную популяцию P_0 . Каждое $X_1, \dots, X_m$ называются хромосомами или особями, а $x_1, \dots, x_n$ - генами в хромосоме. Для алгоритма определено некоторое число операций при помощи которых получается следующая популяция: операция скрещивания, операция мутации. При помощи генетических операций по текущей популяции P_k создаётся новая временная популяция P' . Временная популяция обычно больше по числу особей чем та из которой она получалась. В полученной временной популяции производится отбор. Для отбора особи сортируются в порядке убывания функции качества. Обычно отбор проводится таким образом, что из популяции удаляются особи с наименьшим значением функции качества, но с учётом случайным образом наложенного штрафа. В результате наложения штрафа в следующей популяции останутся как особи с плохим значением функции качества, так и особи с хорошим значением, такая стратегия может дать лучшую особь в следующем, относительно отобранного, поколении.

Операция скрещивания осуществляет перемешивание генетического материала между особями. Полученный таким образом перемешанный потомок, может собрать в себя положительные качества обоих родителей, в результате получится

особь со значением функции качества большим чем у родителей. В процессе применения операции скрещивания 2 хромосомы X_i и X_j обмениваются своими фрагментами при этом родительские особи остаются в новой временной популяции. Далее на рисунке приведён пример с операцией скрещивания, где случайным образом выбирается точка k внутри хромосомы, всё что находится правее точки k меняется местами в обоих хромосомах. Голова первой хромосомы соединяется хвостом второй хромосомы, и наоборот. В результате во временную популяцию добавляется 2 новых особи X'_i и X'_j. Математически это можно записать следующим образом: $X'_i = \begin{pmatrix} x_{i,r}, & r<k \\ x_{j,r}, & r \geq k \end{pmatrix}$ и $X'_j = \begin{pmatrix} x_{j,r}, & r<k \\ x_{i,r}, & r \geq k \end{pmatrix}$, где r задаёт смещение по хромосоме для осуществления обмена частями. То есть $r \in \{1,...,n\}$.

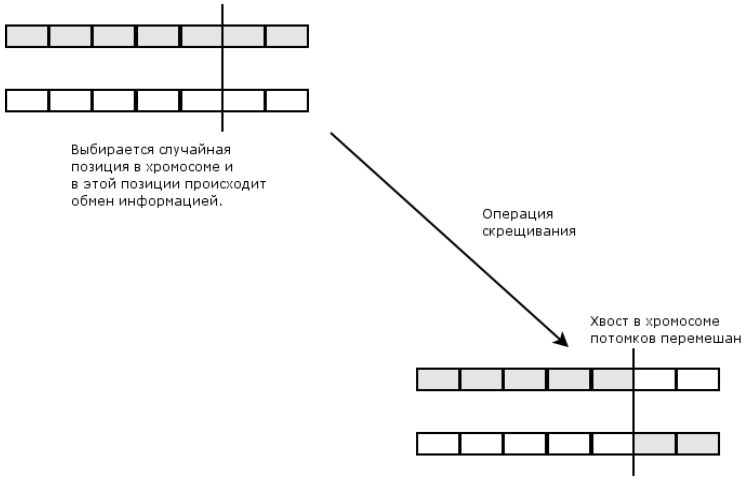

Рисунок 5.1.5.1. Пример операции скрещивания, в генетическом алгоритме.

Операция мутации вводится для того, чтобы алгоритм не скатывался в локальный экстремум. Если применять только операцию скрещивания, то в популяции не будет появляться принципиально новых особей и в конце концов будут исчерпаны все возможные вариации обменов фрагментами хромосом между особями. То есть операция скрещивания либо не будет менять особи либо будет приводить к появлению особей с более худшим значением функции качества. С целью предотвращения такой ситуации в новую популяцию могут добавлять некоторое количество случайных хромосом. Однако иногда можно незначительно изменить один или несколько генов в хромосоме чтобы существенно улучшить значение функции качества для особи. Операция мутации – как раз такое незначительное изменение одного или нескольких генов в хромосоме. Например в X_i в позиции k случайным образом меняем значение на

$x'_{i,k}$. В результате получим новую хромосому $X'_i=(x_{i,1},\ldots,x'_{i,k},\ldots,x_{i,n})$.

Хотя Холландом в книге [10] было доказано, что генетический алгоритм для двоичных хромосом фиксированной длинны всегда сходится к максимальному значению функции качества, реально алгоритм может работать довольно долго, поэтому обычно алгоритм останавливают раньше. Обычно бывает достаточно просто улучшить уже имеющееся приближение. В этом случае алгоритм может останавливаться в нескольких ситуациях:

1. по получении некоторого количества популяций,

2. в случае незначительного изменения функции качества для нескольких поколений популяций,

3. в случае получения в популяции особи с приемлемым значением функции качества.

Для решения задачи составления расписания исполнения работ на многопроцессорной системе генетический алгоритм составляется несколько отличным от классического способом. Здесь в качестве функции качества обычно используют функцию вычисляющую время работы многопроцессорной системы по рассматриваемому расписанию $time(S)$. Однако в отличии от классического генетического алгоритма здесь ищется не максимум функции качества, а минимум. В качестве хромосомы в данном алгоритме выступает сама запись расписания, где ген – это пара $(proc_j, \text{order\_number})$ привязанная к каждой работе входящей в расписание. Для операции

скрещивания существенных отличий нет, а операция мутации может перемещать работу с одного процессора на другой и менять порядковый номер работы на процессоре, либо оба варианта одновременно.

Впервые о использовании генетического алгоритма для построения расписаний исполнения работ было написано в статье [11]. О использовании генетического алгоритма для построения расписания исполнения работ с учётом задержек на передачу данных на кластере написано в [12]. Генетические алгоритмы также используются для создания расписаний в системах реального времени [13].

5.1.6. Алгоритм поиска критического пути

Предположим, что часть работ находятся в зависимости других работ, то есть пока не закончена одна работа нельзя начинать некоторое количество следующих. Построим граф, где в вершины будут соответствовать работам, ребра зависимостям между работами. Предполагается, что в графе есть вершина сток, к которой сходятся остальные вершины. Предполагается, что если мы достигли вершины стока, то никаких работ больше выполнять не надо, поскольку достигнута цель деятельности. В результате будет получен граф в чём-то аналогичный графу зависимостей по данным рассмотренному ранее. Для данного графа определяется понятие критического пути. Критический путь – это такой путь от вершины истока к стоку, который имеет максимальную длительность по времени исполнения входящих в него вершин.

Основная идея заключается в том, что критический путь ограничивает время исполнения параллельной программы, поскольку остальные пути от истока к

стоку в графе короче по времени исполнения вершин. Таким образом можно уменьшить время работы параллельной программы если каким-то образом уменьшить время затрачиваемое процессорами на обработку вершин критического пути. Для достижения нужного эффекта в первую очередь на процессоры многопроцессорной системы будут назначаться вершины входящие в критический путь. Сама задача поиска критического пути в ориентированном графе сводится к задаче поиска кратчайшего пути и может быть решена например при помощи алгоритма Дейкстры, который обладает сложностью $O(v^2)$, где v - число вершин в графе [14].

Пример применения такой стратегии к гетерогенным вычислительным системам можно найти в статьях [15,16].

5.1.7. Алгоритм обратного заполнения

Алгоритм обратного заполнения (Backfill) предназначен в первую очередь для «справедливого» распределения ресурсов многопроцессорной системы между работами. Предполагается, что каждая работа требует для своего исполнения не одного процессора, как это было в начальной постановке задачи планирования вычислений, а сразу нескольких процессоров. Каждая работа требует резервирования определённого количества процессорного времени. По мере поступления в многопроцессорную систему для исполнения работы выстраиваются в очередь с учётом приоритета работы.

Для многопроцессорной системы определяется конструкция отображённая на рисунке 5.1.5.1. По вертикали задаётся шкала процессоров. По горизонтали задаётся шкала времени, где текущий

момент времени отображается слева в начале шкалы. Шкале времени ставится в соответствие очередь работ требующих своего выполнения. Начало очереди совпадает с началом шкалы времени. Работы здесь представляются прямоугольниками, где по одной шкале откладывается число процессоров, возможно с именами, а по другой шкале откладывается зарезервированное работой процессорное время. Если работа находится в самом начале шкалы времени, то это означает, что она назначена на соответствующее подмножество множества процессоров многопроцессорной системы. На рисунке 5.1.5.1 работы 1 и 2 назначены на процессоры. В очереди могут быть работы, для которых, в текущий момент времени, нет необходимого количества свободных ресурсов. Такие работы следуют за назначенными на исполнение, например работа 3 на рисунке 5.1.5.1. Вследствие невозможности исполнения работы на многопроцессорной системе будут образовываться окна. Окно – множество свободных процессоров и интервал времени в который невозможно назначить никакую работу на свободный процессор. На рисунке 5.1.5.1 представлено два окна. Момент запуска работы 3 ограничен работой 1. В результате образуется окно 2, интервал которого во времени совпадает со временем исполнения работы 1 на процессоре. Поскольку работа 3 не блокирует собой всё множество ресурсов образуется окно 1.

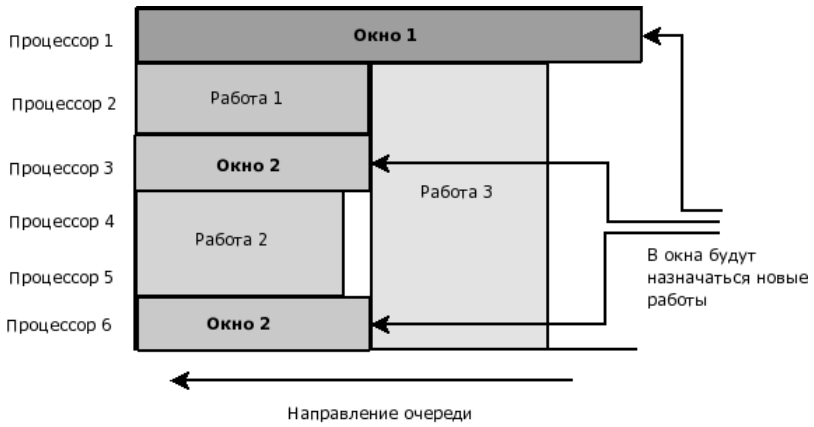

Рисунок 5.1.7.1. Пример работы алгоритма обратного заполнения.

Цель алгоритма – наиболее плотно заполнить образующиеся окна. Для этого среди имеющихся окон выбирается наиболее широкое окно, тоесть с максимальным количеством процессоров, и следующие работы, которые попадают в очередь, будут назначаться на процессоры этого окна. Если новая работа не помещается ни в одно из доступных окон, то она помещается в конец очереди. Таким образом работы распространяются в обратную сторону относительно шкалы времени.

Алгоритм обратного распространения довольно часто используется в системах очередей осуществляющих «справедливый» доступ пользовательским задачам к ресурсам многопроцессорных систем. К таким системам можно отнести например Maui. В его состав в качестве одного из алгоритмов управления задачами пользователя используется Backfill [17].

5.1.8. Алгоритм управления группами работ с прерываниями

Как и в случае с алгоритмом обратного заполнения алгоритм планирования группами работ (Gang) осуществляет распределение ресурсов многопроцессорной системы между множеством работ. Основное назначение данного алгоритма - распределять ресурсы между группами работ. Предполагается, что работы объединены в группы по степени важности, тоесть приоритет работы p_i приписан сразу же для всей группы работ. Работы одной группы разделяют множество процессоров таким же образом, как и в случае алгоритма Backfill, однако допускается прерывание работ в случае, если на многопроцессорную систему поступает группа работ с большим приоритетом. В этом случае страницы оперативной памяти на многопроцессорной системе занимаемые работой с меньшим приоритетом сохраняются во внешнюю память, где они образуют очередь отложенных работ. Из полученной очереди работы восстанавливаются согласно приоритетам групп и порядку сохранения во внешнюю память. Сперва будут восстанавливаться работы из группы с наибольшим приоритетом в том порядке в котором они были отложены, а затем произойдёт переход к группам с меньшим приоритетом.

О такого рода способе распределения ресурсов многопроцессорной системы написано в статьях: [18,19]. Данный алгоритм используется на SMP системах в том числе, на IBM pSeries 690. Алгоритм Gang включён в систему управления очередями IBM LoadLeveler [20].

5.2. Системы ведения очередей задач пользователей

5.2.1. Основные функции систем ведения очередей

В случае использования кластера для высокопроизводительных вычислений оказывается, что стандартный способ разделения ресурсов для задач пользователей не подходит. Стратегия, когда задаче отводится определённый квант времени по истечении которого ресурс отдаётся другой задаче, оказывается проигрышной, по той причине, что каждая задача стремится захватить всё множество доступных ресурсов и на переключение задачи тратится большое количество времени. Обычно задачи пользователей выстраиваются в очередь и ресурсы многопроцессорной системы отдаются задачам в порядке очереди. Задачи пользователей обычно являются параллельными программами, то есть множеством взаимодействующих между собой процессов каждый из которых должен быть распределён на отдельный процессор. Эти процессы могут создавать дочерние процессы. Новые, порождённые процессы должны быть распределены на те же процессоры, что и родительские и по завершению родительского процесса должны быть завершены, также как и родительские. Таким образом, для людей желающих запускать свои задачи на многопроцессорной вычислительной системе запуск задач производится по принципу пакетного режима функционирования операционной системы. Можно сформулировать несколько принципов такого пакетного режима:

- Для каждой пользовательской задачи определяется время не дольше которого будет выполняться задача пользователя.
- По причине не интерактивности исполнения задачи пользователя весь производимый задачей в процессе выполнения стандартный вывод (печать на экран), а также стандартный вывод сообщений об ошибках перенаправляются в специальные выходные файлы.
- Каждая запущенная задача получает уникальное имя, состоящее из символьного имени задачи и ее номера, что позволяет запускать одну и ту же задачу одновременно в нескольких экземплярах.
- Предполагается, что пользователь не имеет непосредственного доступа к вычислительным узлам входящим в кластер. Если пользователь желает запустить задачу, то он обязан воспользоваться системой очередей поставив свою задачу в очередь на исполнение. Система очередей пользователя обычно установлена на отдельной машине не являющейся одним из узлов кластера.

На примере LoadLeveler рассмотрим функции отдельных компонентов системы управления очередями. В LoadLeveler все машины, разделены по ролям [20]. Одна и та же машина может исполнять несколько ролей одновременно:

- Scheduling Machine (сервер планировщик) - машина собственно занимающаяся планированием назначений работ пользователей. В её функции входит следить за

состоянием очереди и отдавать команды вычислительным узлам кластера исполнять соответствующий фрагмент задачи пользователя. Сервер планировщик непосредственно осуществляет передачу необходимых данных для запуска команды на узле кластера, однако саму команду запускает специальная программа демон запущенная на вычислительном узле. Данная машина опрашивает демоны на вычислительных узлах с целью выяснения состояния запущенных там задач пользователя.

- Central Manager Machine (центральный менеджер) – машина занимающаяся предварительной фильтрацией задач с целью определения возможности их запуска и возможности помещения их в очередь на одном из серверов планировщиках. С данной машиной взаимодействуют Submit only машины.

- Executing Machine (вычислительный узел) – машина на которой будет запускаться код задачи. На вычислительных узлах запускается специальная программа демон, которая отвечает за непосредственный запуск кода задачи, следит за тем, чтобы задача не занимала больше ресурсов чем ей положено и завершает процессы задачи если они исчерпали отведённое время. Также данные машины информируют сервер планировщик к которому они приписаны о состоянии запущенных на вычислительном узле задач пользователя. Демон может на время остановить задачу пользователя усыпив все процессы задачи и принудительно отправить задачу в swap если это необходимо.

- Submiting Machine – машины с которых разрешено помещать задачи в очередь. На этих машинах пользователям позволяется запускать команды по подписыванию задач на помещение в очередь. Эти машины общаются с центральным менеджером и именно через него пользователю предоставляется информация о возможности или невозможности поставить задачу в очередь, а также посмотреть на состояние очередей.

Такая сложная и многоуровневая система построена специально для того, чтобы как можно сильнее изолировать пользователя от вычислительных узлов кластера.

5.2.2. Параметры настройки систем ведения очередей

В случае с LoadLeveler в системе присутствует 2 файла LoadL_admin и LoadL_config. В файле LoadL_admin содержится информация о машинах, которые находятся под наблюдением системы очередей, их ролях и способе распределения задач пользователей по классам задач. В случае с LoadLeveler задачи пользователей приписываются к классам и для класса определяется приоритет. Задачи из более приоритетного класса будут попадать на узлы кластера раньше задач приписанных к классу с меньшим приоритетом. Приписывать задачи к классу можно исходя из разных соображений. Например исходя из того, сколько времени для исполнения задача себе запрашивает. Далее приведён пример такого класса:

```
nightmare:      type = class
                wall_clock_limit = 12:00:00
                priority = 0
                exclude_users = prac
```

Можно также позволять включать и исключать задачи отдельных пользователей или групп пользователей в класс, например так как это выло показано во фрагменте выше. В этом файле описываются также параметры пользователей. Пользователям можно ограничивать число задач, которые могут одновременно находится в очереди, максимальное число одновремено запущенных задач. Для пользователя можно указывать приоритет, который будет прибавляться к приоритету класса, а также класс по умолчанию, ограничение на запрашиваемые ресурсы.

```
#oz:    type = user
#       default_class = oz_class
#       priority = 50
#       maxjobs = 5
#       maxqueued = 16
#       max_node = 3
#       max_processors = 8


jesus:  type = user
       priority  = 0
       maxjobs = 3
       maxqueued = 4
       priority = 0
       maxjobs = 4             # default maximum jobs user is
allowed


nivz:   type = user
       default_class = nivz_class
       maxjobs = 16            # default maximum jobs user is
allowed
       maxqueued = 16
```

В файле LoadL_config указывается всякая служебная информация, как то на каких портах будет устанавливаться tcp соединие с различными программами демонами запущенными в системе, какой

тип планировщика используется, в каких файлах ведётся журнал, каким пользователям и на каких машинах доверено администрирование очереди задач. Также в этом файле указывается иерархия классов, если необходимо один класс в другой и унаследовать его атрибуты.

5.2.3. Разнесение задач по приоритетам

Одной из довольно тяжёлых проблем, с которой приходится сталкиваться администратору вычислительного комплекса является проблема расстановки приоритетам задачам. В случае с LoadLeveler приоритеты вычисляются статически и определяются исходя из того, какому пользователю принадлежит задача и к какому классу относится задача. Из общих рекомендаций целесообразно задачи разделять минимум на 3 класса: о*тладочные, пакетные, фоновые* задачи.

- *отладочные задачи* - такие задачи, для которых пользователь ожидает, что они завершатся быстро. обычно такие задачи не требуют большого количества ресурсов.
- *пакетные задачи* – такие задачи, которые требуют большого количества ресурсов, могут требовать относительно продолжительного времени, время может измеряться не минутами, а часами, однако пользователь ожидает, что задача получит результаты относительно быстро, например в течении суток.
- *фоновые задачи* – это задачи, которые требуют большого количества ресурсов, обычно они выполняются неделями и пользователю многопроцессорной системы

не очень жалко если они будут прерваны на время.

Целесообразно выстраивать систему приоритетов таким образом, чтобы в первую очередь исполнялись отладочные задачи, затем пакетные и в последнюю очередь фоновые.

Довольно часто с системой приоритетов возникает проблема. Например на доверенном вам кластере пользователи очень часто пускают отладочные задачи в результате чего пользователи с задачами с большим временем исполнения никак не могут дождаться завершения своих задач. Возможна ситуация, когда пользователи используют кластер как набор мощных однопроцессорных машин и заказывают по много-раз на большое время один процессор или очень маленькое число процессоров по отношению к числу процессоров доступных в кластере. Весьма часто бывает ситуация, когда запрошенное число процессоров не совпадает с реально использующимся. Такая ситуация возникает в случае неэффективной параллельной реализации программы. Частично эти проблемы можно решить введя систему динамических приоритетов.

Например в системе ведения очередей на машине МВС-15000 используются динамические приоритеты [21]. К приоритету пользователя добавляется штраф, который вычисляется следующим образом. С каждым пользователем сопоставляется шкала

(120,300,600,1200,0)

В этой шкале задаётся учёт числа процессорных часов, которые пользователь занял в течении

определённого периода времени. Раз в период счётчик числа процессорных часов сбрасывается. Если пользователь превысил отведённую ему квоту, например в 120 часов, то к его приоритету добавляется штраф, в результате его задачи не будут мешать задачам других пользователей, которые насчитали меньшее число часов к текущему моменту. В случае превышения следующего барьера например в 300 часов ещё увеличить штраф. 0 в конце означает конец списка.

6. Список литературы

1. Экономико математическая библиотека: «Теория расписаний и вычислительные машины» под редакцией Э. Г. Кофмана Москва изд. «Наука» 1984г. 336 стр.
2. Garey, M. R. and Johnson, D. S. 1990 «Computers and Intractability; a Guide to the Theory of Np-Completeness.» W. H. Freeman & Co., pages: 338, ISBN: 0716710455.
3. M. Saleh, Z. Othman, and S. Shamala «FCFS Priority-based: An Adaptive Approach in Scheduling Real-Time Network Traffic» Networks and Communication Systems proceeding 527, 2006, ISBN 0-88986-590-6.
4. Marcia C. Cera, Guilherme P. Pezzi, Elton N. Mathias, Nicolas Maillard, Phillipe O.A. Navaux «Improving the Dynamic Creation of Processes in MPI-2» Lecture Notes in Computer Science LNCS 4192 Recent Advantages in Parallel Virtual Machine and Message Passing Interface, Volume 4192, pp. 247-255, 2006, ISBN-10: 3-540-39110-X ISBN-13: 978-3-540-39110-4.
5. Josh Aas «Understanding the Linux 2.6.8.1 CPU Scheduler» online publication: (http://citeseer.ist.psu.edu/aas05understanding.html) pp. 38.
6. O. Sename, D. Simon and D. Robert: "Feedback scheduling for real-time control of systems with communication delays" ETFA'03 9th IEEE International Conference on Emerging Technologies and Factory Automation, Lisbonne. Volume 2, 16-19 Sept. 2003 Page(s):454 - 461 vol. 2

7. S. Kirkpatrick and C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, Science, Vol 220, Number 4598, pages 671-680, 1983.

8. Уссермен Ф. Нейрокомпьютерная техника. - М.: Мир, 1992.

9. Костенко В.А., Калашников А.В. Исследование различных модификаций алгоритмов имитации отжига для решения задачи построения многопроцессорных расписаний// Дискретные модели в теории управляющих систем. Труды VII Международной конференции. М.: МАКС Пресс, 2006. - С.179-184.

10. John H. Holland «Adaptation in Natural and Artificial Systems» April 1992 7 x 9, 228 pp. ISBN-10: 0-262-08213-6, ISBN-13: 978-0-262-08213-6.

11. E.S.H. Hou, N. Ansari, H. Ren, "A Genetic Algorithm for Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 05, no. 2, pp. 113-120, Feb., 1994. ISSN 1045-9219

12. Michelle Moore, "An Accurate and Efficient Parallel Genetic Algorithm to Schedule Tasks on a Cluster," *ipdps*, p. 145a, International Parallel and Distributed Processing Symposium (IPDPS'03), 2003. ISBN: 0-7695-1926-1

13. Костенко В.А., Смелянский Р.Л., Трекин А.Г. Синтез структур вычислительных систем реального времени с использованием генетических алгоритмов// Программирование, 2000., №5, С.63-72. **(Kostenko V.A., Smeliansky R.L., and Trekin A.G. Synthesizing Structures of Real-Time Computer Systems Using Genetic Algorithms. Programming and Computer Software, Vol. 26, No. 5, 2000, pp. 281-288.)**

14. **Thomas H. Cormen**, **Charles E. Leiserson**, **Ronald L. Rivest**, and **Clifford Stein**. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. **ISBN 0-262-03293-7**.

15. P. Bjorn-Jorgensen, J. Madsen, "Critical path driven cosynthesis for heterogeneous target architectures," *codes*, p. 15, 5th International Workshop on Hardware/Software Co-Design (Codes/CASHE '97), 1997.

16. Chun-Hsien Liu, Chia-Feng Li, Kuan-Chou Lai, Chao-Chin Wu, "Dynamic Critical Path Duplication Task Scheduling Algorithm for Distributed Heterogeneous Computing Systems," *icpads*, pp. 365-374, 12th International Conference on Parallel and Distributed Systems - Volume 1 (ICPADS'06), 2006, ISBN: 0-7695-2612-8.

17. David Jackson, Quinn Snell, Mark Clement «Core Algorithms of the Maui Scheduler» **Lecture Notes in Computer Science Job Scheduling Strategies for Parallel Processing : 7th International Workshop, JSSPP 2001, Cambridge, MA, USA, June 16, 2001. Revised Paper, Volume 2221, pp. 87-102, 2001, ISSN: 0302-9743.**

18. **Julita Corbalan, Xavier Martorell and Jesus Labarta. "Improving Gang scheduling through job performance analysis and malleability" Proceedings of the 15th international conference on Supercomputing pp. 303 - 311 Sorrento, Italy 2001. ISBN:1-58113-410-X**

19. Helen D. Karatza «Perfomance Analysis of Gang Scheduling in a Distributed System under Processors

Failuries» «International Journal of Simulation Systems, Science and Technology» Volume 2, Number 1, pp. 14-23, 2001, ISSN: 1473-804x, ISSN: 1473-8031.

20. IBM LoadLeveler for AIX 5L using and Administrating Version 3 Release 1, pp.381-394, 2001.
21. А.В. Баранов, А.О. Лацис, М.Ю. Храмцов, С.В. Шарф «Руководство системного программиста (администратора) системы управления прохождением задач МВС-1000/М (версия 2.01)» http://www.jscc.ru/informat/1000MPrgGuide.zip.
22. http://parallel.ru/ [ru]
23. http://parallel.ru/vvv/index.html [ru]
24. http://cray.com/products/xt3/ [en]
25. http://cray.com/products/x1e/index.html [en]
26. http://www.citforum.ru/nets/protocols2/2_07_00.shtml [ru]
27. http://www.myri.com/myrinet/overview/index.html [en]
28. http://en.wikipedia.org/wiki/Symmetric_multiprocessing [en]
29. http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access [en]
30. http://en.wikipedia.org/wiki/Massively_parallel_processing [en]
31. http://en.wikipedia.org/wiki/Vector_processor [en]
32. http://novell.eureca.ru/0a/9.html [ru]
33. http://www.nersc.gov/nusers/resources/software/ibm/xlc.html [en]

34. http://rsusu1.rnd.runnet.ru/tutor/method/m1/page04.html [ru].
35. Гук М, Дисковая подсистема ПК. – Санкт-Петербург, 2001.
36. Интерфейс малой компьютерной системы (ИМКС) / Всесоюзный центр переводов научно-технической литературы и документации (ВЦП). Северо-Кавказский филиал. – Ростов-н/Д, 1989.
37. SCSI Interface: Product Manual. Vol. 1, 2 / Seagate.
38. SCSI-2 Specification.
39. IA-64 Application Developer's Architecture Guide, Intel Corp.,1999.
40. IA-64 Application Instruction Set Architecture Guide, Rev.1.0, Intel Corp., HP, 1999.
41. М. Кузьминский, "Открытые системы", 1999, N 5-6, стр.8.
42. SGI Power Challenge. Technical Report. SGI, 1996
43. http://www.citforum.ru/hardware/articles/ia64.shtml
44. http://www.citforum.ru/hardware/pc/scsi/
45. http://www.ixbt.com/storage/raids.html
46. http://parallel.ru/computers/reviews/raid-technology.html
47. http://ru.wikipedia.org/wiki/RAID
48. http://www.osp.ru/text/302/183797/
49. М. Кузьминский, *«Power4: легенда мира RISC»*. // Открытые системы, 2003, № 6.
50. J.M. Tendler, J.S. Dodson, J.S. Fields, H. Le, B. Sinharoy, «IBM e-server POWER4 System Microarchitecture». IBM White Paper, Oct. 2001.

51. J.M. Tendler, J.S. Dodson, J.S. Fields, H. Le, B. Sinharoy, «POWER4 system microarchitecture», IBM J. Res. Develop., 2002, v. 46, No. 1.

52. М. Кузьминский, *«Супер-чип для супер-ЭВМ»*. // Computerworld Россия, 1997, № 16.

53. IBM eServer pSeries performance and sizing. Technical White Paper, IBM, 2001.

54. D. Quinerto, M. Genty, S.K. На e.a., Performance and Tuning Consideration for the p690 in a cluster 1600; IBM eServer Cluster 1600 Hardware Planning, Installation and Service, GA22-7863-03, IBM, Dec. 2002.

55. H. Schulz, IBM eServer pSeries with the HPC Feature, IBM, Nov. 2001.

56. IBM eServer pSeries 690 Reliability, Availability, Serviceability (RAS), IBM, Sep. 2001.

57. Домашняя страница проекта Sun Grid Engine http://gridengine.sunsource.net/.

58. IBM XL C/C++ Compiler Reference. http://www.ibm.com/software/awdtools/ccompilers/

59. Palo Alto, Interprocedural optimization: eliminating unnecessary recompilation, Symposium on Compiler Construction, Proceedings of the 1986 SIGPLAN symposium on Compiler construction, pp. 58 – 67, ISSN:0362-1340.

60. Королёв Л. Н., Архитектура ЭВМ, изд. Научный мир, 2005, ISBN 5-89176-274-9.

61. Домашняя страница NetCDF http://www.unidata.ucar.edu/software/netcdf