ИСПОЛЬЗОВАНИЕ КЛАСТЕРНОЙ СИСТЕМЫ “OPENMOSIX” ДЛЯ ПОСТРОЕНИЯ РАСПРЕДЕЛЁННЫХ ВЫЧИСЛЕНИЙ

КАЦУБО ДМИТРИЙ ВЛАДИМИРОВИЧ

Минск 2003

СОДЕРЖАНИЕ

СПИСОК СОКРАЩЕНИЙ

1. SISD (Single Instruction – Single Data) – одиночный поток команд, одиночный поток данных
2. SIMD (Single Instruction – Multiple Data) – одиночный поток команд, множественный поток данных
3. MISD (Multiple Instruction – Single Data) – множественный поток команд, одиночный поток данных
4. MIMD (Multiple Instruction – Multiple Data) – множественный поток команд, множественный поток данных
5. SMP (Symmetric Multi Processors) – симметричные мультипроцессоры
6. MPP – системы с массовым параллелизмом
7. SCC (Scalable Computing Clusters) – масштабируемые вычислительные кластеры
8. SSI (Single System Image) – система, состоящая из одного образа (однообразная)
9. COW (Cluster of Workstations) – кластеры из рабочих станций
10. MPI (Message Passing Interface) – интерфейс для передачи сообщений
11. PVM (Parallel Virtual Machine) – параллельная виртуальная машина
12. openMosix (Open Multicomputer Operating System for UNIX) – открытая мультикомпьютерная операционная система для UNIX.
13. IPC (Inter Process Communication) – межпроцессное взаимодействие
14. LWP (Light Weight Process) – легковесный процесс
15. HWP (Heavy Weight Processes) – тяжеловесный процесс
16. PPM (Preemptive Process Migration) – приоритетная миграция процессов
17. UHN (Unique Home Node) – уникальный домашний узел
18. oMFS (openMosix File System) – файловая система openMosix
19. DFSA (Direct File System Access) – файловая система прямого доступа

ВВЕДЕНИЕ

В одной из наиболее известных классификаций параллельных ЭВМ, предложенной Филлином, вводятся понятия “поток команд” и “поток данных”. Согласно этой классификации имеется четыре больших класса ЭВМ:

1. SISD (Single Instruction – Single Data) – это последовательные ЭВМ, в которых выполняется единственная программа
2. SIMD (Single Instruction – Multiple Data) – выполняется единственная программа, но каждая команда обрабатывает массив данных (векторная форма параллелизма)
3. MISD (Multiple Instruction – Single Data) – несколько команд одновременно работают с одним элементом данных
4. MIMD (Multiple Instruction – Multiple Data) – одновременно и независимо друг от друга выполняется несколько программных ветвей, которые в определённые промежутки времени обмениваются данными

Существующие параллельные средства класса MIMD образуют три технических подкласса: симметричные мультипроцессоры (SMP), системы с массовым параллелизмом (MPP) и кластеры.

Симметричные мультипроцессоры используют принцип разделяемой памяти. В этом случае система состоит из нескольких однородных процессоров и массива общей памяти. Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью. Процессоры подключены к памяти с помощью общей шины или коммутатора. Аппаратно поддерживается когерентность кэшей. Наличие общей памяти сильно упрощает взаимодействие процессоров между собой, однако накладывает сильное ограничение на их число – не более 32 в реальных системах.

Системы с массовым параллелизмом содержат множество процессоров (обычно RISC) с индивидуальной памятью в каждом из них (прямой доступ к памяти других узлов невозможен), коммуникационный процессор или сетевой адаптер, иногда – жёсткие диски и/или другие устройства ввода-вывода. Узлы связаны через некоторую коммуникационную среду, например, высокоскоростную сеть. Общее число процессоров в реальных системах достигает нескольких тысяч.

Кластерные системы – более дешёвый вариант MPP систем, поскольку они также используют принцип передачи сообщений, но строятся из компонентов высокой степени готовности. Вычислительный кластер – это совокупность компьютеров, объединённых в рамках некоторой сети для решения одной задачи. В качестве вычислительных узлов обычно используются доступные на рынке однопроцессорные компьютеры, одно-, двух- или четырёхпроцессорные SMP-серверы.

В настоящее время малостоящие кластерные системы вытесняют традиционные суперкомпьютеры и мэйнфреймы, потому что они могут

предоставить хорошие решения для требовательных приложений. Компьютерные кластеры особенно подходят для маленьких и средних организаций, которые не могут позволить себе дорогих систем. Кластеры выполнены из обычных компонент, и поэтому их постепенный рост легко подстраивается под бюджет и потребности.

Чаще всего по количеству установленных систем выделяют три типа кластеров: кластеры с защитой от сбоев, кластеры с балансировкой нагрузки и высокопроизводительные вычислительные кластеры.

Кластеры с защитой от сбоев состоят из двух или более компьютеров, соединённых сетью и отдельным синхронизирующим соединением, которое используется для мониторинга того, что все службы задействованы, и в том случае если одна из служб даёт сбой, другой узел перенимает её выполнение.

Для **кластеров с балансировкой нагрузки** концептуальным является то, что когда узел становится доступным, кластер проверяет, какой из узлов является наименее загруженным, и посылает запрос к этому узлу. Фактически, большую часть времени кластер с балансировкой нагрузки действует как кластер с защитой от сбоев, но с дополнительной балансирующей функциональностью, и, как правило, содержит меньшее число узлов.

Последним типом кластеров являются **высокопроизводительные вычислительные кластеры**, сконфигурированные специально для того, чтобы обеспечивать центрам обработки данных крайне высокую производительность, которую они требуют. Этот тип кластеров также обладает некоторыми свойствами балансировки нагрузки, так как кластер пытается распределить процессы по узлам для увеличения производительности. И главным следствием из этого является то, что в случае когда процесс распараллеливается, то подпроцессы могут быть распределены по различным узлам, вместо того, чтобы выполняться один за другим.

Сегодня существуют две основные парадигмы для кластерных Linux окружений: openMosix (клон Mosix) (см. [MOSIX] и [oMosix]) и Beowulf (основан на MPI). В то время как параллелизация в решениях, основанных на MPI, требует явного кодирования с использованием специальных библиотек или директив, в openMosix можно придерживаться обычного Unix программирования, потому что кластерная функциональность обеспечивается прозрачным образом ядром операционной системы.

Чем более мощным становиться кластер из рабочих станций, тем более важным является мудрое использование его ресурсов. Плохая стратегия назначения заданий может вызвать сильно неуравновешенные нагрузки и своппинг машин, что наносит вред вычислительной мощности кластера. Ресурсы могут использоваться более эффективно, если кластер может заставить задания мигрировать – перемещаться прозрачно от одной машины к другой. Однако даже уравновешенные системы, которые могут переназначать

задания, всё равно могут извлечь выгоду от тщательно выбранной стратегии назначения. Миграция заданий привлекательна, потому что время поступления и количество запросов на ресурсы от прибывающих заданий непредсказуемы. Из-за этой непредсказуемости задания иногда будут назначаться на неоптимальную машину, и переназначение даёт системе ещё один шанс исправить такую ошибку. Интуитивно понятно, что способность мигрировать задания может привести к лучшей эффективности – то есть более быстрое время завершения для среднего задания. Так как не известно, где должно выполняться задание в любое заданное время, то стратегия переназначения также может делать ошибки. Например, расширение openMosix для ядра Linux позволяет такой вид прозрачной миграции заданий. Таким образом, openMosix включает в себя технологии, реализующие алгоритмы миграции заданий (см. [BAR1]).

Кластер openMosix, состоящий из компьютеров под управлением ОС Linux, есть то, что называется однообразной системой (SSI). В различных статьях неоднократно обсуждалось, что нельзя получить настоящий кластер до тех пор, пока вы не построите SSI. В SSI кластере пользователь не заботится о том, на каком узле он выполняет команды, и любая программа, которую он запускает, будет запущена на том узле, который сможет удовлетворить потребности программы наилучшим образом.

openMosix – это расширение ядра Linux. Поэтому для того, чтобы проинсталлировать openMosix, необходимо запустить специальный инсталляционный скрипт, который применит всё необходимые изменения к исходным кодам ядра Linux. Такой скрипт в дальнейшем будем называть «патч» (от англ. patch). Изменения касаются 3% всего исходного кода ядра, что не очень много (см. гл. 5.3).

Замечательной особенностью openMosix является то, что пользователь только запускает программу, а кластер решает, где её выполнить (если не оговорено обратное). Разработчики openMosix называют это свойство «разветвись- и-забудь» (fork-and-forget).

openMosix вносит новые мерки в масштабирование кластерных вычислений с использованием Linux. openMosix делает возможным конструирование высокопроизводительных масштабируемых кластерных систем из компьютеров широкого использования, причём масштабирование не приводит к затратам производительности. Главным преимуществом openMosix перед другими кластерными системами является её возможность отвечать во время выполнения на непредсказуемые и нестандартные запросы многих пользователей к ресурсам.

Отличительными свойствами выполнения приложений на openMosix являются адаптивная политика распределения ресурсов, симметрия и гибкость конфигурации. Комбинированный эффект этих свойств подразумевает то, что пользователь не должен знать текущего состояния использования ресурсов на различных узлах и даже их количества. Параллельные приложения могут выполняться, позволяя openMosix назначать и переназначать процессы на

наилучшие из возможных узлов, почти так же, как и SMP.

Тем не менее, есть области, в которых openMosix не сможет принести хоть какую-то пользу. Так, например, приложения, использующие потоки, должны оставаться в пределах одного узла.

Проект openMosix развивается в нескольких направлениях. Уже закончены работы по реализации мигрируемых сокетов (migratable sockets), которые должны снизить затраты на межпроцессное взаимодействие. Похожей оптимизацией является мигрируемые временные файлы, которые позволят удалённым процессам (например, компиляторам) создавать временные файлы на удалённых узлах. Основной концепцией этих оптимизаций является миграция большего числа ресурсов вместе с процессом для снижения затрат на удалённый доступ. Также ведутся разработки сетевой RAM, с помощью которой процесс сможет использовать память, доступную на нескольких узлах. Идея состоит в том, чтобы распределить данные процесса среди нескольких узлов и мигрировать (обычно маленькие) процессы к данным, нежели перемещать данные к процессам. Активно развивается проект по реализации мигрируемой разделяемой памяти (см. гл. 4).

В данной работе проведено исследование основных принципов работы openMosix и основ построения распределённых систем с её использованием.

1. ВВЕДЕНИЕ В OPENMOSIX

1.1. Что такое openMosix

openMosix – это программное обеспечение, которое было разработано для того, чтобы расширить ядро Linux возможностями кластерных вычислений. openMosix расшифровывается как Open **M**ulticomputer **O**perating **S**ystem for UN**IX**. Ядром openMosix являются адаптивные интерактивные алгоритмы, балансирующие нагрузку, управляющие памятью и файловым вводом-выводом. Они реагируют на изменения в использовании ресурсов кластера, например, неравномерную распределённую нагрузку или чрезмерный обмен с диском из-за недостатка свободной памяти на одном из узлов (см. [BAR5]). В таких случаях, openMosix инициирует перемещение процесса от одного узла к другому, чтобы сбалансировать нагрузку, или перемещение процесса на узел, который имеет достаточно свободной памяти, или уменьшение числа удалённых файловых операций ввода-вывода.

openMosix работает незаметно – его операции прозрачны для прикладных программ. Это означает, что пользователи могут выполнять последовательные и параллельные прикладные программы точно так же, как на SMP. Пользователи не должны заботиться о том, где процессы выполняются, ни беспокоиться о том, что делают другие пользователи. Вскоре после создания нового процесса, openMosix пытается назначать его лучшему доступному узлу на этот момент. openMosix контролирует все процессы, и, в случае необходимости, мигрирует процессы между узлами, чтобы максимизировать общую эффективность. Все это делается без изменения интерфейса Linux. Это означает, что вы продолжаете видеть (и контролировать) все ваши процессы, как будто они выполняются на узле, с которого вы вошли систему.

Алгоритмы openMosix децентрализованы – каждый узел является и хозяином для процессов, которые были созданы локально, и сервером для удалённых процессов, которые мигрировали с других узлов. Это означает, что узлы могут быть добавлены или удалены из кластера в любое время с минимальными беспокойством выполняющихся процессов. Другое полезное свойство openMosix – его алгоритмы для мониторинга, которые определяют быстродействие каждого узла, его нагрузку и свободную память, также как и производительность IPC и системы ввода-вывода для каждого процесса. Эта информация используется, чтобы принимать решения для распределения процессов, близкие к оптимальным (см. [BAR4]).

1.2. Прозрачность местоположения

В прошлом принцип прозрачности местоположения применялся к файловым системам, например, NFS. Эта идея была расширена в openMosix на распределение процессов между узлами кластера, чтобы улучшать общую

эффективность и сделать кластер более удобным для использования.

Системная модель образа openMosix основана на модели “домашнего узла” (UHN), в которой кажется, что все пользовательские процессы выполняются на узле, с которого пользователь вошёл в систему. Каждый новый процесс создаётся на том же самом узле, что и его родительский процесс. Процессы, которые мигрировали, взаимодействуют с пользовательской средой окружения через домашний узел пользователя, но там, где возможно, используют локальные ресурсы. Пока нагрузка узла, с которого пользователь вошёл в систему, остаётся ниже порогового значения, все пользовательские процессы ограничены этим узлом. Однако когда эта нагрузка начинает превышать порогового значения, некоторые процессы могут мигрировать к другим узлам (см. рис. 1).

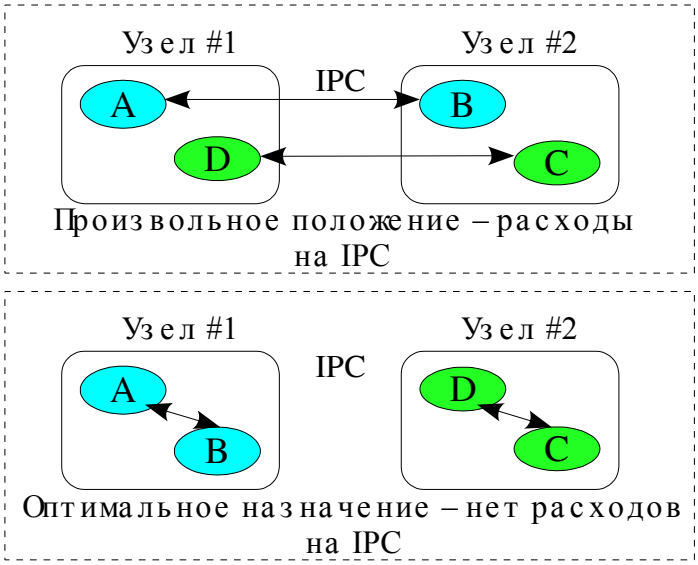
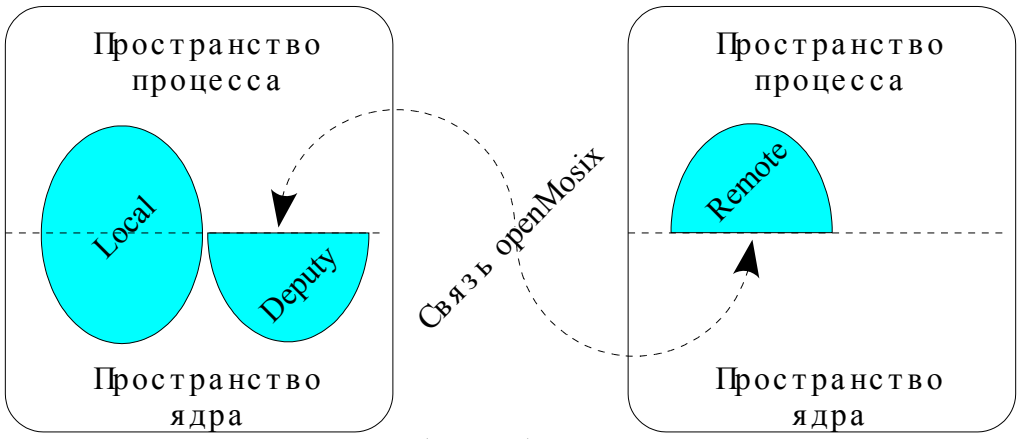


Рис. 1. Интеллектуальная балансировка нагрузки

Миграция происходит следующим образом. Процесс разбивается на две части: пользовательская часть и системная часть. Пользовательская часть переносится на удалённый узел, в то время как системная часть должна оставаться на домашнем узле, уникальном для всего кластера (UHN). Системную часть иногда называют представительской (deputy), поскольку она отвечает за разрешение большинства системных вызовов (syscalls). openMosix берёт на себя задачу обеспечения взаимодействия между этими двумя частями (см. рис. 2).



- Пользовательский контекст (Remote) не зависим от узла – может мигрировать
- Системный контекст (Deputy) зависим от узла – должен оставаться на домашнем узле
- Связь для синхронных (syscalls) и асинхронных (сигналы, события) вызовов

Рис. 2. Разбиение процесса и приоритетная миграция

Определение оптимального местоположения для задания – сложная проблема. Наибольшая сложность состоит в том, что ресурсы, доступные на рабочих станциях кластера, являются гетерогенными. В действительности, затраты на память, CPU, взаимодействие между процессами и т.д. не сравнимы. Они даже не измеримы в одних и тех же единицах: ресурсы взаимодействия измеряются в терминах пропускной способности, память – в терминах размерности, а CPU – в терминах циклов. Естественная “жадная” стратегия, балансирующая ресурсы среди всех машин, также точно не определена.

В [AMIR1] представлена новая стратегия назначения заданий, основанная на “экономических” принципах и конкурентном анализе. Эта стратегия даёт возможность управлять гетерогенными ресурсами способом, близким к оптимальному. Ключевая идея этой стратегии состоит в том, чтобы преобразовать общее использование нескольких гетерогенных ресурсов, таких как память и CPU, в единую гомогенную “стоимость”. Тогда задания назначаются на ту машину, где они имеют самую низкую стоимость.

1.3. Файловая система прямого доступа (DFSA)

Поддержка файловой системы прямого доступа (DFSA) расширяет способность мигрировавшего процесса выполнять некоторые операции ввода-вывода на текущем узле. Эта поддержка уменьшает потребность процессов, ориентированных на ввод-вывод, связываться с их домашним узлом, таким образом позволяя этим процессам более свободно мигрировать между узлами кластера, например, для балансировки нагрузки и обеспечения параллельных файловых операций ввода-вывода.

DFSA работает поверх любых подходящих общедоступных всему кластеру файловых систем, которые соответствуют следующим требованиям:

- Файловая система(-ы) смонтирована в той же точке монтирования всеми узлами;
- Идентификаторы пользователей/групп являются идентичными среди всех узлов кластера.

В настоящее время GFS и файловая система openMosix (oMFS) отвечают стандартам DFSA (см. [AMAR1]). Обе файловые системы делают все каталоги и обычные файлы доступными для всех узлов по всему кластеру openMosix, обеспечивая согласованность во время просмотра файлов с различных узлов, то есть, такую же согласованность, как если бы весь доступ к файлу был произведён с того же самого узла. В гл. 3.4 приведено сравнение кластерных файловых систем с традиционными.

1.4. Масштабируемость

openMosix может поддерживать конфигурации с большим числом компьютеров с минимальными накладными расходами на масштабирование, ухудшающих эффективность. Low-end конфигурация может включать несколько PC, которые связаны локальной сетью Ethernet, в то время как большая конфигурация может включать автоматизированные рабочие места и сервера, которые связаны более высокоскоростной локальной сетью, например, FastEthernet. High-end конфигурация может включать большое количество SMP и не-SMP автоматизированных рабочих мест и серверов, которые связаны высокоэффективной локальной вычислительной сетью, например, Gigabit-Ethernet (или Myrinet).

2. АЛГОРИТМЫ МИГРАЦИИ И РАЗДЕЛЕНИЯ РЕСУРСОВ

2.1. Алгоритм миграции

openMosix – это инструмент для ядер Linux, включающий в себя адаптивные алгоритмы разделения ресурсов. Он позволяет нескольким одно- и мультипроцессорным узлам работать в тесном взаимодействии. Адаптивные алгоритмы спроектированы так, чтобы отвечать ежеминутным вариациям при использовании ресурсов всеми узлами. Это достигается за счёт миграции процессов от одного узла к другому, приоритетно и прозрачно, для балансировки нагрузки и предотвращения своппинга памяти. Их целью является улучшение общекластерной производительности и создание удобной мультипользовательской среды с разделением времени для выполнения как последовательных, так и параллельных приложений. Стандартным окружением времени выполнения openMosix является SCC, в котором общекластерные ресурсы доступны всем узлам. Отключая автоматическую миграцию процессов, пользователь может переключиться на обычный SCC или однопользовательский режим (см. гл. 5.2).

Технология openMosix состоит из двух частей: механизм приоритетной миграции процессов (PPM) и набор механизмов для адаптивного разделения ресурсов. Обе части реализованы на уровне ядра и поэтому являются полностью прозрачными для уровня приложения, что позволяет создавать распределённые приложения без необходимости использования сторонних библиотек или технологий, а также даёт возможность использовать ресурсы кластера уже написанными программами, не спроектированных специально для работы на кластере, без их модификации и/или перекомпиляции.

PPM может мигрировать любой процесс в любое время на любой доступный узел. Обычно миграция базируется на информации, обеспечиваемой одним из алгоритмов разделения ресурсов, но пользователь может переопределить эту автоматическую системную миграционную политику и мигрировать процессы самостоятельно. Такая ручная миграция может быть инициирована синхронно самим процессом или явным запросом от другого процесса того же пользователя или суперпользователя. Ручная миграция процессов может быть полезна для реализации специфической политики миграции или для тестирования планировочных алгоритмов. Также суперпользователь имеет дополнительные привилегии касательно PPM, такие как тонкая настройка балансировки и настройка подсистемы ограничений (loadlimit, miggroup, – см. гл 6.1.1).

С каждым процессом ассоциирован идентификатор уникального домашнего узла (UHN), на котором он был запущен. Обычно, это есть узел, с которого пользователь вошёл в систему. Для PVM это есть узел, где была порождена задача демоном PVM. Стандартной моделью openMosix является однообразная модель SCC, в которой кажется, что каждый процесс выполняется на своём домашнем узле, и все процессы одной сессии пользователя разделяют

окружение времени выполнения домашнего узла. Мигрированный процесс по возможности использует ресурсы удалённого узла, но взаимодействует со средой выполнения через домашний узел. Так, например, предположим, что пользователь запустил несколько процессов, и часть из них мигрировала. Тогда если пользователь просмотрит вывод команды ps, то он увидит все процессы, включая те, которые выполняются удалённо. При этом суммарное использование процессора может оказаться больше 100%, а использование памяти – больше, чем размер физически установленной на узел памяти. Если один из мигрировавших процессов прочитает текущее время, то есть, вызовет gettimeofday(), то он получит значение текущего времени на UHN.

PPM – главный инструмент алгоритмов управления ресурсами. Его главой целью является максимизировать производительность за счёт эффективного использования общесетевых ресурсов. Уровнем детализации сетевого распределения openMosix является процесс. Пользователь может выполнять параллельные приложения, инициируя запуск нескольких процессов на узле, и затем позволяя системе назначать эти процессы на лучшие доступные узлы на данный момент. Если во время выполнения процесса становятся доступными новые ресурсы, то алгоритмы разделения ресурсов спроектированы таким образом, чтобы воспользоваться этими новыми ресурсами за счёт переназначения процессов между узлами. Эта возможность назначения и переназначения заданий является особенно важной для простоты использования и для обеспечения эффективной мультипользовательской среды выполнения с разделением времени.

openMosix не имеет центрального места управления или отношений управляющий/подчинённый между узлами: каждый узел может оперировать как автономная система и принимать управленческие решения независимо. Эта схема делает возможным создание динамической конфигурации, в которой узлы могут присоединяться и покидать сеть с минимальными сложностями. Дополнительно это увеличивает масштабируемость и гарантирует, что система будет работать хорошо в больших конфигурациях точно так же, как и в маленьких, а также исключает принцип “единой точки сбоя” (single point of failure). Масштабируемость достигается за счёт внедрения фактора случайности в алгоритмы управления системой, когда каждый узел принимает решения, базируясь на частичном знании состояния других узлов, и не пытается определить состояние всех узлов кластера или кластера в целом. Например, согласно вероятностному алгоритму рассеивания информации, каждый узел кластера посылает через регулярные интервалы времени информацию о своих (локально) доступных ресурсах случайному подмножеству всех узлов. В то же время, он поддерживает небольшое окно недавно полученной информации. Эта схема поддерживает масштабирование, равномерное рассеивание информации и динамическую конфигурацию (см. т.ж. гл. 2.4.6).

Как было сказано выше, для миграции процесс разбивается на две части: пользовательский контекст и системный контекст. Пользовательский

контекст состоит из кода программы, стека, данных, карт памяти и регистров процесса. Таким образом он инкапсулирует процесс, выполняющийся на уровне пользователя. Системный контекст содержит описание ресурсов, которые закреплены за данным процессом, и стек ядра для выполнения системных вызовов. Таким образом он инкапсулирует процесс, когда тот выполняется на уровне ядра. Так как он содержит машинно-зависимую часть системного контекста процесса, то, следовательно, он должен оставаться на UHN процесса. Поскольку вся память пространства пользователя находится на удалённом узле, представительский контекст не имеет своей карты памяти. Вместо этого он разделяет карту памяти ядра таким же образом, как и поток ядра.

Интерфейс между пользовательским контекстом и системным контекстом ядра хорошо определён, а следовательно, можно вмешаться в каждое такое взаимодействие и перенаправить его посредством сети. Это реализовано на уровне связи (link layer) как специального коммуникационного канала для взаимодействия. На рис. 2 показаны два процесса, которые разделяют UHN: левый процесс – это обычный Linux процесс, в то время как правый процесс разделён, и удалённый контекст мигрировал на другой узел.

Время миграции состоит из фиксированной компоненты для установки нового кадра процесса на удалённом узле, и линейной компоненты, пропорциональной количеству страниц памяти для перемещения. Для уменьшения накладных расходов на миграцию перемещаются только таблицы страниц и “грязные” страницы процесса. Остальные страницы перемещаются по тому же принципу, что и страницы, выгруженные в область подкачки: в случае, если при выполнении процесса требуемая страница отсутствует, то возникает исключение, страница пересылается удалённому узлу, и выполнение процесса продолжается с прерванной точки.

Таки образом, при выполнении процесса в openMosix прозрачность расположения достигается за счёт перенаправления машиннозависимых системных вызовов на UHN. Системные вызовы являются синхронной формой взаимодействия между двумя контекстами процесса. Все системные вызовы, производящиеся процессом, прерываются и, если они являются машиннонезависимыми, выполняются локально (на удалённом узле). В противном случае они направляются на UHN, где выполняются от имени приложения. Результат выполнения возвращается на удалённый узел, который затем продолжает выполнять код на уровне пользователя.

Другой формой взаимодействия между контекстами процесса являются доставка сигналов и события пробуждения процесса, такие как прибытие сетевых данных. Эти события требуют асинхронного взаимодействия между представительским и удалённым контекстами. В типичном сценарии ядро на UHN информирует представительский контекст о событии, который проверяет, необходимо ли предпринять какие-либо действия, и, если да, то информирует удалённый. Удалённый контекст проверяет коммуникационный канал на наличие сообщений об асинхронных событиях перед тем, как

продолжать выполнение на уровне пользователя (т.е. каждый раз, когда планировщик переключается на выполнение этого процесса).

При многих действиях с ядром, таких как выполнение системных вызовов, необходимо перемещать данные между пространством пользователя и ядра. Обычно это происходит при помощи примитив ядра copy_to_user(), copy_from_user(). В openMosix любые операции с памятью ядра, которые затрагивают доступ к пространству пользователя, требуют взаимодействия представительского и удалённого контекстов для пересылки необходимых данных.

Затраты на взаимодействие из-за удалённых операций копирования, которые могут повторятся несколько раз в пределах одного системного вызова, могут быть достаточно существенными в основном из-за задержек сети. Для того чтобы уменьшить чрезвычайное количество удалённых операций копирования, был реализован специальный кэш, который уменьшает число взаимодействий посредством предвыборки наибольшего количества данных во время первоначального запроса на системный вызов, в то же время буферизируя частичные данные на UHN для передачи удалённому контексту при завершении системного вызова.

Для того, чтобы предотвратить удаление или перекрытие отображённых в память файлов (для загрузки страниц по требованию), представительский контекст, в отсутствие карты памяти, содержит специальную таблицу таких файлов, которые отображены на удалённую память. Пользовательские регистры удалённого процесса обычно подответственны удалённому контексту, тем не менее, любой регистр или их комбинация может временно перейти в принадлежность представительскому для манипуляций.

Удалённые процессы не доступны другим процессам, выполняющимся на этом узле (локально или мигрировавших с других узлов) и наоборот. Они не принадлежат какому-либо определённому пользователю на удалённом узле, где они выполняются, и не могут посылать сигналы или каким-либо иным образом манипулировать локальными процессами. Единственное, что администратор может заставить мигрировать данный процесс с этого узла.

Иногда процессу нужно выполнить некоторые фукнции openMosix, в то время как он находится в логическом состоянии останова или сна. Такие процессы выполнят функции openMosix “в состоянии сна”, после чего продолжат сон, если тем временем не произошло события, которого они ждали. Примером является миграция процесса, возможно происходящая, когда процесс спит. Для этих целей openMosix сохраняет логическое состояние, описывающее, как другие процессы будут видеть этот процесс, которое является противоположным его непосредственному состоянию.

Подход, описанный в этой главе, является устойчивым, и даже большие изменения в ядре не влияют на него. Он практически не полагается на машиннозависимые свойства ядра, и поэтому ничто не мешает перенести его на другие архитектуры. Недостатком этого подхода являются, очевидно, дополнительные накладные расходы на выполнение системных вызовов.

Дополнительные расходы возникают при файловых и сетевых операциях. Так, например, все сетевые сокеты создаются на UHN, что является причиной усиления взаимодействия, если процесс мигрирует с UHN. Для преодоления этой проблемы разрабатываются мигрируемые сокеты, которые могут перемещаться вместе с процессом, делая возможным прямое соединение с мигрировавшим процессом. В настоящее время эти расходы могут быть существенно уменьшены за счёт первоначального распределения взаимодействующих процессов на различные узлы, например, при помощи PVM. Если система становится несбалансированной, алгоритмы openMosix переназначают процессы для увеличения производительности (см. [BAR6]).

2.1.1. Сбор информации о процессе

Статистика о поведении процесса собирается периодически, например, при системных вызовах и каждый раз, когда процесс получает доступ к пользовательским данным. Эта информация используется для оценки того, что процесс должен мигрировать с UHN. Эта статистика устаревает через некоторое время для того, чтобы приспособиться к процессам, изменяющих свой профиль выполнения. Она также полностью очищается при системном вызове execve() (который замещает процесс новым), потому что, скорее всего, процесс изменит свой характер.

Каждый процесс имеет некоторый контроль над сбором и устареванием своей статистики. Например, процесс может приостановить своё выполнение, зная, что его характеристики должны вот-вот измениться, или он может циклически переключаться, комбинируя вычисления и ввод-вывод.

2.1.2. Ограничения на миграцию

Некоторые функции ядра не совместимы с разделением контекста процесса. Очевидными примерами являются непосредственное манипулирование устройствами ввода-вывода, взаимодействие напрямую при помощи привилегированных инструкций с шиной или прямой доступ к устройству памяти. Другие примеры включают в себя использование потоков или приложений реального времени (в последнем случае невозможно гарантировать их корректное функционирование после миграции). Процессы, которые используют эти механизмы, автоматически замыкаются на домашнем узле. Если процесс уже мигрировал, то он высылается обратно на UHN. При этом `/proc/[PID]/cantmove` содержит имя конфликтного системного вызова (см. гл. 6.1.1).

2.2. Алгоритмы разделения ресурсов

Основными алгоритмами разделения ресурсов openMosix являются алгоритмы балансировки нагрузки и алгоритмы распределения памяти[1] (или

1 Не имеет ничего общего с проблемой распределённой разделяемой памяти (distributed shared memory)

предотвращения своппинга памяти).

Динамический алгоритм балансировки нагрузки непрерывно пытается уменьшить разность нагрузки между парами узлов, мигрируя процессы от более загруженных узлов к менее загруженным. Эта схема является децентрализированной: на всех узлах работает один и тот же алгоритм, и уменьшение разности загрузки выполняется независимо любой парой узлов. Количество выполняемых процессов и быстродействие узла являются важными факторами алгоритма балансировки нагрузки. Этот алгоритм отвечает на изменения нагрузки узла или на изменение характеристик времени выполнения процесса. Он остаётся преобладающим то тех пор, пока система не испытывает чрезмерного недостатка других ресурсов, таких как свободная память или свободные слоты для процессов.

Второй алгоритм – алгоритм распределения (предотвращения истощения) памяти – приспособлен для того, чтобы размещать максимальное число процессов в общекластерной RAM для избежания, насколько возможно, засорения памяти или выгрузки страниц памяти в область подкачки. Этот алгоритм включается, когда узел начинает интенсивный обмен страницами с областью подкачки из-за нехватки памяти. В этом случае, этот алгоритм перекрывает алгоритм балансировки нагрузки и пытается мигрировать процесс на узел с достаточным количеством свободной памяти, даже если эта миграция приведёт к неуравновешенному распределению.

Несколько позже в openMosix был внедрён новый алгоритм для выбора узла, на котором должен выполняться процесс. Математическая модель этого планировочного алгоритма пришла из области экономических исследований. Этот алгоритм пытается уравновесить различия между различными ресурсами основываясь на экономических принципах и конкурентном анализе. Эта экономическая стратегия обеспечивает унифицированную алгоритмическую основу для распределения вычислительных, коммуникационных ресурсов, памяти и ресурсов ввода-вывода. Она позволяет разрабатывать близкие к оптимальным online алгоритмы для распределения и совместного использования этих ресурсов.

2.3. Метод оценивания стоимости для назначения и переназначения заданий

2.3.1. Построение модели

Сформулируем цель как улучшение производительности кластера из *n* машин, где машина *i* имеет процессорные ресурсы скоростью $r_c(i)$ и ресурсы памяти размером $r_m(i)$. Мы абстрагируемся от других ресурсов, ассоциированных с машиной, хотя эта структура может быть расширена для управления дополнительными ресурсами.

Есть некоторая последовательность поступающих заданий, которые должны быть назначены этим машинам. Каждое *j* задание определяется тремя параметрами:

- время прибытия *a(j)*;
- количество процессорного времени *t(j)*, которое требуется для него;
- количество памяти *m(j)*, которое требуется для него.

Предполагаем, что *m(j)* известно, когда поступает задание, а *t(j)* – нет. Задание должно быть назначено машине непосредственно после поступления и может быть позже перемещено от одной машины к другой. Пусть *J(t,i)* – множество заданий на машине *i* в момент времени *t*. Тогда загрузка процессора и памяти на этой машине соответственно равны:

$l_c(t,i)=|J(t,i)|$ (мощность множества *J(t,i)*) и

$$l_m(t,i)=\sum_{j\in J(t,i)} m(j)$$

Предполагаем, что когда у машины заканчивается память, то она замедляется на некоторый мультипликативный фактор *s* из-за сброса страниц на диск. Тогда эффективная процессорная нагрузка машины *i* в момент *t*:

$$L(t,i)=\begin{cases} l_c(t,i)=|J(t,i)|, если\, l_m(t,i)\leqslant r_m(i) \\ l_c(t,i)*s, если\, l_m(t,i)>r_m(i) \end{cases}$$

Для простоты также предполагаем, что все машины выполняют задания “справедливо”, то есть в момент времени *t* каждое задание на машине *i* получит *1/L(t,i)* процессорного времени. Поэтому время завершения задания *c(j)* удовлетворяет следующему уравнению:

$\int_{a(j)}^{c(j)} \frac{r_c(i)}{L(t,i)}dt=t(j)$, где *i* – это номер машины, на которой выполняется задание в данный момент времени *t*. Если *L(t, i) = 1*, то из последнего выражения следует, что $c(j)-a(j)=\frac{t(j)}{r_c(i)}$, то есть, если на процессоре выполняется единственное задание, то реальное время его выполнения совпадает с временем выполнения на процессоре. Нетрудно видеть, что если *L(t, i) > 1* , то

$$c(j)-a(j)>\frac{t(j)}{r_c(i)} \quad .$$

Величина $\frac{c(j)-a(j)}{t(j)}$ называется замедлением задания. Необходимо разработать метод назначения заданий и/или переназначения, который минимизирует среднее замедление по всем заданиям.

Будем оценивать эффективность наших online алгоритмов их сравнительным коэффициентом, измеренного по отношению к производительности оптимального offline алгоритма. Online алгоритм *ALG* называется *c*-сравнимым, если для любой входной последовательности *I* следует, что $ALG(I) < c\ OPT(I) + \alpha$, где *OPT* – оптимальный offline алгоритм, α является константой.

2.3.2. Основные определения

Теоретическая часть этого раздела сосредоточится на том, как минимизировать максимальное использование различных ресурсов системы, другими словами, как наилучшим способом сбалансировать нагрузку системы. Один алгоритм для выполнения этого, описанный в [AWER1], доказывает свою практическую полезность даже тогда, когда наша цель состоит в том, чтобы минимизировать среднее квадратичное замедление, которое соответствует уменьшению суммы квадратов нагрузок.

В подготовке к обсуждению алгоритма исследуем проблему минимизации с тремя различными моделями машин и двумя различными видами заданий.

Три модели машин:

1. Идентичные машины. Все машины идентичны, и скорость выполнения задания на данной машине определена только нагрузкой машины.
2. Связанные машины. Машины идентичны за исключением того, что некоторые из них имеют различные скорости – в модели выше они имеют различные значения r_c, и память, связанная с этими машинами, игнорируется.
3. Несвязанные машины. Много различных факторов могут влиять на эффективную нагрузку машины и время завершения заданий, выполняющихся на них. Эти факторы известны.

Два возможных вида заданий:

1. Постоянные задания. Как только задание поступает на машину, оно останется там навсегда.
2. Временные задания. Задания покидают систему, когда они получили некоторое количество процессорного времени. Также исследуем связанную проблему, называемую проблемой online маршрутизации.

2.3.3. Идентичные и связанные машины

Сейчас предположим, что никакие переназначения не возможны, и что единственный ресурс является процессорное время. Наша цель поэтому

состоит в том, чтобы минимизировать максимальную нагрузку CPU. Когда машины идентичны и никакие другие ресурсы не существенны, жадный алгоритм работает хорошо. Этот алгоритм для назначения заданий назначает следующее задание на машину с минимальной текущей нагрузкой CPU. Если машины идентичны, и никакие другие ресурсы не существенны, жадный алгоритм имеет сравнительный коэффициент $c = 2 - \frac{1}{n}$ (см. [BART1]).

Когда машины связаны, задания постоянны, и никакие другие ресурсы не существенны, эффективен алгоритм Аспнеса (см. [ASPN1]). Определим *OPT* как нагрузку оптимального offline алгоритма; аппроксимация *OPT* даётся в [ASPN1]. Этот алгоритм назначает каждое прибывающее задание на самую медленную машину с результирующей нагрузкой ниже *2 * OPT*. Если *OPT* известна, этот алгоритм имеет конкурентный коэффициент 2. Чтобы аппроксимировать *OPT*, можно использовать методику дублирования.

Для несвязанных машин и временных заданий без переназначений не известно никакого алгоритма с конкурентным коэффициентом лучше, чем *n*.

2.3.4. Несвязанные Машины

Рассматриваемый алгоритм – это алгоритм для несвязанных машин и постоянного назначения заданий, основанный на показательной функции от "стоимости" машины с данной нагрузкой. Этот алгоритм назначает каждое задание на машину, чтобы минимизировать общую стоимость всех машин в кластере. Более точно, положим:

- a – константа, $1 < a < 2$;
- $l_i(j)$ – нагрузка машины i перед назначением задания j;
- $p_i(j)$ – задание j для нагрузки, которое будет добавлено к машине i.

Online алгоритм назначит j на машину i, что минимизирует граничную стоимость:

$$H_i(j) = a^{l_i(j)+p_i(j)} - a^{l_i(j)}$$

Этот алгоритм сравним с *O(log n)* для несвязанных машин и постоянных заданий. Работа, представленная в [AWER2], расширяет этот алгоритм и коэффициент сравнения на временные задания, используя до *O(log n)* переназначений на задание. Переназначение перемещает задание с его предварительно назначенной машины на новую машину. При наличии переназначений положим:

- $h_i(j)$ – нагрузка машины i перед тем, как j было в последний раз назначено на i.

Когда все задания завершились, алгоритм в [AWER2] проверяет "условие устойчивости" для каждого задания j и каждой машины M. Это условие устойчивости, где i означает машину, на которой в настоящее время находится задание j, есть:

$$a^{h_i(j)+p_i(j)} - a^{h_i(j)} \leqslant 2\ (a^{l_M(j)+p_M(j)} - a^{l_M(j)})$$

Если это условие устойчивости не удовлетворено некоторым заданием j, алгоритм переназначает j на машину M, что минимизирует $H_M(j)$.

2.3.5. Online маршрутизация виртуальных каналов

Рассматриваемый алгоритм минимизирует максимальное использование единственного ресурса. Чтобы расширить этот алгоритм на несколько ресурсов, мы исследуем связанную с данной проблему online маршрутизации виртуальных каналов. Коротко приведём объяснение того, что эта проблема является соответствующей данной. В этой проблеме нам даны:

- взвешенный граф $G(V, E)$, с объёмом $u(e)$ для каждого ребра e;
- предельно допустимая нагрузка mx;
- последовательность независимых запросов $(s_j, t_j, p_j: E \rightarrow [0, mx])$, прибываемых в произвольное время; s_j и t_j – исходящий и приёмный узлы, $p(j)$ – требуемая ширина пропускания.

Запрос, который назначен на некоторый путь P от источника к приёмнику, увеличивает нагрузку l_e на каждом ребре e из P на величину $p_e(j) = \frac{p(j)}{u(e)}$.

Наша цель состоит в том, чтобы минимизировать максимальную перегрузку связей, которая является отношением ширины пропускания, выделенной на связь, к её объёму.

Минимизация максимального использования CPU и памяти, где использование памяти измеряется в доле используемой памяти, может быть сведено к проблеме online маршрутизации. Это сведение производится следующим образом: создадим два узла $\{s, t\}$ и n непересекающихся путей, состоящих из одного ребра, с двумя вершинами из s в t (см. рис. 3). Каждая машина i представляет собой некоторый путь e с ребром памяти объёмом $r_m(i)$ и ребром CPU с объёмом $r_c(i)$. Каждое задание j есть запрос с источником s, сливом t и функцией p, которая отображает рёбра памяти на требования к памяти для задания и рёбра CPU – на 1. Максимальная перегрузка связи есть большее из максимальной нагрузки CPU и максимального использования памяти.

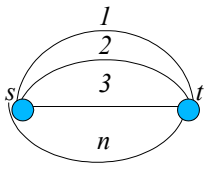


Рис. 3. Граф маршрутов

Рассматриваемый алгоритм расширен в [ASPN1], чтобы обращаться к проблеме online маршрутизации. Алгоритм вычисляет граничную стоимость для каждого возможного пути P от s_j к t_j следующим образом:

$$H_p(j) = \sum a^{l_e + p_e(j)} - a^{l_e}, \forall e \in P$$

и назначает запрос j на путь P, который даёт минимальную граничную стоимость.

Этот алгоритм сравним с $O(\log n)$ (см. [ASPN1]). При помощи сведения этот алгоритм порождает алгоритм для управления гетерогенными ресурсами,

который является сравнимым с *O(log n)* при максимальном использовании каждого ресурса.

2.3.6. Методика практического исследования

Для каждой машины в кластере из *n* машин, с ресурсами $r_1 \ldots r_k$, мы определяем стоимость это машины как:

$$\sum_{i=1}^{k} f(n, использование\, r_i)$$

где *f* – некоторая функция. На практике выберем *f* так, чтобы эта сумма была равна:

$$\sum_{i=1}^{k} n^{\frac{использованное\, r_i}{макс.\, использование\, r_i}}$$

Граничная стоимость назначения задания на данную машину – это количество, на которое эта сумма увеличивается, когда задание назначено на эту машину. Подход “стоимости возможности” (“opportunity cost”) для распределения ресурсов назначает задания на машины способом, который минимизирует эту граничную стоимость.

В этом разделе мы заинтересованы только двумя ресурсами, CPU и памятью, и мы игнорируем другие рассуждения. Следовательно, вышеупомянутая теория подразумевает, что при логарифмически большем количестве памяти, чем у оптимального online алгоритма, данный алгоритм достигнет максимального замедления в пределах *O(log n)* от максимального замедления оптимального алгоритма.

Это не гарантирует, что алгоритм, основанный на данном, будет конкурентным в его среднем замедлении по всем процессам. Это также не гарантирует, что такой алгоритм будет совершеннее, чем существующие методы. Следующим шагом мы должны проверить, что этот алгоритм фактически совершеннее, чем существующие методы, на практике.

Ресурс памяти легко транслируется в модель ресурсов данного алгоритма. Стоимость использования некоторого объёма памяти на машине есть n^u, где *u* – пропорциональное использование памяти ($u = \frac{используемая\, память}{всего\, памяти}$). Для ресурса CPU мы должны знать максимальную возможную нагрузку. Предположим, что *L* – самая маленькая целочисленная степень двойки, большая, чем самая большая нагрузка, которую мы наблюдали в любое заданное время, – является максимально возможной нагрузкой. Это предположение, будучи неточным, не меняет сравнительный коэффициент алгоритма.

Стоимость CPU и нагрузки памяти для заданной машины, используя наш метод, есть:

$$n^{\frac{\text{использованная память}}{\text{всего памяти}}} + n^{\frac{\text{загрузка CPU}}{L}}$$

Будем назначать или переназначать задания так, чтобы минимизировать сумму затрат на всех машинах в кластере.

Для того, чтобы исследовать поведение этого подхода “стоимости возможности”, оценим четыре различных метода для назначения задания:

1. PVM. PVM – очень популярная система метавычислений для систем без приоритетного перемещения процессов. Если пользователь системы специально не вмешивается, PVM назначает задания машинам, использующих строгую циклическую стратегию. Она не переназначает задания после того, как они начинают выполнение.
2. Усовершенствованный PVM. Усовершенствованный PVM использует стратегию, основанную на стоимости возможности, которая назначает каждое задание на машину, для которой задание имеет самую маленькую граничную стоимость. Как и для PVM, начальные назначения постоянны.
3. openMosix. openMosix – расширение для ядра Linux, позволяющее системе перемещать процессы от одной машины к другой без прерывания их работы. openMosix использует усовершенствованную стратегию балансировки нагрузки, которая также пытается сохранить некоторую память свободной на всех машинах. openMosix не всезнающий: когда система обменивается информацией о процессе при подготовке к возможному переназначению процесса, каждая машина контактирует с ограниченным набором других машин (см. гл. 2.1).
4. Усовершенствованный openMosix. Усовершенствованный openMosix – это стратегия, основанная на стоимости возможности, предназначенная для использования на системах (таких как кластеры openMosix), которые могут приоритетно перемещать процессы. Она назначает или переназначает задания, чтобы минимизировать сумму затрат на всех машинах. Расширенный openMosix имеет те же самые ограничения на свои “знания”, как нерасширенный openMosix.

2.3.7. Экспериментальные результаты

Первое испытание данного алгоритма было эмуляция Java задач для четырёх методов (пере)назначения заданий, упомянутых выше. Были сделаны следующие предположения:

1. Кластер содержит шесть машин со следующими характеристиками:

Тип машины	Количество таких машин	Скорость обработки	Установленная память
Pentiom Pro	3	200 МГц	64 Мб
Pentium	2	133 МГц	32 Мб
Laptop, Pentium	1	90 МГц	24 Мб

Таблица 1. Машины в эмулируемом кластере

2. Для каждого входящего задания, положим, что *r* и *m* – независимо сгенерированные случайные числа между 0 и 1. Типичный процесс будет требовать *2/r* секунд CPU и (*1/m*)% памяти Pentium Pro. Распределение основано на наблюдениях за процессами в реальной жизни. Приблизительно 95% всех заданий – задания с единственным процессом, соответствующие этой конфигурации. Поскольку данная система – система метавычислений, предполагается, что 5% всех заданий – большие параллельные задания, использующие мощность метавычислений. Эти задания содержат от 1 до 20 идентичных процессов, требующих *20/r* секунд CPU и (*1/m*)% памяти Pentium Pro. Чтобы сделать эмуляцию полной, предполагается, что никакой процесс не требует больше чем 100% памяти Pentium Pro, никакой процесс из числа заданий с единственным процессом не требует больше чем 1,000 секунд CPU, и никакой процесс из числа больших параллельных заданий не требует больше чем 10,000 секунд CPU.
3. Задания прибывают в произвольном порядке в течение первых 1,000 моделируемых секунд. Приблизительно одно задание прибывает каждые 10 секунд. Можно наблюдать, что для каждого из этих четырёх методов имеются примеры эмуляций, когда система была перегружена, недогружена и загружена как обычно, и примеры, где система переходила из любого из этих состояний в любое другое.
4. Когда использование памяти машины превышает объём памяти, предполагается что эта машина находится в режиме интенсивной подкачки. Это замедляет машину на коэффициент *t*, который был аппроксимирован постоянным множителем, равным 10. То есть, при эмуляции каждое задание требовало в 10 раз больше времени на машине с интенсивной подкачкой, как это бы потребовалось на машине со свободной памятью.

При каждом выполнении все четыре метода использовали один и тот же сценарий, в котором те же самые задания прибывали с той же самой скоростью.

2.3.8. Результаты эмуляции Java

Каждое выполнение возвращало среднее замедление по всем заданиям, также как и некоторую информацию относительно непосредственно сценария. Эти результаты были оценены двумя различными способами.

Самый важный интерес представляет общее замедление, испытанное каждым из этих четырёх методов. Среднее замедление при выполнении – это невзвешенное среднее число всех результатов эмуляции, независимо от

числа заданий при каждом выполнении. Среднее замедление задания – это среднее замедление по всем заданиям во всех запусках эмуляции. Эти результаты, объединяющие 3000 выполнений, даются в таблице 2.

Поведение усовершенствованного PVM и усовершенствованного openMosix различно в слегка загруженных и тяжело загруженных сценариях. Это поведение иллюстрировано на рис. 4 – рис. 7, детализирующих первые 1000 выполнений эмуляции.

Замедление	**PVM**	**Усовершен-ствованный PVM**	**oMosix**	**Усовершен-ствованный oMosix**
среднее для выполнения	14.3338	9.79463	8.55676	7.47886
среднее для задания	15.4044	10.7007	9.4208	8.20262

Таблица 2. Среднее замедление в эмуляции Java для различных методов

Каждая точка на рисунке представляет собой отдельное выполнение эмуляции для двух названных методов. На рис. 4 ось X – среднее замедление для PVM, а ось Y – среднее замедление для усовершенствованного PVM. Точно так же на рис. 5 ось X – среднее замедление для openMosix, а ось Y – среднее замедление для усовершенствованного openMosix.

Светлая линия определена как “x = y”. Выше этой линии неусовершенствованный алгоритм работает лучше, чем усовершенствованный алгоритм. Усовершенствованный PVM, как показано в таблице 2, работает

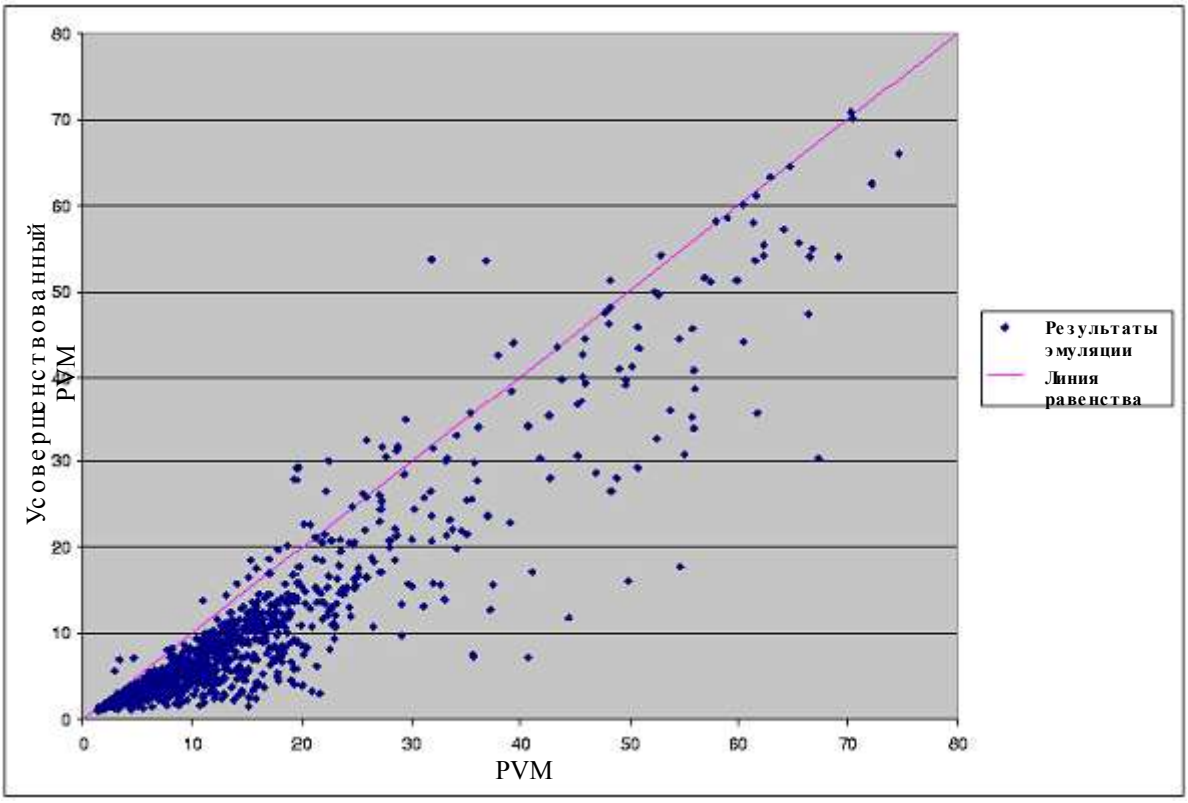


Рис. 4. PVM против усовершенствованного PVM

значительно лучше, чем обычный PVM почти в каждом случае. Более интересным, однако, является поведение усовершенствованного openMosix по сравнению с openMosix. Чем больше было среднее замедление у openMosix на данном выполнении, тем больше улучшений дало ведённое усовершенствование. Интуитивно понятно, что когда выполнение было напряжённым для всех четырёх моделей, усовершенствованный openMosix работал намного лучше, чем неусовершенствованный openMosix. Если данное выполнение было относительно лёгким, и система не была тяжело загружена, усовершенствование имело меньше положительного эффекта.

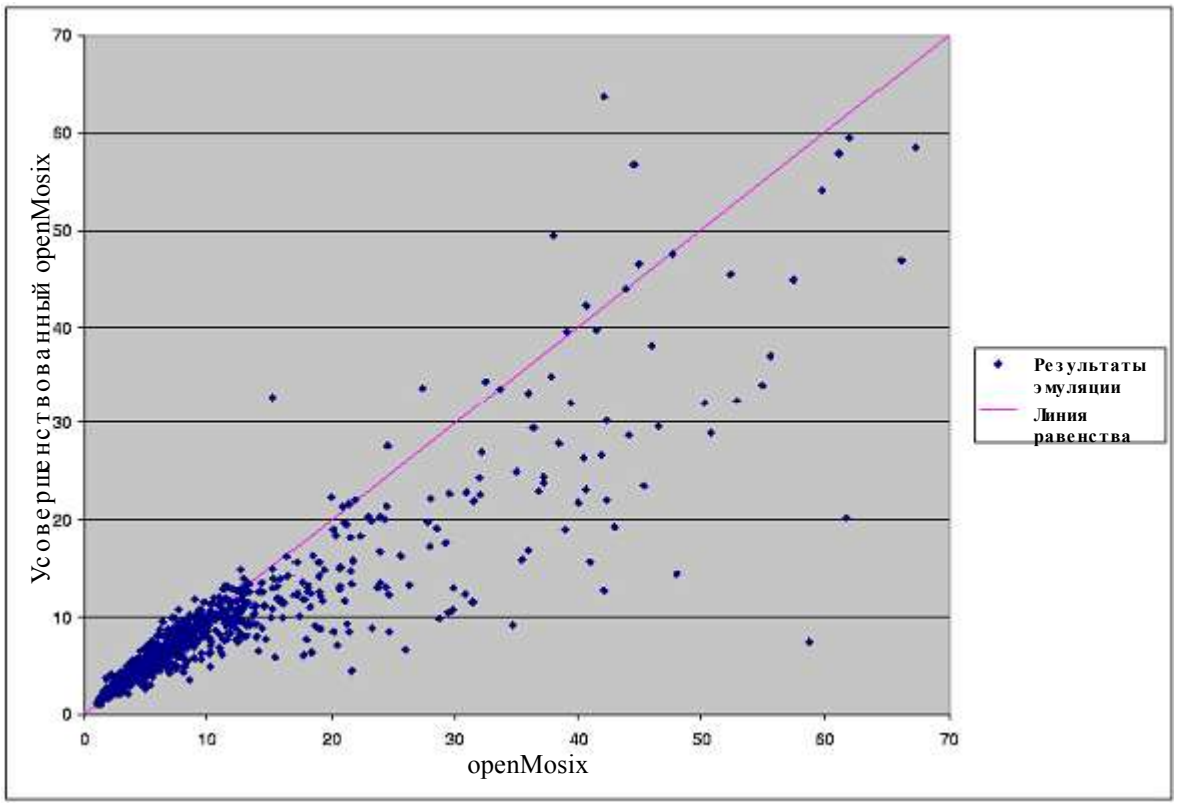


Рис. 5. openMosix против усовершенствованного openMosix

Это можно объяснять следующим образом. Когда машина становится тяжело загруженной или начинает интенсивную подкачку, это затрагивает не только время завершения для заданий, уже назначенных системе. Если машина не станет разгруженной прежде, чем следующий набор больших заданий будет назначен системе, то эта машина становится эффективно недоступной для них, увеличивая нагрузку на всех других машинах. Если много машин начинает интенсивную подкачку или становятся тяжело загруженными, этот эффект будет использовать сам себя. Каждое входящее задание займёт ресурсы системы на намного более длинный промежуток времени, увеличивая замедление заданий, которые прибывают, в то время как вычисляется это задание. Из-за этого эффекта пирамиды, “мудрое” начальное назначение заданий и осторожное восстановление равновесия может привести (в экстремальных случаях) к существенным улучшениям над стандартным openMosix, как показано при некоторых выполнениях на рис. 5.

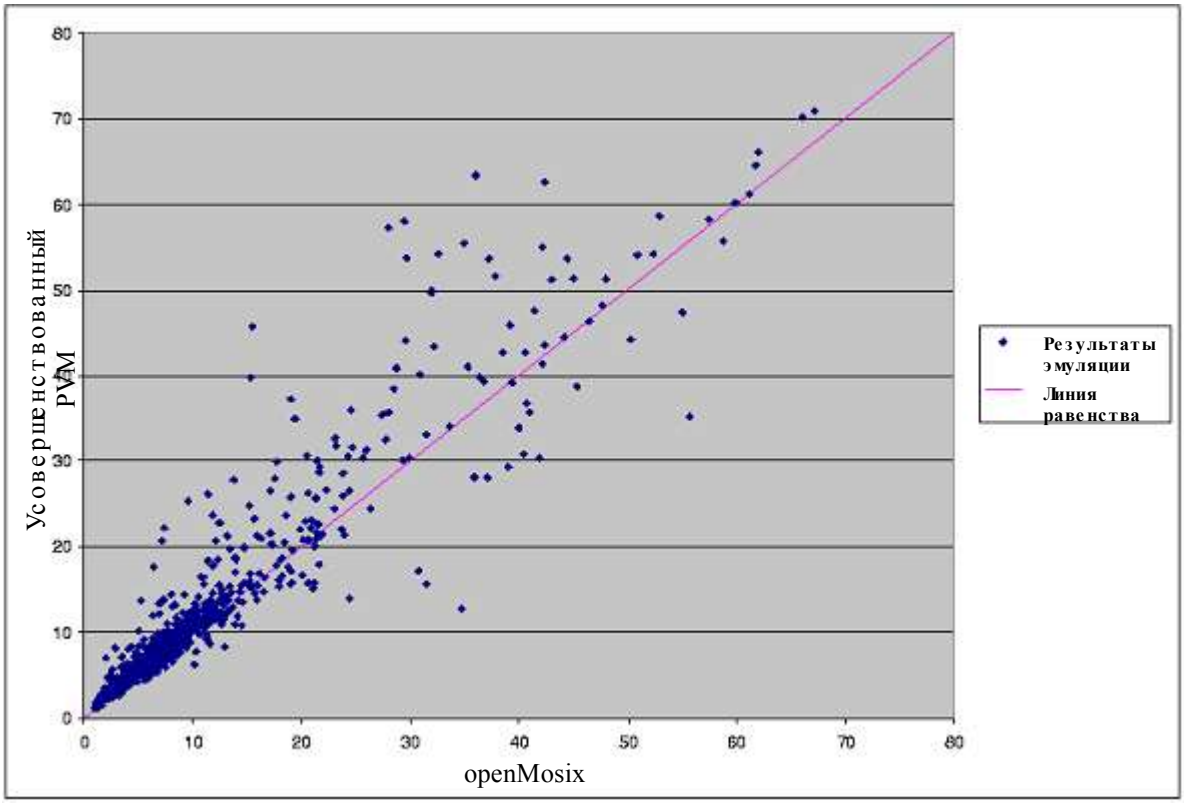


Рис. 6. openMosix против усовершенствованного PVM

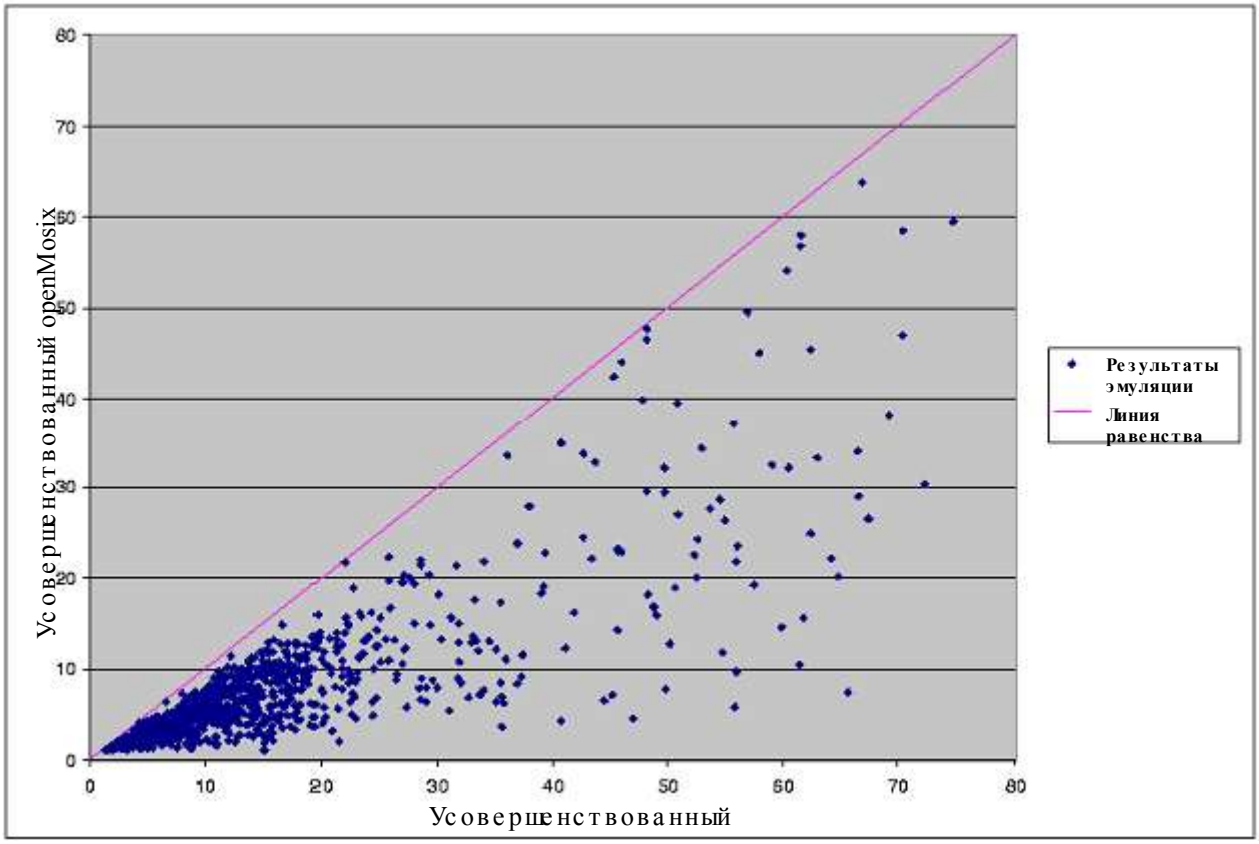


Рис. 7. Усовершенствованный openMosix против усовершенствованного PVM

Особенно интересно обратить внимание на то, что, как видно из таблицы 2 и рис. 6, усовершенствованный PVM метод, который не делает никаких переназначений вообще, умеет достигать приличной (хотя и худшей)

производительности по сравнению с openMosix. Это подчёркивает мощность подхода стоимости возможности: его производительность на нормальной системе не подавлено производительностью более превосходящей системы, которая может исправлять ошибки первоначального назначения.

Важность миграции демонстрируется на рис. 7. Даже при использовании алгоритма стоимости возможности все ещё очень полезно иметь способность миграции в системе. Фактически, усовершенствованный openMosix превосходят по быстродействию усовершенствованный PVM во всех случаях, иногда значительно.

2.3.9. Выполнение на реальной системе

Алгоритмы были также проверены на реальном кластере. Использовалась та же самая модель для входящих заданий, и задания назначались, используя PVM, усовершенствованный PVM и openMosix стратегии. Результаты следующие:

Замедление	PVM	Усовершенство - ванный PVM	openMosix
среднее для выполнения	29.98788	16.29643	13.67707
среднее для задания	33.31620	16.76646	14.00990

Таблица 3. Среднее замедление на реальном кластере для 3-х методов (пере-) назначения

Эти начальные результаты подразумевают, что перегрузки в реальной жизни, меньший размер кластера и другие разнообразные факторы увеличили среднее замедление, что говорит о том, что мы были слишком консервативны в выборе параметров для эмуляции. Однако результаты не изменяют существенно относительные значения. Фактически, усовершенствованные методы показали себя ещё лучше в реальной жизни по сравнению с их оригинальными версиями, чем то, что предсказала Java эмуляция. Предположительно, это есть проверка правильности начальной Java эмуляции и сильное подтверждение качества данного подхода оценки возможности.

Замедление в реальной системе	PVM / oMosix	Enh PVM / oMosix	PVM / Enh PVM
среднее для выполнения	2.19257	1.20416	1.84015
среднее для задания	2.37804	1.20946	1.98707
Замедление при эмуляции	**PVM / oMosix**	**Enh PVM / oMosix**	**PVM / Enh PVM**
среднее для выполнения	1.67514	1.14467	1.46346
среднее для задания	1.63515	1.13585	1.43957

Таблица 4. Средние относительные замедления для 3-х методов (пере-) назначения заданий

2.3.10. Заключение

Рассмотренная экономная стратегия обеспечивает объединённую алгоритмическую структуру для распределения вычислений, взаимодействий, памяти и ресурсов ввода-вывода. Это позволяет разработать почти оптимальные интерактивные алгоритмы для распределения и совместного использования этих ресурсов.

Исследованная стратегия гарантирует почти оптимальную непрерывную эффективность для всей системы в целом в каждом отдельном случае порождения задания и доступности ресурса. Это достигается при помощи использования интерактивных алгоритмов, которые ничего не знают о будущем и предполагают, что между прошлым и будущим нет никакой корреляции, а только осведомлены о текущем состоянии. Несмотря на это, можно строго доказывать, что их эффективность будет всегда сопоставима с оптимальной предвидящей стратегией.

Этот раздел показал, что унифицированный подход стоимости возможности предлагает хорошую практическую эффективность. Сначала были выполнены испытания, использующие моделируемый кластер и “типичный” ряд поступающих заданий. Данный метод, с и без возможности переназначения заданий, был сравнён с методами PVM, доминирующей статической стратегией назначения заданий. Каждому методу предложили идентичные потоки работ. Когда все переназначения были запрещены, метод показал себя печальным усовершенствованием PVM. Когда переназначения позволялись, метод был существенно лучше, чем хорошо настроенная PVM стратегия.

Вторая серия испытаний была выполнена на реальной системе для подтверждения этого моделирования. Реальная система была построена на UNIX машинах с Pentium 133, Pentium Pro 200 и Pentium II, с различным объёмом памяти, связанных Fast Ethernet сетью на основе протокола CSMA-CD. Физический кластер и моделируемый кластер были слегка различны, но пропорциональная эффективность различных стратегий была очень близка к той, которая была получена моделированием. Это показывает, что моделирование соответствующим образом отражает события на реальной системе.

Подход стоимости возможности – универсальный каркас для эффективного распределения гетерогенных ресурсов. Теоретические гарантии слабы: можно только доказать логарифмический предел отставания алгоритма от оптимального offline планировщика. Однако оптимум offline планировщика не является реальной альтернативой; в действительности этот алгоритм конкурирует с наивной online эвристикой.

На практике, этот подход уступает простым алгоритмам, которые значительно превосходят по быстродействию широко используемые и тщательно оптимизированные методы. Заключаем, что теоретические гарантии логарифмической оптимальности – хороший показатель того, что алгоритм будет хорошо работать и практически.

2.4. Алгоритм распределения памяти

Этот алгоритм позволяют узлу, который исчерпал свою основную память, использовать доступную свободную память на других узлах мигрируя процессы, вместо того, чтобы выгружать страницы в область подкачки. Алгоритм распределения памяти наиболее полезен в случаях, когда память используется неравномерно, или когда узлы имеют различные объёмы памяти. Другим случаем, когда этот алгоритм может быть полезен, является случай, когда создан большой процесс, не вмещающийся в свободную память любого узла. Тем не менее, этот процесс может поместиться в память некоторого узла при условии, что текущие выполняющиеся процессы мигрируют с этого узла. Нужно подчеркнуть, что алгоритм не предназначен для работы с процессом, большим, чем объём установленной памяти на любом отдельно взятом узле.

В этом разделе мы рассмотрим эмуляцию нескольких алгоритмов для размещения процессов в SCC, включая Round-robin (циклический), Best-fit (лучшего соответствия) и несколько адаптивных схем. Далее будет показано, что Round-robin в комбинации с миграцией процессов приносит наилучшие результаты среди всех эмулируемых алгоритмов, подтверждая результаты [BAL1]. Так же известно, что эта комбинированная схема хорошо масштабируется вместе с размером кластера. На основе этих результатов реализован алгоритм распределения памяти для openMosix. Показана также взаимосвязь этого алгоритма с алгоритмом балансировки нагрузки, который также является частью общей политики разделения ресурсов. Произведённые измерения алгоритма показывают уменьшение времени выполнения тестов на 175%.

2.4.1. Чем полезен алгоритм распределения памяти

Существует несколько алгоритмов назначения процессов на узлы SCC. Статическое назначение использует предопределённую схему распределения вновь прибывших процессов, не используя никакой информации времени выполнения. Динамическое размещение, напротив, использует системную информацию о состоянии на момент назначения. Для обоих методов, если процесс назначен, он остаётся на узле назначения до завершения выполнения. В противоположенность выше сказанному, алгоритмы адаптивного размещения процессов переназначают процессы между узлами в ответ на изменение состояния системы.

Формально проблема распределения между узлами может быть определена следующим образом: для данного набора процессов с известными размерами памяти, временем прибытия и выполнения, расположить максимальное число процессов в основной памяти всех узлов. Целесообразным является избежание своппинга насколько возможно, поскольку доступ к странице на диске на четыре порядка медленнее, нежели доступ к странице основной памяти. Проблема может иметь достаточно много вариаций в зависимости от

порядка прибытия и расположения процессов, требований на время выполнения, и т.п. К сожалению, в большинстве случаев проблема является NP-трудной.

2.4.2. Эмуляция алгоритмов распределения процессов

В каждом случае будем измерять максимальное заполнение памяти в тот момент, когда алгоритм отказывает, то есть процент используемой памяти в момент, когда алгоритм не может разместить следующий процесс в последовательности в основной памяти любых узлов назначения. Предположим, что любой процесс не меняется во время своего выполнения. Это справедливо, так как наблюдения показывают, что большинство процессов достигают более менее фиксированного размера через короткий период после загрузки и инициализации. Любые значительные изменения размера процесса могут быть рассмотрены как завершение одного процесса и инициация другого. Следовательно, данный алгоритм выполняется только по прибытии нового процесса. Решение о местоположении не зависит от неизвестного оставшегося времени выполнения процесса. Поэтому предположим, что время выполнения процесса бесконечно. Для дальнейшего упрощения ситуации также предположим, что:

- размер процесса известен во время прибытия
- процессы назначаются в порядке своего прибытия
- процесс не может быть разбит между несколькими узлами
- однажды назначенный на узел, процесс может мигрировать, но не может быть выгруженным в область подкачки

Эмулируемое окружение состоит из некоторого числа узлов с 128Мб памяти. Новые процессы прибывают согласно случайно предопределённой последовательности. Размеры процессов выбираются с распределением $p(x)=e^{-x}$, со средним размером от 1Мб до 16Мб.

2.4.3. Статическое и динамическое размещение процессов

Проведём эмуляцию трёх алгоритмов распределения процессов. Основным мотивом является раскрытие поведения каждого алгоритма, когда общекластерная память почти использована, и создаются новые процессы. Первый алгоритм – это циклическое размещение (Round-robin; RR), например, как в PVM. Второй – схема лучшего соответствия (Best-fit; BF), в которой новый процесс назначается на узел с большим объёмом свободной памяти. Третий алгоритм использует политику циклического размещения с последующим соответствием (Round-robin Next-fit; NF), в которой все узлы сканируются циклическим образом, и процесс назначается на первый же узел, который имеет достаточно свободной памяти. Можно также было рассмотреть и политику первого соответствия (First-fit; FF). Тем не менее, так как эта политика имеет тенденцию неравномерно распределять нагрузку, его нельзя считать реалистичной альтернативой.

На рис. 8 представлены результаты эмуляции этих трёх алгоритмов размещения для набора процессов со средним размером 1Мб. Этот набор эмулирует распределение процессов по серверам за месяц на реальной системе. Результаты представлены для возрастающего по логарифмической шкале числа узлов. Каждое измерение представляет собой среднее по 500 различным запускам.

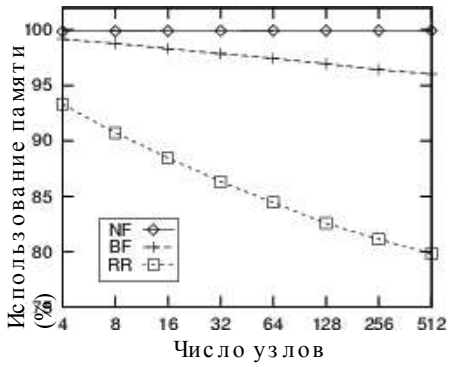


Рис. 8. Производительность алгоритмов размещения процессов

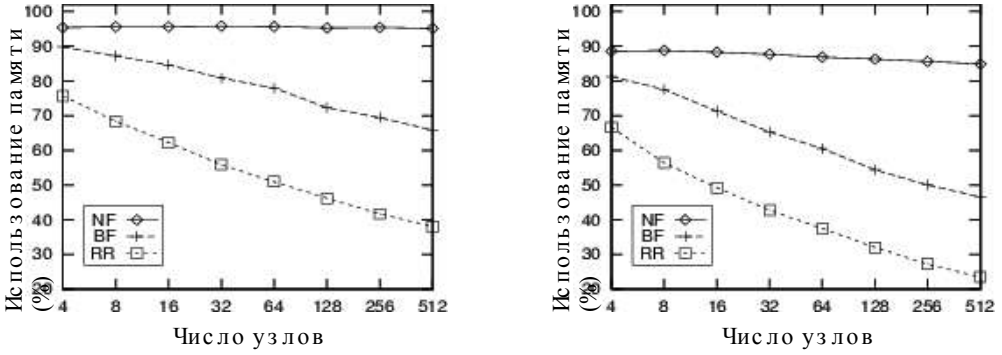
Их рисунка видно, что NF имеет заполнение памяти, близкое к оптимальному, для любого числа узлов. Это может быть объяснено его гибкостью пропускать узлы, которые истощили всю свою свободную память. Соответствующие результаты для алгоритма RR показывают уменьшение максимального заполнения памяти с 93% для четырёх узлов до 80% для 512 узлов. Причиной таких низких значений является то, что RR алгоритм отказывает, как только он получает процесс, не вмещающийся в память следующего узла. Можно наблюдать также, что производительность RR падает линейно по логарифмической шкале.

Производительность алгоритма BF несколько ниже, чем NF, с логарифмическим отставанием до 4% от NF. Поведение BF может быть объяснено тем фактом, что этот алгоритм имеет тенденцию распределять память равномерно между всеми узлами. Следовательно, он откажет более вероятней, чем NF, когда прибывает большой процесс, всё ещё получая большую выгоду от свободы действий, чем RR.

Для того, чтобы подчеркнуть уменьшение производительности RR и BF алгоритмов, вспомним, что размеры процессов выбираются случайным образом

из фиксированного распределения с заданным средним размером. Следовательно, с ростом числа узлов пропорционально увеличивается объём кластерной RAM, таким образом, требуя создания большего числа процессов для того, чтобы заполнить эту память. Тем не менее, создание процессов подразумевает, что число больших процессов также увеличится. Эти большие процессы более трудно разместить, что, соответственно, является главной причиной отказа этих двух алгоритмов.

Для того, чтобы выделить эти последние наблюдения, съэмулируем те же самые три алгоритма размещения, используя средние размеры процессов 8 и 16Мб. Отметим, что такие размеры процессов могут отражать научные вычислительные среды. Результаты эмуляции показаны на рис. 9. На рисунке показано, что производительность алгоритма RR уменьшилась с 75% до 40% для среднего размера процесса 8Мб и до 20% для среднего размера процесса 16Мб. Соответствующая производительность алгоритма NF осталась в районе 95% для 8Мб, и постепенно уменьшилась с 90% до 85% для 16Мб.



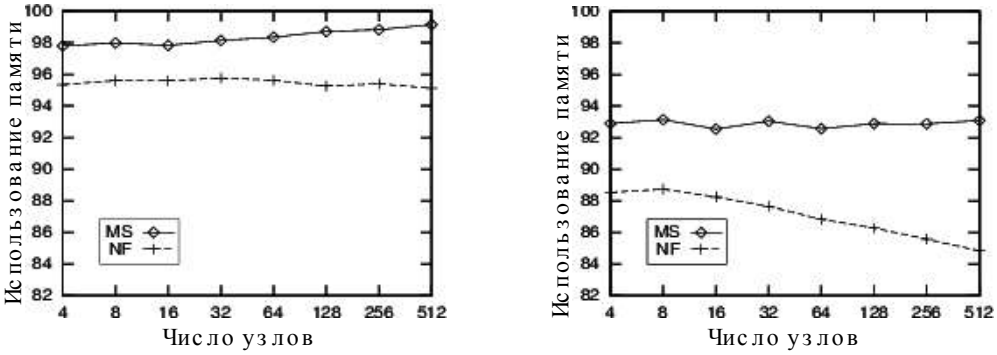
а) Средний размер процесса 8Мб б) Средний размер процесса 16Мб

Рис. 9. Производительность алгоритмов размещения процессов с большим средним размером процесса

Сравнение результатов на рис. 8 и рис. 9 доказывает наше утверждение что, чем больше создаётся больших процессов, чем более увеличивается спад производительности RR и BF алгоритмов по мере увеличения числа узлов.

2.4.4. Адаптивное размещение процессов

Далее будем тестировать вариацию RR алгоритма, использующего адаптивное размещение процессов, называемого “мигрирующий наименьший процесс” (Migrate the Smallest process; MS). Алгоритм MS циклически сканирует узлы и всегда помещает процесс на следующий узел. В отличие от RR, если на узле заканчивается память, то MS мигрирует некоторый процесс с этого узла. Выбранный процесс является наименьшим среди всех процессов, которые достаточно велики, чтобы аннулировать переполнение памяти, вызванное прибывшим процессом, и может быть размещён на некотором другом узле. Согласно этому, миграция подвергает сеть наименьшим затратам. Если такого процесса не существует (потому что все процессы слишком велики или все другие узлы переполнены), то алгоритм отказывает.



а) Средний размер процесса 8Мб б) Средний размер процесса 16Мб

Рис. 10. Производительность алгоритмов MS и NF

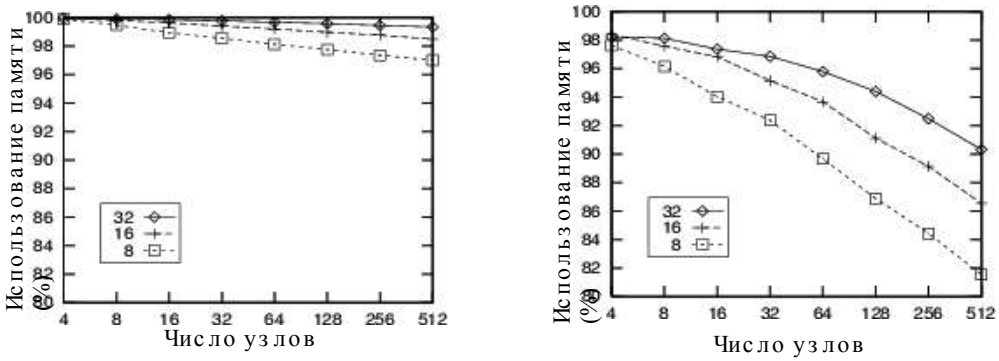
На рис. 10 показаны эмуляции алгоритмов NF и MS со средним размером процесса от 8Мб до 16Мб. Отметим, что для среднего размера процесса в 1Мб производительность обоих алгоритмов близка к 100% заполнению памяти. Из рис. 10 а) видно, что производительность MS от 3% до 4% выше, чем NF. Можно наблюдать также, что производительность алгоритма MS увеличивается с числом узлов и не изменяется для алгоритма NF. Это может быть объяснено большим выбором процессов для миграции для MS при увеличении числа узлов. Улучшенная масштабируемость MS подчёркнута при эмуляции со средним размером процессов 16Мб на рис. 10 б).

Отметим, что также были проведены исследования двух вариантов алгоритмов NF и MS, в которых выбранный процесс помещался на узел с наибольшим объёмом свободной памяти. Тем не менее, оба варианта показали намного худшую производительность и большее уменьшение производительности при увеличении числа узлов.

2.4.5. Масштабируемый алгоритм размещения с децентрализированным управлением

Все выше описанные алгоритмы использовали централизированные схемы, которые полагались на глобальную информацию для принятия решения о размещении. Эти централизированные механизмы не масштабируемы и не могут быть эффективно реализованы на кластерах с большим числом узлов.

Упрощённая децентрализированная схема для размещения процессов может основываться на алгоритме MS, в которой мигрируемый процесс размещается на узле, выбранным из информационного окна (см. гл. 2.1). Существенными параметрами этого алгоритма являются размер окна и частота рассеивания информации. Эмуляция вышеописанной схемы, называемой “оконный MS”, показана на рис. 11.



а) Средний размер процесса 1Мб б) Средний размер процесса 8Мб

Рис. 11. Производительность оконного MS алгоритма для различных размеров окна

Показаны результаты для размеров окон с 8, 16 и 32 элементами. Как и ожидалось, производительность падает при увеличении числа узлов и при уменьшении размера окна. Это происходит из-за ограниченности информации, доступной алгоритму, что препятствует глобальной оптимизации. Несмотря на уменьшение производительности, алгоритм работает хорошо: более чем 96% заполнение памяти для процессов со средним размером 1Мб, и более 90% для 8Мб процессов вплоть до 32 узлов. Алгоритм легко реализуем и не требует никакой глобальной информации.

2.4.6. Реализация алгоритма распределения памяти

В этом разделе будут рассмотрены детали реализации алгоритма распределения памяти в системе openMosix. Этот алгоритм ответственен за принятие решений касательно миграции процессов с узла, исчерпавшего свою свободную память, так же как и предотвращение миграции на этот узел. Этот алгоритм должен решать, какой процесс мигрировать, когда его мигрировать, и где его разместить. В дополнение к этому, алгоритм должен решать, позволить ли удалённым процессам мигрировать на этот узел из-за истощения памяти на других узлах.

Для достижения решения алгоритм использует индекс свободной памяти, обеспечиваемый алгоритмом рассеивания информации. Этот индекс есть объём измеренной свободной памяти узла, который может быть распределён между удалёнными процессами. Строго говоря, это есть сумма действительной свободной памяти за вычетом порогового значения свободной памяти, то есть памяти, зарезервированной для локальных процессов и памяти, уже переданной прибываемым процессам. Это значение замеряется несколько раз в секунду для ослабления кратковременных флуктуаций. Индекс свободной памяти рассылается через периодические интервалы другим узлам, используя вероятностный алгоритм рассеивания информации. Последний алгоритм предназначен для того, чтобы обеспечить каждый узел достаточной информацией о доступных ресурсах на других узлах, без проведения опроса или доверия удалённой информации (см. [BAR3]).

Ниже рассмотрены основные части алгоритма распределения памяти:

активизация, выбор узла назначения и процесс миграции. Последние два решения тесно связаны с соответствующими частями внутрисеточного алгоритма. Если алгоритм отрабатывает успешно, то как минимум один процесс мигрирует, в противном случае, узел начинает выгружать страницы.

2.4.7. Активизация алгоритма

Алгоритм распределения памяти срабатывает, когда свободная память узла становится ниже порогового значения, что может вызвать выгрузку ценных страниц памяти. Пороговое значение – это объём памяти, зарезервированной для локальных процессов в то время, как принимается решение о выборе и миграции процесса. В идеале, объём зарезервированной памяти должен быть динамическим значением, которое соответствует частоте распределения свободной памяти. Это значение должно увеличиваться, когда увеличивается интенсивность распределения страниц и наоборот, чтобы быть уверенным, что алгоритм распределения памяти выберет и, если необходимо, мигрирует процесс перед тем, как сработает процесс выгрузки страниц. В то же время, размер зарезервированной памяти не должен быть слишком большим. Это нужно для избежания ранних и ненужных миграций процессов с узла, а также блокировки миграций на узел. В настоящей реализации пороговое значение установлено в фиксированную величину: на 1/4Мб больше уровня срабатывания механизма выгрузки страниц.

2.4.8. Выбор процесса

Выбор процесса для миграции проходит в несколько этапов в зависимости от объёма недостающей памяти (переполнения) и объёма свободной памяти на других узлах. Прежде всего, выясняется размер переполнения как разность между настоящими требованиями к памяти и доступной памятью. Значение переполнения памяти наблюдается и обновляется регулярно для определения того, является ли это кратковременным явлением или постоянным.

На первой стадии процессы сортируются по числу своих “грязных” страниц. Выбирается процесс с наименьшими затратами на миграцию из числа процессов, для которых число “грязных” страниц превышает значение переполнения. Если эта стадия завершается неудачей, то ли из-за отсутствия узла с достаточным объёмом свободной памяти для предоставления выбранному процессу, то ли потому что все процессы высылающего узла меньше значения переполнения памяти, то алгоритм переходит ко второй стадии.

На второй стадии алгоритм пытается мигрировать наибольший возможный процесс, который размещается на удалённом узле, не заставляя тот узел превысить своё пороговое значение памяти. Сначала алгоритм выбирает узел с наибольшим известным индексом свободной памяти. Потом он находит наибольший процесс, который помещается на этот узел. Потом алгоритм проверяет, поместится ли выбранный процесс в память других узлов. Среди

найденных узлов, он выбирает узел с наименьший нагрузкой и мигрирует процесс на этот узел. После этого, если доступная свободная память всё ещё ниже порогового значения, то алгоритм ждёт некоторое время для прибытия новых индексов свободной памяти и возвращается к первой стадии. В противном случае он завершается.

2.4.9. Узел назначения

Узел назначения выбирается из подмножества узлов, чьи индексы свободной памяти доступны в информационном окне. Выбранный узел должен иметь индекс свободной памяти больший или равный размеру процесса, назначенного для миграции. Это необходимо для того, чтобы предотвратить выгрузку страниц на узле-получателе, и так же предотвратить ситуацию, когда процесс неоднократно мигрирует от одного узла к другому. Из этих узлов выбирается узел с наименьшей нагрузкой. Если никакого узла не найдено, то алгоритм повторяет выполнение через короткую задержку, скажем, 1 секунду, по истечении которой с достаточно высокой вероятностью ожидается, что будет доступна новая информация о свободных индексах памяти других узлов.

В вышеприведённой схеме выбирается узел с наименьший нагрузкой для избежания, насколько возможно, дальнейших миграций процессов в связи с разбалансировкой нагрузки после миграции. Основная идея состоит в том, чтобы ограничить выполнение алгоритма распределения памяти на периоды, когда свободная память на узле становится дефицитом. В противном случае, используется только алгоритм балансировки нагрузки.

Нужно отметить, что перед началом любой миграции процесса необходимо получение подтверждения с узла назначения. Это нужно для избежания нескольких одновременных миграций с различных узлов.

3. МАСШТАБИРУЕМЫЕ КЛАСТЕРНЫЕ ФАЙЛОВЫЕ СИСТЕМЫ OPENMOSIX ДЛЯ LINUX

3.1. Задачи кластерных файловых систем

Недавние успехи в кластерных вычислениях и способности генерировать и мигрировать параллельные процессы на несколько машин создали потребность разработки масштабируемых кластерных файловых систем, которые не только поддерживают параллельный доступ ко многим файлам, но также и согласованность кэшей между процессами, которые обращаются к тому же самому файлу. Наиболее традиционные сетевые файловые системы типа NFS, AFS и Coda не отвечают требованиям параллельной обработки, потому что они зависят от центральных файловых серверов. Новое поколение файловых систем, типа GFS, xFS и Frangipani, соответствуют кластерам в большей степени, потому что эти системы распределяют память, кэш и управление между станциями кластера, а также обеспечивают средства для параллельного доступа к файлу и согласованности кэшей. Масштабируемость этих файловых систем всё ещё не известна. В этом разделе рассматривается новый пример для масштабируемых кластерных вычислений, который объединяет кластерные файловые системы с динамическим распределением заданий.

Целевой кластер состоит из множества гомогенных узлов (с одним CPU и/или SMP компьютеров), которые работают совместно, позволяя общим ресурсам кластера быть доступными каждому процессу, и делая возможным любому узлу обратиться к любому файлу или выполнить любой процесс. Кластерная файловая система состоит из нескольких поддеревьев, которые размещены на различных узлах, чтобы разрешить параллельные операции с различными файлами. Главная особенность в дизайне кластерных файловых систем – способность мигрировать процесс к файлу вместо того, чтобы традиционным способом перенести данные файла к процессу. Эта возможность, которая уже поддерживается openMosix для Linux для балансировки нагрузки, создаёт новый подход в кластерных вычислениях, который обеспечивает “удобную для использования” среду для улучшения производительности и масштабируемости. Обратим внимание на то, что большинство существующих пакетов для распределения работ, такие как PVM или MPI, не соответствуют нашим требованиям, потому что они используют статическое (фиксированное) распределение процессов по узлам.

openMosix особенно эффективен для распределения и выполнения процессов, ограниченных только использованием CPU (CPU-limited jobs). Однако из-за совместимости с Linux, схема openMosix распределения процессов была неэффективна при выполнении процессов с существенным количеством операций ввода-вывода и/или файловых операций. Для преодоления этой неэффективности openMosix был расширен поддержкой DFSA для лучшей обработки процессов, ограниченных только операциями ввода-вывода (I/O

limited jobs). С DFSA большинство системных вызовов мигрировавшего процесса, ориентированного на ввод-вывод, может быть выполнено локально – на узле, где этот процесс в настоящее время расположен.

Основное преимущество DFSA состоит в том, что она позволяет процессу с умеренным или большим объёмом ввода-вывода мигрировать на узел, на котором он выполняет большинство операций ввода-вывода, и воспользоваться преимуществом локального доступа к данным. Другое преимущество – уменьшение накладных расходов на взаимодействие, таких как меньшее количество операций ввода-вывода, производящихся по сети.

Для правильной работы DFSA требуется согласованность каждого файла между процессами, работающих на различных узлах, которую в настоящее время обеспечивают только несколько файловых систем. Одна такая файловая система – GFS, которую планируется использовать, как только она станет полностью работоспособной. В качестве другой альтернативы разработан прототип файловой системы, называемой oMFS, которая рассматривает все файлы и каталоги в пределах кластера openMosix как единую файловую систему.

Интересной исследовательской проблемой является проблема, где разместить процесс, когда он выполняет операции ввода-вывода с файлами, расположенных на различных узлах. Простое решение состоит в том, чтобы мигрировать процесс к узлу, для которого выполнено большинство операций ввода-вывода. Эта проблема несколько усложняется, если также хочется сбалансировать нагрузку, или если модели операций ввода-вывода изменяются во времени, или когда вовлечено несколько процессов. Адаптивная политика распределения процессов openMosix пытается разрешить все это случаи.

3.2. Файловая система прямого доступа (DFSA)

3.2.1. Принцип работы DFSA

DFSA – программное обеспечение, которое решает, выполнить ли системные вызовы файловых операций ввода-вывода мигрировавшего процесса на текущем узле или на его домашнем узле. Для корректного функционирования DFSA требует, чтобы выбранные файловые системы (и символические ссылки) были идентично смонтированы на одноимённые точки монтирования. Кроме того подразумевается, что схема идентификаторов пользователей/групп является или идентичной по всему кластеру, или достаточно безопасной, чтобы не возникало никаких нарушений прав доступа, когда к одной и той же файловой системе обращаются пользователи с идентификаторами, назначенными на различных узлах.

DFSA проверяет, что данные файловые системы смонтированы с теми же самыми флагами монтирования, и их типы поддерживаются DFSA. Тогда она перенаправляет большинство обращений к операционной системе для

выполнения локально (на текущем узле), в то же время всё ещё направляя некоторые (сомнительные) системные вызовы, например, при совместном использовании открытых файлов с другими процессами, к домашнему узлу процесса. DFSA непрерывно синхронизирует состояние файла, например, открытие, закрытие, позицию в файле и т.д. между мигрировавшим процессом и его домашним узлом.

3.2.2. Требования DFSA

DFSA может работать с любой файловой системой, которая удовлетворяет следующим свойствам:

- Согласованность: когда процесс делает изменения в файле с любого узла, все другие узлы должны видеть это изменение немедленно или не позже следующего доступа к этому файлу. На модели клиент/сервер это обычно подразумевает, что становится невозможным использование кэша на клиентских узлах либо единого виртуального кэша, который может находиться на любом узле (например, на сервере). Обратим внимание, что усложнённый сервер, вероятно, может предоставлять кэш отдельных файлов и/или блоков некоторому узлу в любое время. Файловые системы с совместно используемыми аппаратными средствами могут предлагать другие решения для согласованности.
- Временные отметки файлов и между файлами на одной и той же файловой системе должны быть согласованы и увеличиваться (если часы не были преднамеренно установлены назад) независимо от узла, с которого производятся модификации.
- Файловая система должна гарантировать, что файлы/каталоги не освобождаются при удалении до тех пор, пока существует процесс в кластере, который держит их открытыми. Есть несколько возможных методов для достижения этого, например, некоторая модель сборки “мусора”.
- Поддержка нескольких дополнительных функций файловой системы, которые в Linux считаются операциями с супер-блоком, например, “идентифицировать” – сформировать конечную идентификационную информацию по “d-вхождению (dentry)” таким образом, чтобы её было достаточно для восстановления этого открытого файла/каталога на другом узле.

Поддерживаются следующие системные вызовы, которые могут быть выполнены локально, без миграции процесса на домашний узел (UHN):

read, readv, write, writev, readahead, lseek, llseek, open, creat, close, dup, dup2, fcntl, fcntl64, getdents, getdents64, old_readdir, fsync, fdatasync, chdir, fchdir, getcwd, stat, stat64, newstat, lstat, lstat64, newlstat, fstat, fstat64, newfstat, access, truncate, truncate64, ftruncate, ftruncate64, chmod, chown, chown16, lchown, lchown16, fchmod, fchown, fchown16, utime, utimes, symlink, readlink, mkdir, rmdir, link, unlink, rename

В следующих ситуациях системные вызовы на смонтированной с поддержкой

DFSA файловой системе могут не работать:

- различная конфигурация MFS/DFSA на узлах кластера (например, на некоторых не включена поддержка DFSA или MFS смонтирована с различными опциями)
- dup2() вызывается со вторым файловым дескриптором вне поддерева DFSA
- вызов chdir()/fchdir() в случае, если родительский каталог не DFSA
- пути выходят за пределы поддерева DFSA
- процесс, делающий системный вызов из приведённого выше списка, находится в состоянии отладки/трассировки

3.3. Файловая система openMosix (oMFS)

openMosix требует согласованности файлов и кэшей между процессами, которые выполняются на различных узлах, поскольку может оказаться, что даже один и тот же процесс (или группа взаимодействующих связанных процессов) оперирует с различных узлов.

Чтобы воспользоваться DFSA, был реализован прототип файловой системы, называемой файловой системой openMosix (oMFS). oMFS обеспечивает унифицированное представление всех файлов на всех установленных файловых системах (любого типа) на всех узлах кластера openMosix, как будто они все находятся в пределах единственной файловой системы. Например, если MFS смонтирована на точку монтирования `/mfs`, то файл `/mfs/1456/usr/tmp/myfile` относится к файлу на узле #1456. Это делает oMFS и универсальной, так как `/usr/tmp` может иметь любой тип файловой системы, и масштабируемой, так как каждый узел в кластере openMosix потенциально и сервер, и клиент.

В отличие от других файловых систем, oMFS обеспечивает согласованность кэшей, поддерживая только один кэш – на сервере. Чтобы реализовать это, стандартные кэши дисков и директорий Linux используются только на сервере и игнорируются на клиентах. Основное преимущество такого подхода состоит в том, что это обеспечивает простую, всё ещё масштабируемую схему согласованности кэшей между любым числом процессов. Другое преимущество подхода oMFS состоит в том, что она позволяет поднять клиент-серверное взаимодействие на уровень обращений к операционной системе, который является лучшим для более общих, больших и сложных операций ввода-вывода (открытие файлов, ввод-вывод блоков большого размера). Очевидно, отсутствие кэша на клиенте – главный недостаток для операций ввода-вывода с блоками меньшего размера.

3.4. Производительность

Этот раздел демонстрирует производительность, полученную при использования DFSA. Так как DFSA – не файловая система, то её можно оценить, сравнивая производительность MFS с и без DFSA. Для сравнения также оценены производительность локальных операций ввода-вывода и NFS.

Использовался тест PostMark, который моделирует тяжёлые нагрузки файловой системы, например, как на большом сервере электронной почты Internet. PostMark создаёт большой пул текстовых файлов произвольного размера и затем исполняет множество транзакций, где каждая транзакций состоит из пар меньших транзакций. Это выполняется до тех пор, пока не будет получена предопределённая рабочая нагрузка. Все запуски были выполнены на кластере с openMosix 2.4.18 для Linux. Тест был выполнен между парой идентичных (Pentium 550 MHz PC с дисками IDE) машин, используя 4 различных метода доступа к файлу и размеры блоков пределах от 64б до 16Кб.

Результаты теста показаны в таблице 5, где каждое измерение представляет собой среднее время 10 запусков в секундах. Первая строка в таблице показывает время выполнения на локальной машине, при котором процесс и файлы были на серверном узле. Вторая строка показывает время MFS с DFSA. В этом случае тест, запущенный на клиентском узле, мигрировал на узел сервера. Третья строка показывает время NFS между клиентским узлом и серверным узлом. Последняя строка показывает время MFS без DFSA, в котором все файловые операции были выполнены с клиентского узла к узлу сервера через MFS с заблокированной DFSA.

Метод доступа	**Размер блока данных при передаче**						
	64б	**512б**	**1Кб**	**2Кб**	**4Кб**	**8Кб**	**16Кб**
Локальный (на сервере)	102.6	102.1	100.0	102.2	100.2	100.2	101.0
MFS с DFSA	104.8	104.0	103.9	104.1	104.9	105.5	104.4
NFS	184.3	169.1	158.0	161.3	156.0	159.5	157.5
MFS без DFSA	1711.0	382.1	277.2	202.9	153.3	136.1	124.5

Таблица 5. Результаты тестирования PostMark

Из таблицы 5 следует, что время выполнения MFS с DFSA на 1.8-4.9% медленнее, чем время локального выполнения для каждого из измеренных размеров блоков, и со средним замедлением 3.7% для всех измеренных размеров блоков. Также время выполнения MFS с DFSA на 51-76% быстрее, чем NFS, со средним ускорением 59% для всех измеренных размеров блоков. Заметим, что NFS на 56-80% медленнее, чем локальная файловая система. Как ожидалось, MFS без DFSA (четвёртая строка) медленнее, чем NFS для маленьких размеров блоков. Однако, для размеров блоков 4КБ и больше, MFS имеет лучшую производительность, чем NFS. Это делает MFS разумной альтернативой для кластерной файловой системы, которая поддерживает

согласованность кэшей.

В таблице 6 показано, что “DFSA+MFS” имеет производительность, сравнимую с “Local FS” (при тестировании использовался подходящий файл размером 1Гб, буферизация чтения/записи включена).

Тип	Последовательный вывод (блочная запись) (Кб/с)	Последовательный ввод (блочное чтение) (Кб/с)
Local FS	22155	27476
NFS	10176	9372
MFS	8203	8235
DFSA+MFS	21997	28176

Таблица 6. Производительность согласно тестам Bonnie

Второй тест показывает выполнение файловых операций ввода-вывода между множеством клиентских процессов, каждый из которых выполняет на своём собственном узле параллельное файловое приложение, которое взаимодействует с общим файловым сервером. Более определённо, эта параллельная программа читает общую базу данных размером 200Мб из оперативной памяти четырёхпроцессорного (Xeon 550 МГц) файлового сервера.

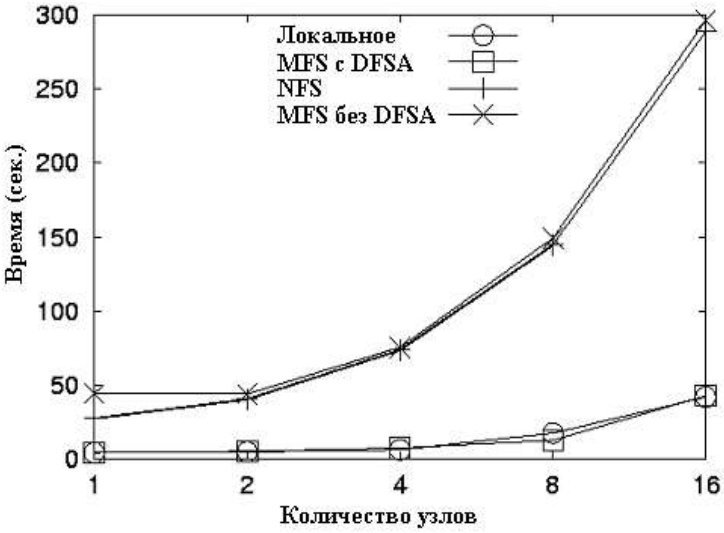


Рис. 12. Среднее время считывания общего файла с четырёхпроцессорного файлового сервера

Приложение было запущено на различных файловых системах, как показано на рис. 12, используя размер блока 4Кб. Для сравнения тест был также выполнен локально на серверном узле. Приложение было запущено одновременно на всех узлах (Pentium 550 МГц), и было зафиксировано время, необходимое для завершения последнего процесса.

Результаты этого теста усреднены по трём различным запускам и показаны на рис. 12. Из рисунка можно заметить, что время выполнения MFS с DFSA очень близко к локальному времени. Фактически, MFS с DFSA был в среднем на 6% медленнее, чем локальная файловая система. Заметим, что эти накладные затраты возникают из-за дополнительного программного уровня в MFS. Из рисунка может также быть замечено, что NFS и MFS без DFSA почти идентичны, причём NFS обеспечивает в среднем на 4% лучшую производительность, чем MFS без DFSA. Из полученных результатов следует, что MFS с DFSA от 6.1 до 11.4 раз быстрее, чем NFS, и от 6.9 до 11.8 раз быстрее, чем MFS без DFSA. Из этих результатов ясно, что MFS с DFSA обеспечивает лучшую

масштабируемость, чем NFS или MFS без DFSA.

3.5. Заключение

Правильное оперирование этой схемы требует согласованности файлов между процессами, которые работают на различных узлах. Ещё одна подходящая файловая система для нашей схемы – глобальная файловая система (GFS), которая позволяет множеству узлов Linux совместно использовать сетевые устройства для хранения. В GFS каждый узел видит сетевые диски как локальные, и сама GFS проявляет себя как локальная файловая система. Другая привлекательная особенность GFS – то, что она использует современные интерфейсы, такие как оптоволоконный канал, для использования присоединённых сетевых устройств хранения.

Один из последующих проектов состоит в том, чтобы приспособить GFS к openMosix и DFSA. Подобные расширения могли быть разработаны для других файловых систем, которые удовлетворяют требования DFSA. Кроме того, было бы интересно исследовать средства комбинирования MFS с локальными сетевыми устройствами хранения, чтобы обеспечить согласованность и большее количество вариантов взаимодействия между масштабируемым кластером и устройствами хранения.

4. МИГРИРУЕМАЯ РАЗДЕЛЯЕМАЯ ПАМЯТЬ

4.1. Цели проекта

Целью проекта является сделать возможным миграцию процессов с явным разделением памяти и миграцию мультипотоковых приложений без потери производительности и с гарантированном сохранением работоспособности на кластере openMosix.

openMosix – это расширение ядра Linux, которое предоставляет вычислительное кластерное окружение с автоматической балансировкой нагрузки между узлами кластера. Миграция процессов с разделяемой памятью не обрабатывается в текущей версии openMosix. Следовательно, такие приложения как Web сервера, сервера баз данных, некоторые инструменты рендеринга, которые используют разделяемую память, не могут получить выгоду от кластера openMosix. Данный проект преодолевает это ограничение openMosix. При этом возможна миграция процессов, которые явно используют разделяемую память посредством таких системных вызовов, как `shmget()`, `shmat()` и `shmctl()`, так же, как и потоков, созданных с использованием системного вызова `clone()`. Поддержание согласованности разделяемой памяти, к которой обращаются процессы, распределённые по кластеру, обеспечивается прозрачной политикой согласованности “желающего освобождения” (“Eager Release” transparent consistency policy). В дополнение сама разделяемая память мигрирует основываясь на частоте использования разделяемой памяти и нагрузки по кластеру. Миграция разделяемой памяти уменьшает число удалённых обращений к ней.

Данная реализация позволяет миграцию Apache Web сервера на кластере openMosix, что было ранее невозможно.

В Linux разделяемая память реализована двумя путями:

- Sys V разделяемая память межпроцессного взаимодействия
- потоки (легковесные процессы Linux)

4.2. Sys V разделяемая память

Sys V разделяемая память – самое быстрое и самое удобное средство IPC. Она позволяет двум и более процессам получать доступ к общим структурам данных, помещая их в сегмент разделяемой памяти. Разделяемая память реализована как файл в виртуальной файловой системе памяти (tmpfs или бывшая shmfs).

Разделяемая память управляется при помощи следующих системных вызовов:

- `shmget()`
  Функция `shmget()` вызывается для получения идентификатора IPC сегмента разделяемой памяти, опционально создавая его, если он не существует.

- `shmat()`
  Функция `shmat()` вызывается для присоединения сегмента разделяемой памяти к процессу. В качестве параметра она получает идентификатор IPC и пытается добавить регион разделяемой памяти к адресному пространству вызывающего процесса. Вызывающий процесс может затребовать специфический стартовый линейный адрес для региона памяти, но обычно адрес не важен, и каждый процесс, который получает доступ к сегменту разделяемой памяти, может использовать отличный от других адрес в своём адресном пространстве. Таблицы страниц процесса остаются неизменными после `shmat()`.
- `shmdt()`
  Функция `shmdt()` вызывается для отсоединения сегмента разделяемой памяти, заданного своим идентификатором IPC, то есть для удаления соответствующего региона памяти из адресного пространства процесса. IPC регион разделяемой памяти является постоянным: даже если никакой процесс его не использует, соответствующие виртуальные страницы не могут быть освобождены, хотя они могут быть выгружены в область подкачки.
- `shmctl()`
  Функция `shmctl()` позволяет пользователю получить информацию о сегменте разделяемой памяти, установить владельца, групповые права доступа, или разрушить сегмент. Пользователь должен убедиться, что сегмент в конце концов разрушен, иначе его страницы будут оставаться в памяти или в области подкачки.

Страницы, добавляемые к процессу при `shmat()`, являются фиктивными; функция добавляет регион памяти к адресному пространству процесса, но не изменяет таблицы станиц процесса. Поэтому когда процесс пытается получить доступ к адресу сегмента разделяемой памяти, возникает “ошибка при обращении к странице”. Соответствующий обработчик прерывания определяет, что сбойный адрес находится внутри адресного пространства процесса, и соответствующая запись в таблице станиц процесса NULL; он вызывает функцию `do_no_page()`. В свою очередь эта функция проверяет, определён ли метод `no_page()` для этого региона памяти. Вызывается этот метод, и запись в таблице станиц устанавливается в адрес, возвращённый этим методом.

4.3. Легковесные и тяжеловесные процессы в Linux

Потоки – это легковесные процессы (light weight processes (LWPs)). Идея в том, что процесс состоит из пяти фундаментальных частей: код, данные, стек, файловый ввод-вывод и таблица сигналов. Тяжеловесные процессы (heavy-weight processes (HWP)) имеют значительные накладные расходы при переключении между процессами: все таблицы должны быть сброшены из процессора при каждом переключении. Также единственным способом достичь разделения информации между HWP является использование каналов и разделяемой

памяти. Если HWP порождает дочерний HWP используя `fork()`, единственной разделяемой частью являются данные. Потоки уменьшают накладные расходы используя разделение фундаментальных частей. При совместном использовании этих частей переключение происходит гораздо более эффективно.

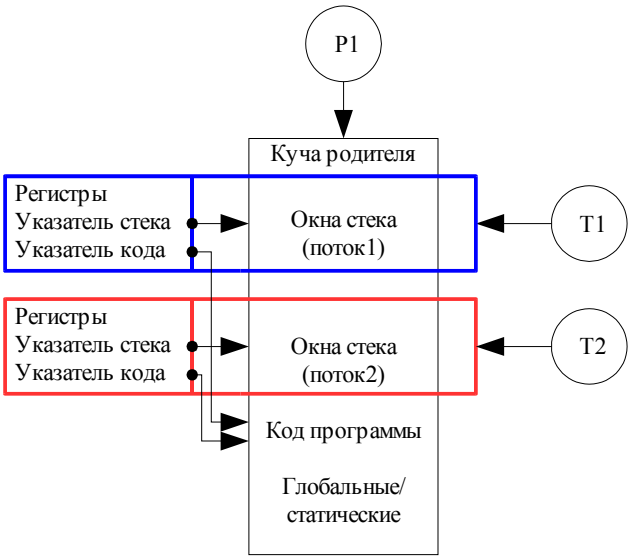


Рис. 13. Легковесные процессы (потоки)

Существует два типа потоков: потоки пространства пользователя и пространства ядра.

- Потоки пространства пользователя (user-space threads)
  Пространство пользователя не использует ядро и управляет таблицами самостоятельно. Часто это называется “кооперативной многозадачностью”, когда задача определяет набор подпрограмм, на которые происходит переключение при манипуляции указателем стека. Типично каждый поток отказывается от выполнения явно вызывая переключение, посылая сигнал или выполняя операцию, которая приводит к переключению. Пользовательские потоки обычно могут переключаться быстрее, чем потоки ядра.
- Потоки пространства ядра (kernel-space threads)
  Потоки пространства ядра чаще всего реализуются в ядре, используя несколько таблиц (каждая задача получает таблицу потоков). В этом случае ядро планирует каждый поток с тем же квантом времени, что и процесс. При этом существуют несколько большие накладные расходы при переключении из контекста пользователя в ядро и обратно и при загрузке больших контекстов, но первоначальные измерения производительности показали ничтожное увеличение времени. Так как тик часов определяет время переключения, то менее вероятно, что задача заберёт квант времени у других потоков в пределах задачи.

Linux поддерживает мультипотоковость пространства ядра, используя системный вызов clone():

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

где `fn` – подпрограмма потока, `child_stack` – указатель на стек потока, `flags` см. в `man clone`, `arg` – параметры, ожидаемые потоком

`clone()` – это библиотечная функция, в основе которой лежит системный вызов `clone()`. Она используется для реализации нескольких потоков управления в программе, которые выполняются одновременно в разделяемом адресном пространстве.

Когда дочерний процесс порождается при помощи `clone()`, она вызывает

функцию приложения `fn(arg)`. Аргумент `child_stack` определяет местоположение стека, используемого дочерним потоком. Так как поток и вызывающий процесс могут разделять память, для дочернего потока невозможно выполнение на том же стеке, что и вызывающий процесс. Поэтому последний должен установить область памяти для дочернего стека и передать указатель на эту область. Стек растёт вниз, поэтому дочерний стек обычно указывает на самый верхний адрес области памяти, установленной для дочернего стека. Обычно он выделяется из секции кучи родительского процесса используя `malloc()`.

Функциональная диаграмма

На рис. 14 представлена общая программная структура migshm и взаимодействие между элементами системы.

Функциональное описание каждого модуля системы:

- Миграция процессов с разделяемой памятью
  Этот модуль позволяет процессу быть выбранным для миграции в любое время своего выполнения, до или после присоединения к региону разделяемой памяти.
- Модуль взаимодействия
  Этот модуль управляет всей функциональностью по взаимодействию между процессами с разделяемой памятью, распределённых по различным узлам кластера.
- Модуль согласованности
  Этот модуль поддерживает согласованность между реплицированными страницами разделяемой памяти, существующих на различных узлах
- Журналирование обращений и решение о миграции
  Этот модуль управляет пересылкой региона разделяемой памяти. Он также реализует политику сбоя страниц.
- Миграция потоков
  Этот модуль управляет миграцией LWP в кластере

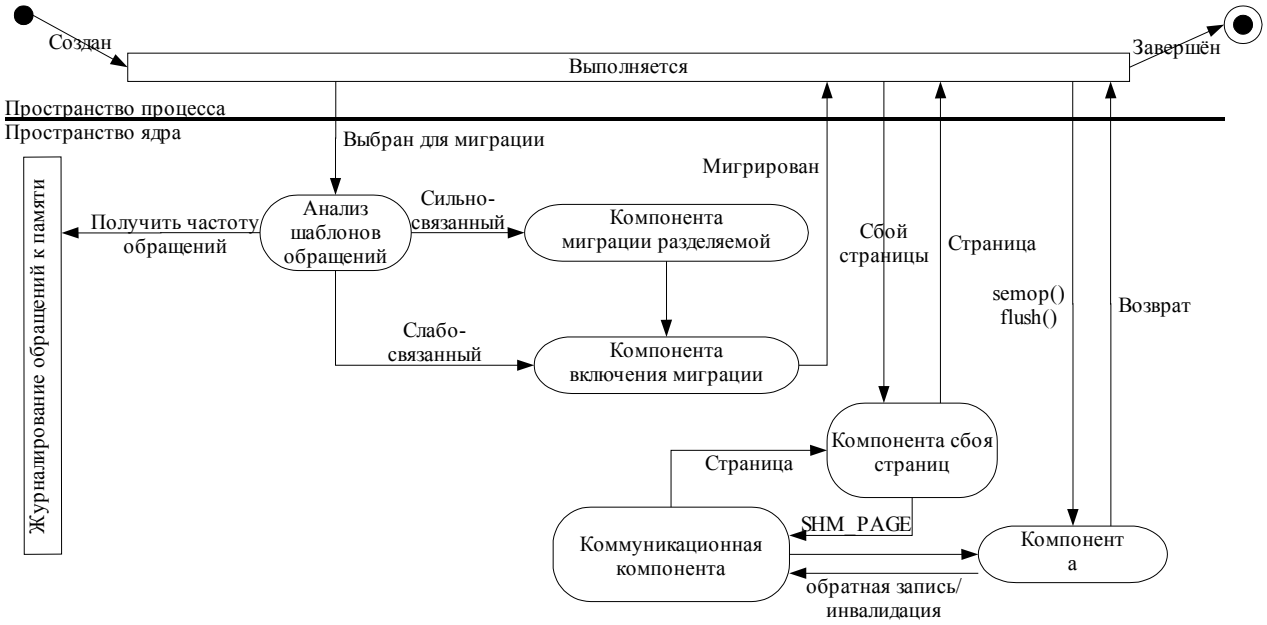


Рис. 14. Диаграмма функционального процесса

Требования к производительности

Система должна быть достаточно масштабируемой. Реализация должна гарантировать, что после миграции процессов, разделяющих память, должна быть полностью обеспечена согласованность разделяемой памяти. Производительность не должна падать по сравнению с немодифицированным openMosix, в котором процессы с разделяемой памятью не мигрируют.

4.4. Требования к приложению

К приложению предъявляются следующие требования:

- Приложения реального времени не должны использовать migshm. Производительность приложений, которые производят интенсивный ввод-вывод может упасть.
- Приложение с разделяемой памятью должно использовать семафоры для синхронизации между процессами, использующих разделяемую память.
- В любой момент выполнения процесс может быть присоединён только к одной области разделяемой памяти.
- При миграции потоков процесс-родитель должен оставаться на домашнем узле (будет заблокирован), в то время как процессы-потомки могут мигрировать.
- Родительские и дочерние процессы не могут воспользоваться системным вызовом malloc() после вызова clone(); процесс-родитель может использовать malloc() до clone().

В обоих случаях: при миграции процессов, использующих SYSV разделяемую память или потоки, – подсистема миграции прозрачным образом обеспечивает

согласованность разделяемых данных. Также ведётся статистика по частоте обращения к разделяемой памяти. Таким образом, в зависимости от использования разделяемой памяти и общей нагрузки кластера, migshm самостоятельно мигрирует разделяемую память и группы процессов.

Включение миграции процессов с разделяемой памятью

Балансировщик нагрузки openMosix не рассматривает все типы приложений для миграции. Например, демоны; процесс init; процессы, использующие kiobuf; процессы, использующие планирование реального времени; процессы, использующие разделяемую память; LWP (потоки) и т.п. не мигрируют.

Для включение миграции процессов с разделяемой памятью, мы должны убедиться, что openMosix рассматривает эти процессы для миграции. openMosix должен быть осведомлён об этих процессах.

Должна быть сохранена следующая информация о процессе с разделяемой памятью:

- уникальный идентификатор для идентификации процесса
- информация о виртуальном адресе региона разделяемой памяти, к которому присоединён процесс
- флаг для идентификации, является ли процесс сильно или слабо связанным

Должны быть зафиксированы некоторые атрибуты самой разделяемой памяти, такие как:

- уникальный идентификатор региона разделяемой памяти
- счётчики обращений к разделяемой памяти всеми процессами, присоединённых к ней
- порог миграции разделяемой памяти

Всякий раз, когда процесс вызывает `shmget()`, инициализируется информация о разделяемой памяти, и всякий раз, когда процесс присоединятся к разделяемой памяти, инициализируется информация о процессе.

Управление памятью

Есть два основных положения по управлению памятью:

1. openMosix регулирует управлением памятью только для приложений, не использующих разделяемую память. Когда процесс мигрирует на удалённый узел, его карты памяти и страницы, вызвавшие “сбой страницы” уничтожаются на UNH и восстанавливаются на удалённом узле. Это управление памятью не достаточно для процессов с разделяемой памятью. Когда мигрирует такой процесс, нельзя уничтожать страницы разделяемой памяти на UNH, потому что могут быть другие процессы, присоединённые к той же разделяемой памяти. Управление памятью должно быть изменено для того, чтобы openMosix сохранял информацию об этих

страницах разделяемой памяти.

2. Когда два процесса с разделяемой памятью находятся на одном узле, они разделяют те же физические страницы разделяемой памяти. То есть, когда эти процессы мигрируют на один и тот же удалённый узел, openMosix устанавливает новые страницы для каждого мигрировавшего процесса. Вместо этого процессы, присоединённые к региону разделяемой памяти, должны разделять физические страницы разделяемой памяти также и на удалённом узле.

Миграция страниц разделяемой памяти приводит к “краже” страниц у файловой системы tmpfs.

Миграция процессов с разделяемой памятью в любой момент выполнения программы

Реализация системных вызовов разделяемой памяти Sys V IPC выполнена для систем с одним процессором. После миграции процесса, последний должен иметь возможность выполнить вся связанные с разделяемой памятью функции на удалённом узле, поскольку на этом узле уже установлена карта памяти.

Можно убедиться, что вызовы функций shmget(), shmat(), shmdt() могут быть выполнены удалённо прозрачным образом:

`shmget()`. `shmget()` – это системный вызов, который выполняется на UNH: создаётся запись в массиве IPC, результирующий идентификатор разделяемой памяти возвращается на удалённый узел.

- `shmat()`. После миграции карта памяти процесса находится на удалённом узле. В течение выполнения `shmat()` вся информация, связанная с файлами, ассоциированными с вновь создаваемым регионом, извлекается из UNH. Используя это, регион памяти фактически создаётся на удалённом узле и становится частью адресного пространства процесса.
- `shmdt()`. Регион разделяемой памяти отсоединяется от адресного пространства процесса на удалённом узле. Также на UNH передаётся информация о процессе для того, чтобы отслеживать количество присоединённых процессов.

Коммуникационный модуль

Для миграции процессами в кластере openMosix, необходим коммуникационный канал между представительским (deputy) и удалённым (remote) контекстом. Это осуществляется установлением специального TCP соединения. Тем не менее, когда необходимо взаимодействие двух процессов на различных узлах, этот коммуникационный канал не может быть использован.

Для преодоления этой проблемы был реализован демон MigSharedMemD. Этот демон активизируется на каждом узле кластера после включения openMosix и слушает на порту 0x3418. Он ответственен за принятие сообщений, которые

идентифицируются при помощи типа заголовка:

Тип заголовка	Модуль
WRITEBACK, INVALIDATE	Согласованности
SHM_OWNER_BROADCAST	Миграции разделяемой памяти
SHM_PAGE	Сбоя страниц
SYNC_DATA	Миграции потоков

Модели согласованности

Когда процессы, использующие разделяемую память, мигрируют на различные узлы, они работают с локальными копиями разделяемой памяти. Поэтому должна поддерживаться согласованность между различными локальными копиями одной и той же разделяемой памяти. Итак, если процессы с разделяемой памятью мигрируют и будут разбросаны по кластеру openMosix, возникнут конфликты между удалёнными процессами. Мыдолжны быть уверены, что читатель всегда прочитает самую последнюю копию данных, а модификации, сделанные писателями, передаются всем удалённым читателям/писателям.

Далее следуют подходы, рассматриваемые для согласованности :

1. Строгая согласованность
   Наиболее жёсткая модель согласованности называется строгой согласованностью. Она определяется следующим условием: “Любое чтение из ячейки памяти X возвращает наиболее недавнюю операцию записи в X”. Это определение неявно подразумевает существование абсолютного глобального времени, для того, чтобы определение “наиболее недавнего” не было неоднозначным. Следовательно, когда память строго согласована все записи мгновенно видны всем процессам и поддерживается упорядоченность согласно абсолютному глобальному времени. Если изменяется ячейка памяти, все последующие чтения из этой ячейки видят новое значение, не важно, как скоро после изменения выполнены эти чтения и не важно, какой процесс выполняет чтение, и где он расположен. Аналогично, если выполнено чтение, оно получает текущее значение, не важно, как быстро произошла следующая запись.
2. Причинная согласованность (causal consistency)
   Модель причинной согласованности представляет собой ослабленную последовательную согласованность в том плане, что она различает события, потенциально причинно связанные, и нет. Рассмотрим на примере памяти. Предположим, что процесс P1 пишет переменную X. После этого P2 читает X и пишет Y. Здесь чтение X и запись Y потенциально причинно связанны, потому что вычисление Y может зависеть от значения X, прочитанного P2 (т.е. от значения, записанного P1). С другой стороны, если два процесса спонтанно и одновременно пишут два значения, они не

причинно связаны. Если есть чтение, за которым следует запись, то эти два события потенциально причинно связаны. Аналогично, чтение причинно связано с записью, которая обеспечила чтение полученными данными. Операции, которые не являются причинно связанными, называются одновременными. Для того, чтобы память была причинно согласованной, необходимо чтобы память подчинялась следующим условиям:

- Записи, которые потенциально причинно связанны, должны быть видимы всеми процессами в том же порядке.
- Одновременные записи могут быть видны в различном порядке на разных узлах.

Согласованность желающего освобождения (eager release consistency)

Процесс задерживает распространение своих изменений в разделяемых данных, пока он не подойдёт к освобождению. Получение и освобождение – две явные операции для синхронизации, необходимые в модели согласованности освобождения, которые соответствуют блокирующему получению и блокирующему освобождению. Таким образом, при освобождении он распространяет модификации среди других процессоров, которые кэшируют модифицированные страницы. Для протокола инвалидации это влечёт за собой рассылку инвалидаций всех модифицированных страниц другим процессам, которые кэшируют эти страницы. Для того, чтобы уменьшить количество обмениваемых данных, протокол обновления высылает разницу (diff) для каждой модифицированной страницы другим кэшам. Разница описывает модификации, сделанные в странице, которые затем объединяются с другими закэшированными копиями. В любом случае освобождение блокируется до получения подтверждения от других кешей.

Изменения, произведённые писателем, сбрасываются в копию владельца только при освобождении блокировки на разделяемую память.

Согласованность ленивого освобождения (lazy release consistency)

Наиболее часто используемая модель согласованности в программном обеспечении DSM – это согласованность ленивого освобождения, в которой инвалидации рассылаются во время получения, вместо времени освобождения, как в модели согласованности желающего освобождения.

В migshm использована модель согласованности желающего освобождения. Это модель, в которой писатель проталкивает локальные копии модифицированных страниц региона разделяемой памяти её владельцу только тогда, когда он освобождает блокировку на память, а не при каждой записи. Эта модель гарантирует то, что узел-владелец разделяемой памяти всегда имеет последнюю копию разделяемой памяти. Таким образом, когда другой удалённый процесс производит последующий доступ к разделяемой памяти,

вырабатывается сбой страницы, и он получает с узла-владельца новейшие страницы.

Для достижения согласованности были использованы базовые операции обратная запись и инвалидация:

• Обратная запись

Обратная запись инициируется на удалённом узле, когда пишущий процесс на удалённом узле освобождает блокировку. Обратная запись приводит к посылке"гзязных" страниц разделяемой памяти, записанных удалённым узлом, узлу-владельцу разделяемой памяти как раз перед тем. как удалённый писатель освобождает блокировку. На узле-владельце полученные данные синхронизируются с соответствующим адресом памяти. Это гарантирует то, что владелец имеет самую последнюю копию разделяемой памяти. Следовательно, копии узла, на котором выполняется писатель и узла-владельца имеют новейшие данные.

Когда удалённый писатель пишет в страницу разделяемой памяти, он отправляет следующую информацию узлу-владельцу:

1. Содержимое разделяемой памяти для синхронизации.
2. Информацию, связанную с адресом памяти страницы разделяемой памяти для её синхронизации по правильному местоположению.
3. Идентификатор и UHN разделяемой памяти для уникальной идентификации её на UNH.

Узел-владелец получает страницу и синхронизирует её по правильному местоположению в памяти.

• Инвалидация

Операция инвалидации инициируется на узле-владельце разделяемой памяти при следующих событиях:

Получение обратной записи от удалённого узла

Освобождение блокировки локальным писателем

Это влечёт за собой инвалидацию записей таблицы страниц всех удалённых процессов для тех страниц, которые были модифицированы локальным/удалённым писателем. Копия узла-владельца всегда обновляется, и поэтому никакой из локальных процессов не должен быть инвалидирован.

Следующая информация высылается в сообщении инвалидации:

4.5. Использование многопоточных приложений

Поддержка многопоточных приложений является одним из важных достоинств любой кластерной системы. Эта поддержка особенно важна для тяжеловесных приложений, какими являются приложения Java, использование которых в производственных масштабах явно подразумевает использование систем распределённых вычислений для обеспечения требуемого уровня надёжности и производительности. В настоящее время существует несколько

реализаций потоков в Java: “native threads”, “classic threads” и “green threads”. Только при использовании последнего типа потоков приложение может получить выгоду от выполнения в среде openMosix. Две первых реализации явно или неявно используют системную библиотеку pthread. Причиной невозможности уменьшения степени детализации выполнения задачи до потоков, как это делает планировщик задач Linux, является определение окружения, в котором выполняется процесс и поток. Например, после выполнения системного вызова fork() процесс- потомок не может производить запись в область данных потомка, в то время как несколько потоков в пределах одного приложения могут иметь общие переменные для чтения/записи. Следовательно, проблема детализации до уровня потоков аналогична проблеме реализации разделяемой памяти, поддержка которой в openMosix весьма ограничена. Поэтому все попытки использовать потоки в среде openMosix оканчиваются неудачей.

5. ИНСТАЛЛЯЦИЯ OPENMOSIX

5.1. Требования к аппаратной и программной части

Для соединения узлов кластера можно воспользоваться любой технологией (Ethernet, ATM, Token Ring), поддерживаемой ядром Linux. Чаще всего используют 100Мб Ethernet (в силу простоты установки и дешевизны), что будет достаточно для кластеров из небольшого числа узлов. Однако, чем производительнее будет сеть, тем быстрее будет работать ваш кластер. Конечно же, можно установить несколько сетевых адаптеров на ключевые узлы кластера, увеличивая тем самым суммарную пропускную способность, тем не менее, использование Gigabit Ethernet является более предпочтительным. Так же понадобятся высокопроизводительные коммутаторы и источники бесперебойного питания. Самым оптимальным вариантом является использование бездисковых клиентов с возможностью загрузки по NFS (при этом NFS-сервер желательно не включать в кластер для надёжности).

Для установки необходим любой современный дистрибутив Linux, основанный на ядре 2.4.x. Понадобится достаточное место под раздел подкачки (swap) на жёстком диске.

5.2. Планировка кластера

Существует несколько типов пулов серверов и рабочих станций, которые можно построить на базе openMosix:

- Одиночный пул: все сервера и рабочие станции используются в едином кластере. В этом случае карта узлов будет одинаковой на всех станциях.
  Преимущество/недостаток: рабочая станция – часть кластера, и любой узел кластера может использовать все вычислительные ресурсы рабочей станции (при условии, что не задействован механизм ограничений loadlimit или фильтрации miggroup).
- Серверный пул: сервера являются совместно используемыми, в то время как рабочие станции не являются частью кластера. В этом случае карта узлов дублируется только на серверах.
  Недостаток: миграция процессов с рабочих станций не возможна; необходимо заходить на сервер для выполнения задачи.
- Адаптивный пул: сервера являются совместно используемыми, в то время как рабочие станции могут присоединяться к кластеру и покидать его. Рабочие станции пользователей могут использоваться в качестве узлов кластера, например, ночью и в выходные дни. Системы такого типа иногда называют COW (Cluster of Workstations).
- Полудуплексный пул: то же самое, что и адаптивный пул, но рабочие станции сконфигурированы таким образом, что миграция на них будет запрещена, но возможно будет прозрачно использовать вычислительные мощности серверов.

5.3. Сборка openMosix

Возможны следующие способы установки openMosix:

- самостоятельная сборка из исходных текстов ядра
  - использование репозитория, который уже содержит ядро Linux 2.4.20 с внесёнными разработчиками openMosix изменениями
  - использование заранее созданного стабильного патча, который представляет собой совокупность изменений между репозиторием и оригинальным ядром
- использование уже собранных пакетов

Каждый вариант имеет как свои плюсы, так и недостатки. Так, например, если вам необходима неординарная конфигурация ядра, либо вы хотите воспользоваться другими расширениями ядра, поставляемых в виде патчей к ядру сторонними разработчиками, либо вы хотите собрать монолитное ядро, то самостоятельная сборка – единственный возможный путь. Однако он требует понимания процесса сборки, и вы можете столкнуться с трудностями при конфигурировании ядра под имеющуюся аппаратуру.

Используя репозиторий, вы можете получить выгоду от использования дополнительной функциональности, доступной только в разрабатываемой версии. Однако, эта функциональность не является хорошо отлаженной и предсказуемой, поэтому может работать неправильно и негативно воздействовать на другие подсистемы openMosix и стабильность системы в целом.

Использование стабильного патча является наиболее оптимальным и безопасным вариантом, поскольку вся функциональность была хорошо протестирована перед выходом новой версии. Патч можно применить как к оригинальному ядру, исходные коды которого можно взять с сайта http://www.kernel.org, так и к ядру, поставляемому производителем дистрибутива. В последнем случае не исключено возникновение конфликтов между патчем openMosix и модификациями, которые внёс производитель.

Однако если процесс пересборки ядра является для вас затруднительным, вы можете воспользоваться уже скомпилированным ядром. Этот вариант является самым быстрым и приемлемым в большинстве случаев.

5.3.1. Использование репозитория

Здесь и далее значок '#' подразумевает, что команды выполняются под администратором (root), значок '$' – под обычным пользователем, значок '\' – перенос на следующую строку.

```
~# cd /usr/src
/usr/src# export CVSROOT=pserver:anonymous@cvs.sourceforge.net:\ /
cvsroot/openmosix
/usr/src# cvs login
/usr/src# cvs -z3 co linux-openmosix
/usr/src# ln -s linux-openmosix/linux-openmosix linux
```

```
/usr/src# cd linux
/usr/src/linux# make menuconfig
```

Теперь необходимо активизировать необходимую для вас функциональность ядра, а также следующие опции:

- CONFIG_MOSIX=y
- CONFIG_MOSIX_SECUREPORTS=y
- CONFIG_MOSIX_DISCLOSURE=1
- CONFIG_MOSIX_FS=y
- CONFIG_MOSIX_DFSA=y

Теперь можно непосредственно собрать и установить ядро, предварительно изменив параметр INSTALL_PATH= в `/usr/src/linux/Makefile` на своё усмотрение:

```
/usr/src/openmosix/linux-openmosix# make dep bzImage install \
modules modules_install
```

После этого необходимо отредактировать файл `/etc/lilo.conf`, добавить в него новую секцию с путём к новому ядру и перезапустить `lilo`.

5.3.2. Использование стабильного патча

Вам понадобятся следующее файлы:

- исходные коды ядра Linux: ftp://ftp.kernel.org/pub/kernel/v2.4/linux-2.4.20.tar.bz2
- последняя версия стабильного патча для данной версии ядра: http://sourceforge.net/project/showfiles.php?group_id=46729&release_id=136769.

Предположим, что данные файлы находятся в каталоге `/usr/src`:

```
~# cd /usr/src
/usr/src# tar -xjf linux-2.4.20.tar.bz2
/usr/src# mv linux-2.4.20 linux-2.4.20-mosix
/usr/src# ln -s linux-2.4.20-mosix linux
/usr/src# cd linux
/usr/src/linux# zcat ../openMosix-2.4.20-2.gz | patch –p1
patching file arch/i386/config.in
patching file arch/i386/defconfig
patching file arch/i386/kernel/entry.S
patching file arch/i386/kernel/i387.c
patching file arch/i386/kernel/ioport.c
...
/usr/src/linux# make menuconfig
```

Необходимо обратить внимание на то, чтобы в выводе команды `patch` не было строк, содержащих слово FAILED.

Далее последовательность шагов аналогична предыдущему разделу.

5.3.3. Использование собранных пакетов

Понадобятся собранные пакеты с:
http://sourceforge.net/project/showfiles.php?group_id=46729&release_id=136769

Инсталляция очень проста:

```
~# cd /usr/src
/usr/src# rpm -ivh openmosix-kernel-2.4.20.i386.rpm
```

После этого необходимо дополнить конфигурацию менеджера загрузки.

5.4. Установка пользовательских утилит

Аналогично, как и при установке ядра, возможный следующие варианты:

- сборка из исходных кодов, взятых их репозитория
- сборка из исходных кодов, взятых с сайта
- установка заранее собранного пакета

Перед сборкой из исходных кодов убедитесь, что `/usr/include/linux` содержит актуальные заголовочные файлы ядра openMosix или является ссылкой на поддерево `/usr/src/linux/include/linux`:

```
~# cd /usr/include
/usr/include# mv linux linux-old
/usr/include# ln -s /usr/src/linux/include/linux
```

Далее необходимо получить версию из репозитория:

```
~# cd /usr/src
/usr/src# export CVSROOT=pserver:anonymous@cvs.sourceforge.net:\ /
cvsroot/openmosix
/usr/src# cvs login
/usr/src# cvs -z3 co userspace-tools
/usr/src# cd userspace-tools
```

Внести изменения в файл `/usr/src/userspace-tools/configuration` (параметр OPENMOSIX должен указывать на директорию с исходными кодами ядра, например, OPENMOSIX = /usr/src/linux; параметр DESTDIR измените на своё усмотрение)

```
/usr/src/userspace-tools# make all install
```

Инсталляция из исходных кодов аналогична. Для этого потребуется архив `openmosix-tools-0.3.tgz` с http://sourceforge.net/project/showfiles.php?group_id=46729&release_id=152286:

```
~# cd /usr/src
/usr/src# tar -xzf openmosix-tools-0.3.tgz
/usr/src# cd openmosix-tools-0.3
/usr/src/openmosix-tools-0.3# make all install
```

5.5. Файл карты узлов

Файл карты узлов `/etc/hpc.map` используется утилитой `mospe` при инициализации openMosix. Каждая строка этого файла состоит из следующих

полей:

openMosix_ID имя_машины количество_последовательных_узлов

Например, следующий файл:

```
1       192.168.1.1         2
10      192.168.1.10        1
51      192.168.1.50        3
```

конфигурирует узлы:

1 192.168.1.1

2 192.168.1.2

10 192.168.1.10

50 102.168.1.50

51 192.168.1.51

52 192.168.1.52

Задание вместо этого файла строки вида:

```
1       192.168.1.1         250
```

нежелательно, так как будет вызывать продолжительные тайм-ауты при опросе отсутствующих узлов. Дополнительно в этом файле можно указывать IP адреса, которые являются синонимами узлов, с несколькими имеющимися сконфигурированными IP адресами (для узлов с несколькими сетевыми интерфейсами) (см. man mospe).

В будущем планируется исключить использование этого файла и полностью перейти на автообнаружение.

5.6. Файл локально монтируемых файловых систем

Общесистемный файл /etc/fstab содержит список локально-монтируемых файловых систем. В простейшем случае он имеет вид:

```
/dev/hda2  /             reiserfs      defaults 1 2
devpts     /dev/pts      devpts        defaults 0 0
proc       /proc         proc          defaults 0 0
usbdevfs   /proc/bus/usb usbdevfs      noauto   0 0
mosix      /mfs          mfs           dfsa=1   0 0
```

Задание атрибута dfsa=1 разрешает возможность использования DFSA.

6. НАСТРОЙКА И АДМИНИСТРИРОВАНИЕ OPENMOSIX

6.1. openMosix API

6.1.1. Proc интерфейс

Так как openMosix является расширением ядра Linux, то почти вся функциональность находится в ядре. Для того чтобы конфигурировать, администрировать и получать статистическую информацию из ядра, необходим хорошо определённый интерфейс. Наиболее часто используемым способом обеспечения этого является так называемый proc-интерфейс (или sysctl-интерфейс). Этот интерфейс позволяет получать и устанавливать значения ядра с уровня приложения. Для получения более полной информации о файловой системе `/proc` см. документацию к ядру Linux в каталоге `Documentation/filesystems/proc.txt`. Аналогичным образом директория `/proc/hpc` обеспечивает доступ к структурам openMosix в ядре. Она содержит файлы, которые используются для локального конфигурирования, так и информацию об удалённых узлах. Вся информация в `/proc/hpc` обновляется по истечению интервала устаревания (decay-interval), который можно установить в любое время. Таким образом, каждый узел в кластере знает состояние удалённых узлов. Это необходимо для вычисления того, как балансировать нагрузку и процессы по кластеру. Так как эти алгоритмы организованы децентрализованным образом, каждый узел решает сам, должен ли процесс мигрировать к другому узлу. Это уменьшает затраты и обеспечивает линейную масштабируемость вплоть до 1000 узлов. Пользователи и программы могут напрямую взаимодействовать с openMosix через этот интерфейс, так как большинство файлов являются текстовыми.

Локальная информация, доступная через `/proc/hpc/admin/`:

- **block** – разрешить/запретить миграцию удалённых процессов на данный узел
- **bring** – вернуть все мигрировавшие с данного узла процессы (переместить процессы назад на домашний узел)
- **dfsalinks** – список текущих ссылок DFSA
- **expel** – переместить все мигрировавшие на данный узел процессы обратно на свои домашние узлы
- **gateways** – максимальное число шлюзов
- **lstay** – локальные процессы должны остаться (запретить миграцию)
- **mospe** – содержит ID узла openMosix (см. гл. 5.5)
- **nomfs** – разрешить/запретить MFS
- **overheads** – для настройки оптимизации (см. гл. 6.2)
- **quiet** – прекратить сбор информации для балансировки нагрузки
- **decayinterval** – интервал для сбора информации для балансировки нагрузки

- **slowdecay** – по-умолчанию 975
- **fastdecay** – по-умолчанию 926
- **speed** – скорость относительно PentuimIII/1ГГц
- **stay** – разрешить/запретить автоматическую миграцию процессов
- **loadlimit** – ограничение нагрузки (если 0, то функция отключена)
- **llimitmode** – режим ограничения нагрузки; может принимать значения 0, 1, 2:
  0 – общая нагрузка не может быть больше чем **loadlimit**; это значит, что если общая нагрузка становится больше **loadlimit**, то удалённые процессы не могут мигрировать на данный узел, а уже мигрированные процессы будут высылаться на свои домашние узлы (UHN) до те пор, пока общая нагрузка не станет меньше порогового значения **loadlimit**
  1 – удалённая нагрузка, то есть суммарная нагрузка всех удалённых процессов, не может быть больше **loadlimit**
  2 – локальная нагрузка, то есть суммарная нагрузка всех локальных процессов, не может быть больше **loadlimit**
- **cpulimit** – ограничение использования процессора (если 0, то функция отключена, 100 – максимальное значение для однопроцессорной системы)
- **cpulimitmode** – режим ограничения использования процессора; может принимать значения 0, 1, 2 (аналогично, как и для **llimitmode**)
- **config** – (двоичный файл) основной конфигурационный файл, записанный утилитой `mospe`.

Информация о локальных узлах, доступная через `/proc/hpc/nodes/[ID]`, где ID – это идентификатор узла openMosix, заданный при инициализации (см. гл. 5.5):

- **cpus** – количество процессоров на узле
- **load** – нагрузка openMosix для этого узла
- **mem** – доступная память, как это кажется openMosix
- **rmem** – доступная память, как это кажется Linux
- **speed** – скорость узла относительно PIII/1ГГц
- **status** – статус узла
- **tmem** – доступная память
- **util** – использование узла
- **loadlocal** – общая нагрузка всех локальных процессов
- **loadremote** – общая нагрузка всех удалённых процессов
- **loadlimit** – текущее значение ограничения нагрузки
- **llimitmode** – текущее значение режима ограничения нагрузки
- **cpulocal** – общее использованием процессора всеми локальными процессами
- **cpuremote** – общее использованием процессора всеми удалёнными процессами

- **cpulimit** – текущее значение ограничения использования процессора
- **cpulimitmode** – текущее значение режима ограничения использования процессора

Эти директории являются одинаковыми для всех узлов кластера, что позволяет получать информацию о каждом узле с каждого узла.

Дополнительная информация о локальных процессах, доступная через `/proc/[PID]/`:

Доступные для чтения:

- **cantmove** – причина, почему процесс не может мигрировать
- **lock** – установлено, если процесс заблокирован на своём домашнем узле
- **nmigs** – сколько раз процесс мигрировал
- **where** – ID узла, на котором процесс в данное время выполняется

Доступные для записи:

- **lock** – если вы хотите заблокировать процесс на домашнем узле, запишите сюда 1; для того, чтобы процесс мог мигрировать, он должен быть разблокирован (записать 0)
- **goto** – запишите ID узла, на который следует мигрировать процесс
- **migrate** – то же самое, что и **goto**
- **migfilter** – записать 1 для включения механизма фильтрации на основе миграционных групп (miggroup)
- **mignodes** – битовая маска идентификаторов узлов (равно сумме 2^(ID-1), где ID – идентификатор узла openMosix)
- **migpolicy** – задание политики фильтрации; может принимать значения 0 или 1:
  0 – запретить: процесс может мигрировать на все узлы за исключением тех, для которых установлен в единицу соответствующий бит в **mignodes**
  1 – разрешить: процесс может мигрировать на все узлы, для которых установлен в единицу соответствующий бит в **mignodes**

Информация об удалённых процессах, доступная через `/proc/hpc/remote/`:

- **from** – домашний узел процесса
- **identity** – дополнительная информация о процессе
- **statm** – статистика процесса по использованию памяти
- **stats** – статистика процесса по использованию CPU

6.1.2. Интерфейс MFS

Кроме proc-интерфейса, пользователь или программа может получать доступ к файловой системе любого узла в кластере, если смонтирована MFS (см. гл. 5.6). В этой директории доступно несколько ссылок, имеющих следующее значения:

- **here** – указывает на корневую файловую систему текущего узла, на котором

выполняется данный процесс

- **home** – указывает на корневую файловую систему домашнего узла
- **magic** – указывает на корневую файловую систему узла, используемого в последнем системном вызове create() (или open() с опцией "O_CREAT"); в противном случае, указывает на последний узел, на котором был успешно создан «магический» файл MFS (является полезным при создании файла и немедленном его удалении)
- **lastexec** – указывает на корневую файловую систему узла, на котором процесс успешно выполнил системный вызов execve()
- **selected** – указывает на корневую файловую систему узла, которая была выбрана в самом приложении или в каком-либо его предке (перед вызовом fork())

6.1.3. Функциональность пользовательской библиотеки libmosix

openMosix также обеспечивает функциональность через библиотеку `libmosix`, которая входит в состав пользовательских утилит:

- int msx_readval(char *path, int *val);
  прочитать 'val' из 'path' ('path' всегда путь для чтения/записи в `/proc/hpc` интерфейсе)
- int msx_readval2(char *path, int *val1, int *val2);
  прочитать 'val1' и 'val2' из 'path'
- int msx_write(char *path, int val);
  записать 'val' в 'path'
- int msx_write2(char *path, int val1, int val2);
  записать 'val1' и 'val2' в 'path'
- int msx_readnode(int node, char *item);
  прочитать 'item' из 'node' ('item' можеть быть load, speed, cpus, util, status, mem, rmem, tmem)
- int msx_readproc(int pid, char *item);
  прочитать 'item' для специфического 'pid' (эта функция читает информацию для 'pid' из `/proc/[PID]/[item]`, 'item' может быть block, bring, stay…)
- int msx_read(char *path);
  прочитать значение из 'path'
- int msx_writeproc(int pid, char *item, int val);
  записать 'val' в 'item' для процесса 'pid'
- int msx_readdata(char *fn, void *into, int max, int size);
- int msx_writedata(char *fn, char *from, int size);
- int msx_replace(char *fn, int val);
- int msx_count_ints(char *fn);
- int msx_fill_ints(char *fn, int *, int);

API libmosix также содержит функцию для контроля и изменения поведения openMosix:

- int msxctl(msx_cmd_t cmd, int arg, void *resp, int len);
  #define msxctl1(x) (msxctl((x), 0, NULL, 0))
  #define msxctl2(x,y) (msxctl((x), (y), NULL, 0))
  #define msxctl3(x,y,z) (msxctl((x), (y), (z), sizeof(*(z))))
  Возможные команды для cmd:
- D_STAY, /* Разрешить автоматическую миграцию процессов с этого узла */
- D_NOSTAY, /* Запретить автоматическую миграцию процессов с этого узла */
- D_LSTAY, /* Разрешить автоматическую миграцию локальных процессов */
- D_NOLSTAY, /* Запретить автоматическую миграцию локальных процессов */
- D_BLOCK, /* Заблокировать автоматическую миграцию к этому узлу */
- D_NOBLOCK, /* Разрешить автоматическую миграцию к этому узлу */
- D_EXPEL, /* Переместить все удалённые процессы с этого узла */
- D_BRING, /* Вернуть все процессы на этот (домашний) узел */
- D_GETLOAD, /* Получить текущую нагрузку */
- D_QUIET, /* Остановить внутреннюю активность по балансировке нагрузки */
- D_NOQUIET, /* Возобновить внутреннюю активность по балансировке нагрузки */
- D_TUNE, /* Войти в режим настройки */
- D_NOTUNE, /* Выйти из режима настройки */
- D_NOMFS, /* Запретить MFS доступ к этому узлу */
- D_MFS, /* Разрешить MFS доступ к этому узлу */
- D_SETSSPEED, /* Установить стандартную скорость, влияет на D_GETLOAD */
- D_GETSSPEED, /* Получить стандартную скорость (по-умолчанию=1000) */
- D_GETSPEED, /* Получить скорость машины */
- D_SETSPEED, /* Установить скорость машины */
- D_MOSIX_TO_IP, /* Сконвертировать из openMosix ID в IP адрес */
- D_IP_TO_MOSIX, /* Сконвертировать из IP адреса в openMosix ID */
- D_GETNTUNE, /* Получить число настраиваемых параметров ядра */
- D_GETTUNE, /* Получить настраиваемые параметры ядра */
- D_GETSTAT, /* Получить статус openMosix */
- D_GETMEM, /* Получить текущие значения памяти (всего и свободной) */
- D_GETDECAY, /* Получить параметры устаревания */
- D_SETDECAY, /* Установить параметры устаревания */
- D_GETRMEM, /* Получить представление ОСо памяти (свободной и всего) */

- D_GETUTIL, /* Получить использование процессора в % */
- D_SETWHERETO, /* Отправить процесс куда-нибудь */
- D_GETPE, /* Получить ID узла */
- D_GETCPUS, /* Получить количество процессоров */

6.2. Оптимизация работы openMosix

В настоящее время данная функциональность не доступна, т.к. функциональность утилит переписывается под лицензию GPL.

openMosix содержит более 14 параметров для тонкой настройки. Естественно, ручная настройка этих параметров утомительна для каждого узла. Для автоматизации процесса служат утилиты `prep_tune`, которая запускается на одном узле, и `tune_kernel`, которая запускается на втором. В течение нескольких минут их работы не стоит нагружать узлы и/или сеть, иначе выходные параметры будут некорректны. После окончания работы будет создан файл `/tmp/overheads`, готовый для использования:

```
~# cat /tmp/overheads > /proc/hpc/admin/overheads
```

Естественно, данную команду необходимо вставить в один из загрузочных скриптов (например, `/etc/init.d/openmosix`), чтобы инициализация параметров имела место при каждой загрузке.

7. ТЕСТИРОВАНИЕ КЛАСТЕРА

7.1. Функциональное тестирование

- Тест с использованием программы расчёта распределённых ключей RC5-72 (dnetc)

Для запуска приложений воспользуемся следующей командой:

```
/usr/local/bin$ for i in 1 2 3 4 5 6 7 8; do ./dnetc -numcpu 0; done
```

Процесс выполнения представлен на рис. 15. Для получения большей информации по использованию утилит openMosix см. [HOWTO].

Наблюдаем, что все процессы распределились равномерно по узлам

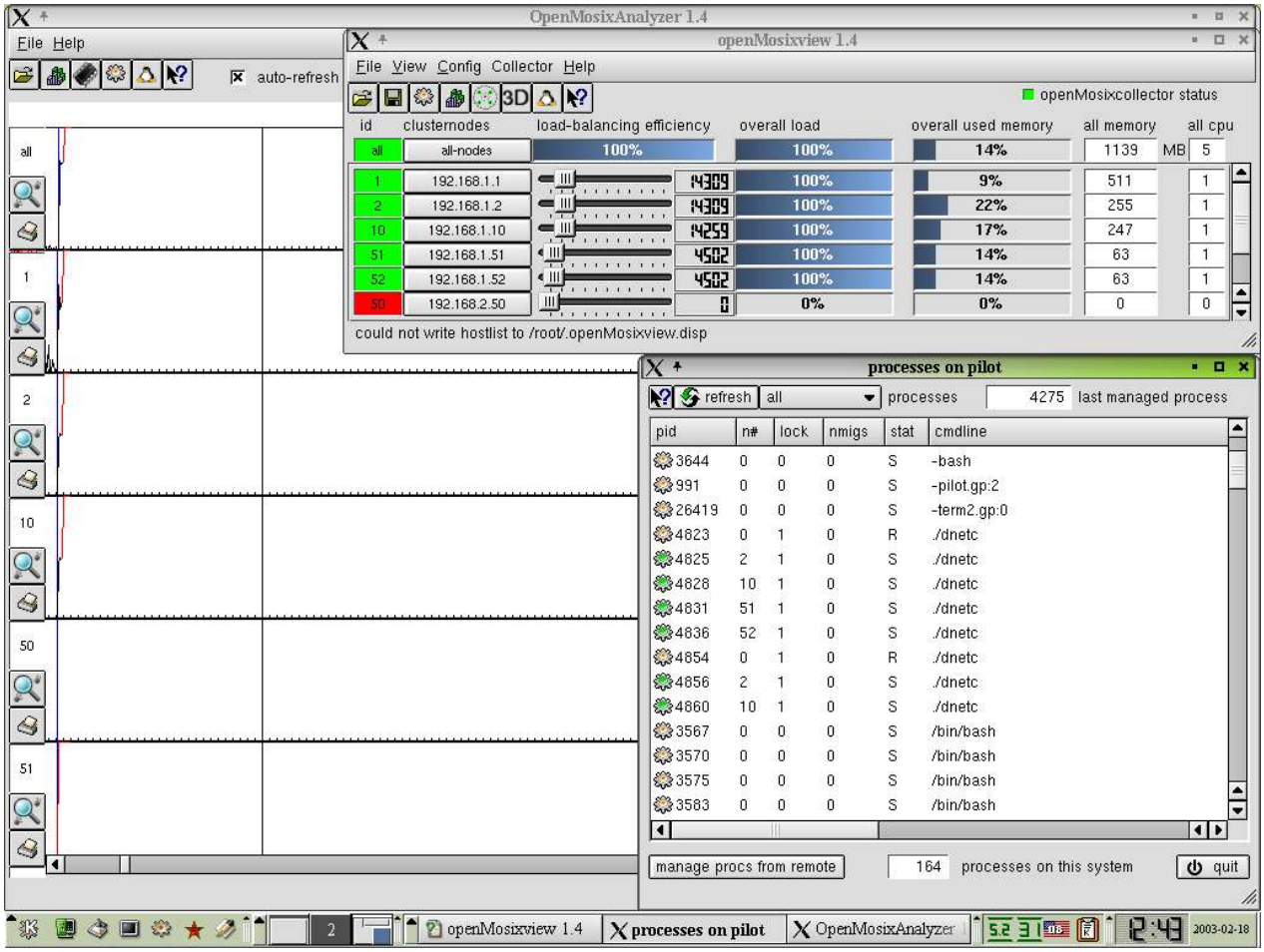


Рис. 15. Выполнение программы dnetc

кластера.

- Тест на ввод-вывод

Для проведения теста скопируем достаточно большой файл на все узлы кластера:

```
/tmp# for i in 10 51 52; do cp 1.wav /mfs/$i/tmp & done
```

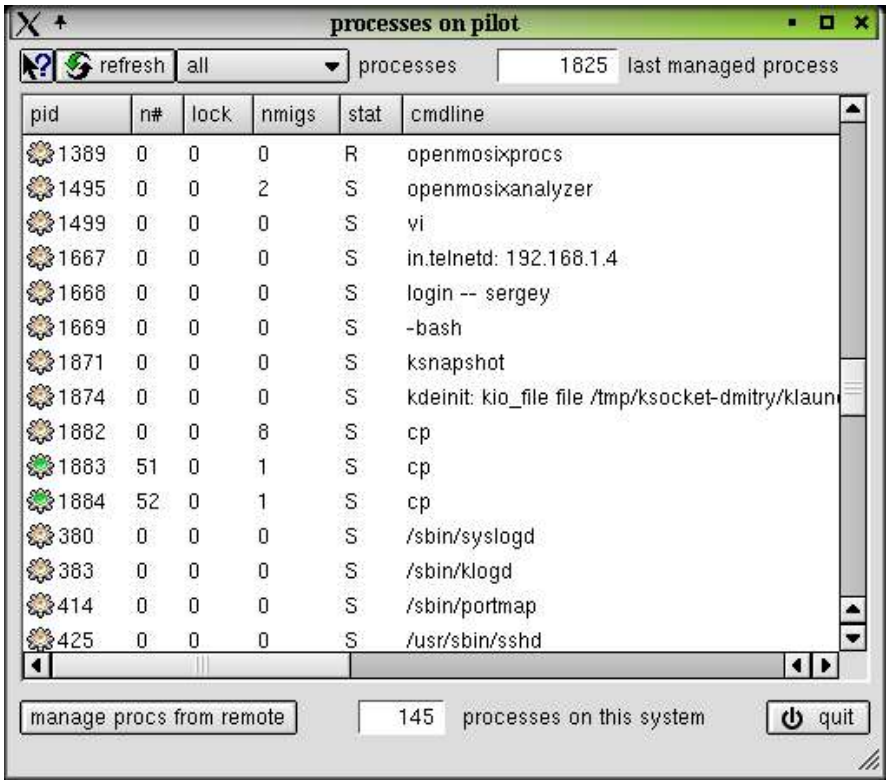


Рис. 16. Процесс выполнения команды cp

Процесс выполнения этих команд представлен на рис. 16:

Как видно из рисунка, процессы cp были мигрированы к тем узлам кластера, на которые производилось копирование. Это не уменьшило нагрузку сети, но позволило разгрузить процессор на узле 1.

- Тест с использованием программы для кодирования аудио (lame)

Для проведения теста воспользуемся уже скопированными файлами и сконвертируем их из wav в mp3 при помощи команды lame:

```
~# for i in 1 1 10 10 51 52
> do
> lame -v -h -S /mfs/here/tmp/1.wav - &> /dev/null &
> done
```

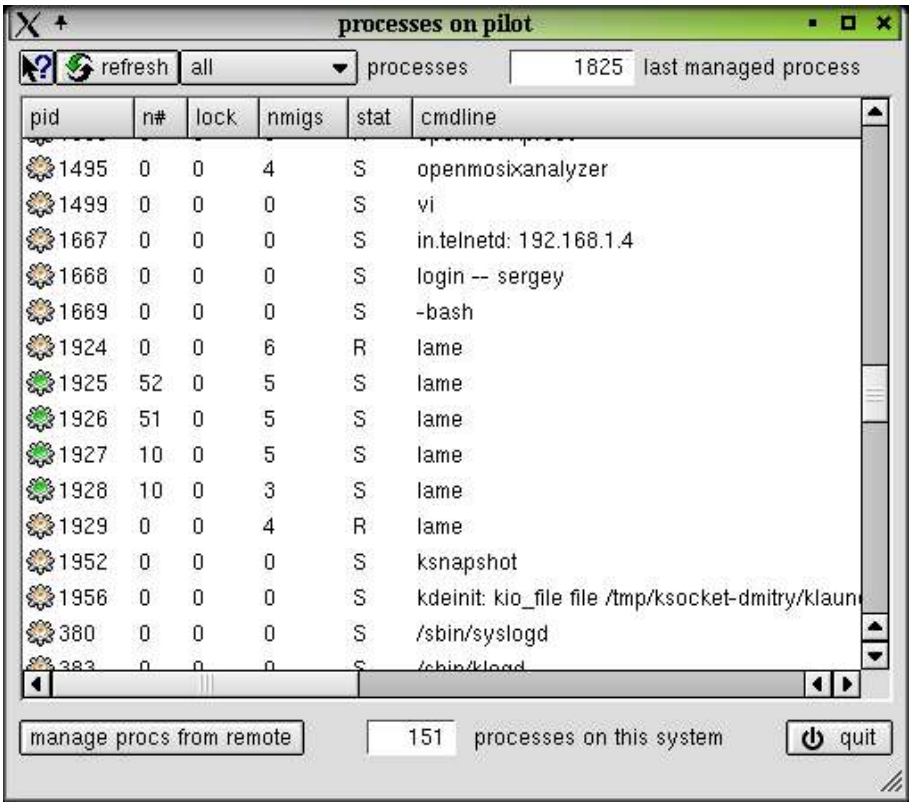


Рис. 17. Процесс выполнения команд lame

Процесс выполнения этих команд представлен на рис. 17. Из рисунка видно, что процессы мигрировали к соответствующим узлам.

7.2. Тестирование общей производительности

В исследовательском центре NASA Ames Research Center был разработан комплекс тестов, позволяющий оценивать производительность суперкомпьютеров. Комплекс тестов NAS состоит из пяти тестов NAS kernel benchmark и трёх тестов, основанных на реальных задачах гидро- и аэродинамического моделирования. Этот круг задач не покрывает всего спектра возможных приложений, однако на сегодняшний день NAS Benchmarks является лучшим общепризнанным комплексом тестов для оценки параллельных многопроцессорных систем, что, собственно, и подтверждается практическими наблюдениями – результатами TOP500.

Комплекс тестов NAS Benchmarks kernel включает следующие расчётные задачи:

1. EP (Embarrasingly Parallel). Вычисление интеграла методом Монте-Карло – тест “усложнённого параллелизма” для измерения первичной вычислительной производительности плавающей арифметики. Это – тест минимального межпроцессорного взаимодействия, который фактически определяет чисто вычислительные характеристика узла при работе с вещественной арифметикой.
2. MG (3D Multigrid). Тест по решению уравнения Пуассона (“трёхмерная

решётка") в частных производных – требует высокоструктурированной организации взаимодействия процессоров. Тестирует возможности системы выполнять как дальние, так и короткие передачи данных.

3. CG (Conjugate Gradient). Вычисление наименьшего собственного значения больших, разрежённых матриц методом сопряжённых градиентов. Это типичное неструктурированное вычисление на решётке, и поэтому тест применяется для оценки скорости передачи данных на длинные расстояния при отсутствии какой-либо регулярности.
4. FT (fast Fourier Transformation). Вычисление методом быстрого преобразования Фурье трёхмерного уравнения в частных производных. Данная задача используется как "серьёзный" тест для оценки эффективности взаимодействия по передаче данных между удалёнными процессорами. При создании программы, реализующей данный тест, могут использоваться библиотечные модули преобразования Фурье различной размерности.
5. IS (Integer Sort). Тест выполняет сортировку целых чисел и используется как для оценки возможностей работы системы с целочисленной арифметикой (главным образом одного узла), так и для выявления потенциала компьютера по выполнению межпроцессорного взаимодействия.

Комплекс тестов NAS Benchmarks по модельным задачам включает следующие модули:

1. LU (LU Solver). Тест выполняет вычисления, связанные с определённым классом алгоритмов (INS3D-LU по классификации центра NASA Ames), в которых решается система уравнений с равномерно-разрежённой блочной треугольной матрицей 5x5.
2. SP (Scalar Pentadiagonal). Тест выполняет решение нескольких независимых систем скалярных уравнений – пентадиагональные матрицы с преобладанием недиагональных членов.
3. BT (Block Tridiagonal). Решение серии независимых систем уравнений – блочные трёхдиагональные матрицы 5x5 с преобладанием недиагональных элементов.

Для установки нужно зарегистрироваться на сайте The NAS Parallel Benchmarks (http://www.nas.nasa.gov/Software/NPB/) и скачать файл с исходными текстами тестов NPB2.4.tar.gz.

```
$ tar -xzf NPB2.4.tar.gz
$ cd ./NPB2.4/NPB2.4-MPI
```

После этого отредактировать файл `./config/make.def` так, чтобы он был аналогичен следующему:

```
MPIF77 = mpif77
FLINK  = mpif77
#FMPI_LIB =
#FMPI_INC =
FFLAGS  = -O3
FLINKFLAGS =
```

```
MPICC = mpicc
CLINK = mpicc
#CMPI_LIB =
#CMPI_INC =
CFLAGS  = -O3
CLINKFLAGS =

# include ../config/make.dummy

CC  = cc -g
BINDIR  = ../bin
RAND   = randdp
```

Убедитесь, что mpif77 и mpicc доступны из вашего $PATH. В противном случае нужно выполнить:

```
$ export PATH=$PATH:/opt/mpich/bin
```

Для удобства можно создать файл ./config/suite.def следующего содержания:

```
# Use tabs
bt    W    9
ep    W    9
cg    W    9
is    W    9
lu    W    9
mg    W    9
sp    W    9
```

Этот файл будет использоваться для сборки пакета тестов. Последний этап – компиляция и запуск тестов (см. приложение 1):

```
$ make suite
$ mpirun -v -np 9 -machinefile machines bt.W.9
```

Для оценки потенциальных возможностей тестируемой конфигурации вычисляется относительная производительность по сравнению с показателями традиционного векторного суперкомпьютера, в качестве которого обычно выступает одна из моделей Cray. Для NAS kernel benchmark определяются четыре класса тестов: класс S, класс A, класс B и класс C, которые фактически отличаются “размерностью” вычислений. Размер задач из класса B превосходит размер задач из класса A примерно в четыре раза. Результаты тестирования в классе A нормируются на производительность однопроцессорного компьютера Cray Y-MP, а класса B – на однопроцессорный Cray C90.

Тестирование производилось на кластере со следующей конфигурацией:

Узел Pilot

cpu: AMD Athlon(tm) processor, 952.189 MHz (1900.54 bogomips), 256 KB cache mem: 1536676 KB, 453716 KB used, 1082960 KB free via: South Bridge: VIA vt82c686b, highest DMA rate: UDMA100, PCI clock: 33.3MHz hda: ST340016A, 38166 GB (78165360 sectors), CHS=4865/255/63, 2048 KB cache hda: buffered disk read timings: 27.83 MB/sec hda: buffer-cache read timings: 64.65 MB/sec
Узел Windows
cpu: AMD Athlon(tm) processor, 949.991 MHz (1893.99 bogomips), 256 KB cache mem: 247908 KB, 21252 KB used, 226656 KB free via: South Bridge: VIA vt82c686b, highest DMA rate: UDMA100, PCI clock: 33.3MHz hda: ST340016A, 38166 GB (78165360 sectors), CHS=4865/255/63, 2048 KB cache hda: buffered disk read timings: 40.51 MB/sec hda: buffer-cache read timings: 168.42 MB/sec
Узел Mech
cpu: AMD Athlon(tm) processor, 952.194 MHz (1900.54 bogomips), 256 KB cache mem: 770200 KB, 74848 KB used, 695352 KB free via: South Bridge: VIA vt82c686b, highest DMA rate: UDMA100, PCI clock: 33.3MHz hda: ST320011A, 19092 GB (39102336 sectors), CHS=2434/255/63, 2048 KB cache hda: buffered disk read timings: 41.03 MB/sec hda: buffer-cache read timings: 162.03 MB/sec
Узел Term1
cpu: Celeron (Mendocino), 300.018 MHz (598.01 bogomips), 128 KB cache mem: 449084 KB, 18512 KB used, 430572 KB free hda: FUJITSU MPC3032AT, 3093 GB (6335280 sectors), CHS=785/128/63, 0 KB cache hda: buffered disk read timings: 12.01 MB/sec hda: buffer-cache read timings: 98.46 MB/sec
Узел Term2
cpu: Celeron (Mendocino), 300.017 MHz (598.01 bogomips), 128 KB cache mem: 449084 KB, 31928 KB used, 417156 KB free hda: FUJITSU MPC3032AT LP, 3093 GB (6335280 sectors), CHS=785/128/63, 0 KB cache hda: buffered disk read timings: 11.70 MB/sec hda: buffer-cache read timings: 97.71 MB/sec

Для тестирования данного кластера применялись тесты класса A. Все тесты были скомпилированы для запуска на четырёх и восьми узлах. В качестве коммуникационной среды использовался Fast Ethernet. Результаты тестов и их сравнение представлены в таблице 7. В приложении 1 представлен сценарий, использованный для запуска всех тестов.

	Имя теста	Класс S, 4 процесса				
		Размерность задачи	Время выполнения, с.		Всего Mop/с.	
MPI+OpenMosix	bt	12x12x12	2,15	3,7%	106,11	3,8%
	cg	1400	1,96	5,1%	33,96	5,6%
	ep	33554432	8,15	-24,7%	4,11	-19,7%
	is	65536	0,12	-133,3%	5,69	-58,2%
	lu	12x12x12	3,70	78,1%	27,66	363,6%
	mg	32x32x32	0,17	-23,5%	44,18	-16,5%
	sp	12x12x12	1,75	-20,0%	55,13	-14,4%
			18,00	2,8%	276,84	31,5%
MPI	bt	12x12x12	2,07	-3,9%	110,13	-3,7%
	cg	1400	1,86	-5,4%	35,86	-5,3%
	ep	33554432	10,16	19,8%	3,30	24,5%
	is	65536	0,28	57,1%	2,38	139,1%
	lu	12x12x12	0,81	-356,8%	128,22	-78,4%
	mg	32x32x32	0,21	19,0%	36,88	19,8%
	sp	12x12x12	2,10	16,7%	47,21	16,8%
			17,49	-2,9%	363,98	-23,9%

	Имя теста	Класс S, 8/9 процессов				
		Размерность задачи	Время выполнения, с.		Всего Mop/с.	
MPI+OpenMosix	bt	12x12x12	3,48	-9,5%	65,56	-8,6%
	cg	1400	3,03	-11,6%	22,00	-10,4%
	ep	33554432	10,96	53,6%	3,06	115,7%
	is	65536	0,18	-83,3%	3,63	-44,6%
	mg	32x32x32	0,28	-21,4%	27,24	-17,5%
	sp	12x12x12	3,40	-9,7%	28,46	-9,0%
			21,33	21,8%	149,95	-8,9%
MPI	bt	12x12x12	3,81	8,7%	59,89	9,5%
	cg	1400	3,38	10,4%	19,71	11,6%
	ep	33554432	5,08	-115,7%	6,60	-53,6%
	is	65536	0,33	45,5%	2,01	80,6%
	mg	32x32x32	0,34	17,6%	22,48	21,2%
	sp	12x12x12	3,73	8,8%	25,90	9,9%
			16,67	-28,0%	136,59	9,8%

	Имя теста	Класс A, 4 процесса				
		Размерность задачи	Время выполнения, с.		Всего Mop/с.	
MPI+OpenMosix	bt	64x64x64	2104,01	-246,2%	79,98	-94,3%
	cg	14000	39,82	35,3%	37,58	54,5%
	ep	536870912	72,96	-122,6%	7,36	-55,0%
	is	8388608	17,03	-91,4%	4,93	-47,9%
	lu	64x64x64	946,47	33,6%	126,04	50,6%
	mg	256x256x256	75,78	-4583,3%	51,37	-95,2%
	sp	64x64x64	1079,64	24,0%	78,74	31,5%
			4335,71	-188,3%	386,00	-5,6%
MPI	bt	64x64x64	7283,12	71,1%	4,53	1665,6%
	cg	14000	25,78	-54,5%	58,05	-35,3%
	ep	536870912	162,43	55,1%	3,31	122,4%
	is	8388608	32,59	47,7%	2,57	91,8%
	lu	64x64x64	628,36	-50,6%	189,86	-33,6%
	mg	256x256x256	3549,02	97,9%	2,47	1979,8%
	sp	64x64x64	820,71	-31,5%	103,58	-24,0%
			12502,01	65,3%	364,37	5,9%

Таблица 7. Результаты тестов NAS

Из результатов видно, что отставание алгоритмов балансировки нагрузки

openMosix от циклической стратегии распределения незначительно для процессов, среднее время выполнения которых меньше минуты. openMosix значительно выигрывает при распределении долговременных и требовательных заданий между гетерогенными узлами кластера.

Далее проведём тест с использованием программы из приложения 2. Тест компилируется командой:

```
$ mpicc mm.c -lm -o mm
```

Усреднённые результаты тестов по 10 запускам приведены в таблице 8.

	4 процесса		8 процессов	
	Время выполнения, с.		Время выполнения, с.	
MPI+OpenMosix	35,26	-34,88%	158,56	54,64%
MPI	47,57	25,86%	71,93	-120,46%

Таблица 8. Результаты теста программы перемножения матриц

ВЫВОДЫ

В данной работе была представлена мультикомпьютерная система openMosix для масштабируемых кластеров из PC и её эффективность при выполнении нескольких больших параллельных прикладных программ. В гл. 5.3 приведён пример построения вычислительного кластера на её основе. Основной парадигмой openMosix является то, что при написании параллельных программ можно придерживаться обычного Unix программирования. В то же время, программы, использующие такие библиотеки, как PVM и MPI, могут получить значительное преимущество при использовании в гетерогенных кластерах.

В гл. 7.2 разработана методика тестирования и произведены тесты на основе тестов NAS Benchmark. Исследования показали целесообразность использования openMosix в качестве альтернативы дорогих SMP-систем за счёт своей открытости и доступности. Производительность компьютерного кластера openMosix демонстрирует хорошее использование ресурсов, относительно хорошее увеличение быстродействия при масштабировании конфигурации и конкурентоспособные результаты при сравнении с другими системами [BAR2].

Главный результат проекта openMosix – это то, что возможно построить дешёвый, масштабируемый компьютерный кластер из потребительских компонент типа PC, UNIX и PVM, как альтернативу традиционных универсальных ЭВМ (mainframes) и MPP. Главное преимущество openMosix по сравнению с другими компьютерными кластерными средами – его способность отвечать во время выполнения на непредсказуемые и беспорядочные запросы/требования к ресурсам от многих пользователей, например, время выполнения, использование памяти или число процессов, – openMosix хорошо адаптируется ко всем таким случаям. Также openMosix предлагает удобную среду общего назначения для выполнения крупномасштабных требовательных последовательных и параллельных прикладных программ.

Проект openMosix расширяется в нескольких направлениях. Во-первых, разрабатываются новые конкурентные алгоритмы для адаптивного управления ресурсами, которые могут работать с различным видом ресурсов, например, нагрузкой, памятью, IPC и вводом – выводом. Также исследуются алгоритмы для сетевой RAM, в которых большой процесс может использовать доступную память на нескольких узлах. Дополнительно разрабатываются расширения к JAVA для поддержки адаптивной миграции объектов подобно алгоритмам openMosix. Кроме того, рассматривается перенос openMosix на различные OS (FreeBSD, MacOS).

СПИСОК ЛИТЕРАТУРЫ

3. MOSIX: MOSIX official web site, http://www.mosix.cs.huji.ac.il/
4. oMosix: openMosix official web site, http://www.openmosix.org/
5. BAR1: A. Barak, MOSIX – Scalable Cluster Computing for Linux, http://www.mosix.cs.huji.ac.il/slides/index.htm
6. BAR5: Barak A. and Braverman A., Memory Ushering in a Scalable Computing Cluster, http://www.mosix.cs.huji.ac.il/ftps/usher.ps.gz
7. BAR4: Barak A., La'adan O. and Shiloh A., Scalable Cluster Computing with MOSIX for LINUX, http://www.mosix.cs.huji.ac.il/ftps/mosix4linux.ps.gz
8. AMIR1: Amir Y., Awerbuch B., Barak A., Borgstrom R.S. and Keren A., An Opportunity Cost Approach for Job Assignment in a Scalable Computing Cluster, http://www.mosix.cs.huji.ac.il/ftps/opc.ps.gz
9. AMAR1: Amar L., Barak A., Eizenberg A. and Shiloh A., The MOSIX Scalable Cluster File Systems for LINUX, http://www.mosix.cs.huji.ac.il/ftps/mfs.ps.gz
10. BAR6: A. Barak, A. Braverman, I. Gilderman, and O. Laden, Performance of PVM with the MOSIX Preemptive Process Migration, June 1996
11. AWER1: B. Awerbuch, Y. Azar and A. Fiat, Packet Routing via Min-Cost Circuit Routing, 1996
12. BART1: Y. Bartal, A. Fiat, H. Karloff and R. Vohra, New algorithms for an ancient scheduling problem, 1992
13. ASPN1: J. Aspnes, Y. Azar, A. Fiat, S. Plotkin and O. Waarts, On-Line Machine Scheduling with Applications to Load Balancing and Virtual Circuit Routing, May 1993
14. AWER2: B. Awerbuch, Y. Azar, S. Plotkin and O. Waarts, Competitive Routing of Virtual Circuits with Unknown Duration, January 1994
15. BAL1: M. Harchol-Balter and A.B.Downey, Exploring Process Lifetime Distributions for Dynamic Load Balancing, May 1996
16. BAR3: A. Barak, S. Guday, and R.G. Wheeler, The MOSIX Distributed Operating System, Load Balancing for UNIX, 1992
17. HOWTO: Kris Buytaert et.al., openMosix-HOWTO, http://howto.ipng.be/openMosix-HOWTO/
18. BAR2: Barak A. and La'adan O., The MOSIX Multicomputer Operating System for High Performance Cluster Computing, March 1998

ПРИЛОЖЕНИЕ 1

Сценарий для проведения тестов NAS benchmark

```
#!/bin/bash

function cleanup()
{
     for node in `cat machines`
     do
          ssh root@$node "killall mpirun $cmd 2> /dev/null"
     done
}
function openmosix()
{
     mode=0
     if [ $1 = "stop" ]; then mode=1; fi

     for node in `cat machines`
     do
          ssh root@$node <<EOM 2> /dev/null
#/etc/init.d/openmosix $*
echo $mode > /proc/hpc/admin/lstay
EOM
     done
}

function run_bench()
{
     echo "Benching class \"$CLASS\", $PROC procs, $RUN run" >> nas_$RUN.out
     for cmd in bt.$CLASS.$PROC mg.$CLASS.$PROC
     do
          CMD="mpirun -np $PROC $* $cmd"
          echo -n "Running \"$CMD\"... "
          $CMD &> ${cmd}_$RUN.out
          echo -ne "${cmd:0:2}\t" >> nas_$RUN.out
perl -ne '
     $found ||= /Benchmark Completed/;
     next if !$found;
     /^ Size\s+=\s+(.*)$/ && (($s = $1) =~ s/\s+//g && print("$s\t") || print
("$s\t")) && next;
     /^ Time in seconds\s+=\s+([.\d]*)$/            && print("$1\t") && next;
     /^ Mop\/s total\s+=\s+([.\d]*)$/               && print("$1\n") && next;
     /^Version\s+=\s+(\w+)$/ && ($1 ne "SUCCESSFUL" && die || exit);
' < ${cmd}_$RUN.out >> nas_$RUN.out
          echo "ok"
          cleanup
     done
}
function run_suite()
{
     #PROC=1 CLASS=S run_bench $*
     PROC=4 CLASS=S run_bench $*
     PROC=8 CLASS=S run_bench $*
     PROC=9 CLASS=S run_bench $*
     PROC=4 CLASS=A run_bench $*
}

function run_all_benches()
{
     openmosix start
     run_suite
     openmosix stop
     run_suite "-machinefile machines"
}
function run_mm_bench()
{
     openmosix start
     time mpirun -np 4 mm
```

```
     time mpirun -np 8 mm
     openmosix stop
     time mpirun -np 4 -machinefile machines mm
     time mpirun -np 8 -machinefile machines mm
}

for RUN in 0 1 2 3 4 5 6 7 8 9
do
     run_mm_bench &> all_mm_$RUN.out
done

for RUN in 1
do
     run_all_benches
done
```

ПРИЛОЖЕНИЕ 2

Программа перемножения матриц с использованием MPI

```
/******************************************************************************
* FILE: mm.c
* DESCRIPTION:
*   In this template code, the master task distributes a matrix multiply
*   operation to numtasks-1 worker tasks.
******************************************************************************/

#include <stdio.h>
#include <math.h>
#include "mpi.h"
#define NRA 600                 /* number of rows in matrix A */
#define NCA 670                 /* number of columns in matrix A */
#define NCB 500                 /* number of columns in matrix B */
#define MASTER 0                /* taskid of first task */
#define FROM_MASTER 1           /* setting a message type */
#define FROM_WORKER 2           /* setting a message type */

MPI_Status status;
main(int argc, char **argv)
{
int numtasks,                   /* number of tasks in partition */
    taskid,                     /* a task identifier */
    numworkers,                 /* number of worker tasks */
    source,                     /* task id of message source */
    dest,                       /* task id of message destination */
    nbytes,                     /* number of bytes in message */
    mtype,                      /* message type */
    intsize,                    /* size of an integer in bytes */
    dbsize,                     /* size of a double float in bytes */
    rows,                       /* rows of matrix A sent to each worker */
    averow, extra, offset,      /* used to determine rows sent to each worker */
    i, j, k,                    /* misc */
    count;
double a[NRA][NCA],             /* matrix A to be multiplied */
       b[NCA][NCB],             /* matrix B to be multiplied */
       c[NRA][NCB];             /* result matrix C */

intsize = sizeof(int);
dbsize = sizeof(double);

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
numworkers = numtasks-1;

/**************************** master task ************************************/
if (taskid == MASTER) {
  printf("Number of worker tasks = %d\n",numworkers);
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j] = i+j;
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j] = i*j;

  /* send matrix data to the worker tasks */
  averow = NRA/numworkers;
  extra = NRA%numworkers;
  offset = 0;
  mtype = FROM_MASTER;
  for (dest=1; dest<=numworkers; dest++) {
    rows = (dest <= extra) ? averow+1 : averow;
    printf("   sending %d rows to task %d\n",rows,dest);
    MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
```

```
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    count = rows*NCA;
    MPI_Send(&a[offset][0], count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    count = NCA*NCB;
    MPI_Send(&b, count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);

    offset = offset + rows;
    }

  /* wait for results from all worker tasks */
  mtype = FROM_WORKER;
  for (i=1; i<=numworkers; i++) {
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    count = rows*NCB;
     MPI_Recv(&c[offset][0], count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD,
&status);

    }

  /* print results */
/*
  printf("Here is the result matrix\n");
  for (i=0; i<NRA; i++) {
    printf("\n");
    for (j=0; j<NCB; j++)
      printf("%6.2f   ", c[i][j]);
    }
  printf ("\n");
*/
  }  /* end of master section */


/***************************** worker task ************************************/
if (taskid > MASTER) {
  mtype = FROM_MASTER;
  source = MASTER;
  printf ("Worker %d started: ", taskid);
  MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  printf ("got offset = %d, rows = %d\n", offset, rows);
  count = rows*NCA;
  MPI_Recv(&a, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
  printf ("Recv: a[1][2] = %0.2f\n", a[1][2]);
  count = NCA*NCB;
  MPI_Recv(&b, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
  printf ("Recv: b[1][2] = %0.2f\n", b[1][2]);
  for (k=0; k<NCB; k++)
    for (i=0; i<rows; i++) {
      c[i][k] = 0.0;
      for (j=0; j<NCA; j++)
        c[i][k] = sin(sqrt(c[i][k]) + sqrt(a[i][j]) * sqrt(b[j][k]));
      }

  mtype = FROM_WORKER;
  printf ("Worker %d complete: ", taskid);
  MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
  MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
  printf ("results send back\n");

  }  /* end of worker */
  MPI_Finalize();
} /* of main */
```