Сбитнев Юрий Иванович, Екатеринбург, Россия

Кластеры

Практическое руководство

Статья посвящена вопросам построения и применения на практике высокопроизводительного параллельного кластерного суперкомпьютера из общедоступных компьютеров на базе процессоров Intel и недорогих Ethernet-сетей. Рассмотрены методы и особенности программирования параллельных программ. Выведены аналитические соотношения основных оценок эффективности вычислений через параметры многопроцессорной системы.

Оглавление

Параллельные вычисления. Введение.

Эта книга посвящена вопросу организации параллельных вычислений, без использования суперкомпьютеров. Или, если угодно, как построить суперкомпьютер из подручных материалов.

Тема увеличения скорости вычислений весьма актуальна для всех тех, чья деятельность связана с большим объемом вычислительных работ. Так, например, рассчеты гравитирующих газовых дисков, которыми я в свое время занимался, зачастую требуют нескольких месяцев непрерывного счета на машине класса iPIII. В середине 80х, будучи студентом, выбрав для себя в качестве специализации теоретическую астрофизику, я напрямую столкнулся с проблемой быстродействия. Но, ни тогда, ни даже сейчас у российских университетов чаще всего нет средств для закупки мощных компьютеров типа nCube, Cray или подобных. Однако с развитием программного обеспечения и появлением свободно распространяемой операционной системы Linux стало возможным создать вычислительный комплекс с эффективным быстродействием, сравнимым с быстродействием суперкомпьютеров, но со стоимостью в десятки раз меньшей.

В наше время круг задач, требующих для своего решения применения мощных вычислительных ресурсов, постоянно расширяется. Это связано с тем, что произошли фундаментальные изменения в самой организации научных исследований. Вследствие широкого внедрения вычислительной техники, значительно усилилось направление численного моделирования и численного эксперимента. Численное моделирование, заполняя промежуток между физическими экспериментами и аналитическими подходами, позволило изучать явления, которые являются либо слишком сложными для исследования аналитическими методами, либо слишком дорогостоящими или опасными для экспериментального изучения. При этом численный эксперимент позволил значительно удешевить процесс научного и технологического поиска. Стало возможным моделировать в реальном времени процессы интенсивных физико-химических и ядерных реакций, глобальные атмосферные процессы, процессы экономического и промышленного развития регионов и т.д. Очевидно, что решение таких масштабных задач требует значительных вычислительных ресурсов.

Мощности современных процессоров вполне достаточно для решения элементарных шагов большинства задач, а объединение нескольких десятков таких процессоров позволяет быстро и эффективно решать многие поставленные задачи, не прибегая к помощи мэйнфреймов и

супер компьютеров.

Сейчас в наших научных организациях и университетах, как правило, имеются энтузиасты бесплатного распространяемого ПО и специалисты по ОС Linux. В то же время парк более-менее современных персональных компьютеров в этих организациях так же имеется. Закономерно появилась идея создания параллельных вычислительных систем из общедоступных компьютеров на базе процессоров Intel и недорогих Ethernet-сетей, устанавив на эти компьютеры Linux и, объединив с помощью одной из бесплатно распространяемых коммуникационных библиотек (PVM или MPI) эти компьютеры в кластер. Оказалось, что на многих классах задач и при достаточном числе узлов такие системы дают производительность, сравнимую с той, что можно получить, используя дорогие суперкомпьютеры.

> *При отсутствии высококвалифицированных параллельных программистов кластеры Beowulf (а речь на этом сайте пойдет имено о них) создаются и используются людьми с минимальным опытом параллельного программирования.*

В самом деле, кластеры Beowulf обеспечивают университеты с ограниченными ресурсами хорошей платформой для изучения параллельного программирования и недорогой производительной вычислительной системой для ученых. Затраты на установку в университетах минмиальны: многие студенты заинтересованы в таких проектах и используют Linux на собственных компьютерах, установка кластера и написание параллельных программ является частью процесса обучения.

Beowulf - это мультикомпьютерная архитектура, которая может использоваться для параллельных вычислений. Это система, обычно состоящая из одного серверного узла и одного или более клиентских узлов, соединенных при помощи Ethernet или некоторой другой сети. Это система, построенная из готовых промышленных компонент, например ПЭВМ, на которых может работать ОС Linux, стандартных адаптеров Ethernet и коммутаторов. Она не содержит специфических аппаратных компонентов и легко воспроизводима. Beowulf также использует программные продукты, такие как ОС Linux, среды программирования Parallel Virtual Machine (PVM) и Message Passing Interface (MPI). Серверный узел управляет всем кластером и является файл-сервером для клиентских узлов. Он также является консолью кластера и шлюзом во внешнюю сеть. Большие системы Beowulf могут иметь более одного серверного узла, а также возможно специализированные узлы, например, консоли или станции мониторинга. В большинстве случаев клиентские узлы в Beowulf пассивны. Они конфигурируются и управляются серверными узлами и выполняют только то, что предписано серверным узлом. В бездисковой конфигурации клиентов, клиентские узлы даже не имеют IP-адресов или имен, пока их не назначит сервер. Одно из основных отличий Beowulf от кластера рабочих станций состоит также в том, что Beowulf работает как одна машина. В большинстве случаев клиентские узлы не имеют клавиатур и мониторов, и могут быть доступны только через удаленное подключение. Узлы Beowulf могут рассматриваться как элементы процессор+память, которые вставляются в кластер так же как процессор или модуль памяти вставляются в материнскую плату.

> *Комплекс работает под операционной системой Linux. Отказ от использования каких-либо специальных hardware-компонентов делает архитектуру Beowulf легко воспроизводимой и имеющей уникальное соотношение быстродействие/стоимость.*

Beowulf - это не специфический пакет программ, новая топология сети или новейшая модификация ядра ОС. Beowulf - это технология кластеризации компьютеров, работающих под управлением ОС Linux в разновидность параллельного, виртуального суперкомпьютера.

Хотя существует много программных пакетов, таких как модификации ядра, библиотеки PVM и MPI и конфигурационные утилиты, которые делают архитектуру Beowulf более быстрой, простой в конфигурировании и эффективной, можно построить машину класса Beowulf, используя только стандартный дистрибутив Linux, без какого либо дополнительного математического обеспечения. Если имеются два соединенные сетью компьютера под ОС Linux, которые разделяют по меньшей мере файловую систему /home при помощи NFS, и позволяют друг другу выполнять удаленный шелл (rsh), то можно говорить, что это простой кластер Beowulf из двух узлов.

Кроме Beowulf, известна еще одна близкая кластерная технология - NOW. В NOW персональные компьютеры обычно содержат информацию о самих себе и поставленных перед ними задачах, а в обязанности системного администратора такого кластера входит формирование данной информации. Кластеры Beowulf в этом отношении (то есть с точки зрения системного администратора) проще: там отдельные узлы не знают о конфигурации кластера. Лишь один выделенный узел содержит информацию о конфигурации; и только он имеет связь по сети с внешним миром. Все остальные узлы кластера объединены локальной сетью, и с внешним миром их связывает только "тоненький мостик" от управляющего узла.

При обилии в Интернете информации о способах построения кластеров и технологии параллельных вычислений наблюдается недостаток систематизированных работ, в которых бы подробно описывались различные ньюансы, понятые специалистам, но являющиеся предметом затруднений для тех, кто впервые решил приобщится к миру кластерных компьютеров. Особенно это верно для материалов на русском языке. В этой книге я попытаюсь дать подробное описание тех проблем и задач, а так же их решение, с которыми придется столкнуться при попытке применить кластерные технологии для решения инженерных и научных задач.

Краткая история

История кластеров началась 1994 году. Пионером в этом деле является научно-космический центр NASA - Goddard Space Flight Center (GSFC), точнее созданный на его основе CESDIS (Center of Excellence in Space Data and Information Sciences). Специалистами GSFC летом 1994 года был собран первый кластер, состоявший из 16 компьютеров 486DX4/100MHz/16Mb RAM и трех параллельно работавших 10Mbit сетевых адаптеров. Данный кластер, который был назван "Beowulf", создавался как вычислительный ресурс проекта Eart and Space Sciences Project (ESS).

Официальная страница проекта Beowulf - http://www.beowulf.org/.

Четыре годя спустя. в 1998 году, в Лос-Аламосской национальной лаборатории (США) астрофизик Майкл Уоррен и другие ученые из группы теоретической астрофизики построили суперкомпьютер Avalon, который представлял собой Linux-кластер на базе процессоров Alpha 21164A с тактовой частотой 533 МГц. Первоначально Avalon состоял из 68 процессоров, затем был расширен до 140. В каждом узле установлено по 256 Мбайт оперативной памяти, жесткий диск на 3 Гбайт и сетевой адаптер Fast Ethernet. Общая стоимость проекта Avalon составила 313 тыс. долл., а показанная им производительность на тесте LINPACK - 47,7 GFLOPS, позволила ему занять 114 место в 12-й редакции списка Top500 рядом с 152-процессорной системой IBM RS/6000 SP. В том же 1998 году на самой престижной конференции в области высокопроизводительных вычислений Supercomputing'98 создатели Avalon представили доклад "Avalon: An Alpha/Linux Cluster Achieves 10 Gflops for $150k", получивший первую премию в номинации "наилучшее отношение цена/производительность".

В настоящее время Avalon активно используется в астрофизических, молекулярных и других научных вычислениях.

Выполненный для кластера параллельный тест Linpack показал производительность 19,7 Гфлоп/сек, результаты программы расчета молекулярной динамики (SPaSM) - 12,8 Гфлоп/сек, результаты программы решения гравитационных задач (gravitational treecode) - 10,0 Гфлоп/сек. Тесты NAS Class B версии 2.3 показали следующие результаты: BT - 2,2 Гфлоп/сек, SP - 1,0 Гфлоп/сек, LU - 3,5 Гфлоп/сек, MG - 2,1 Гфлоп/сек).

Для сравнения этих результатов с производительностью "настоящих" параллельных машин можно сказать, что SPaSM, treecode и Linpack выполнялись примерно с той же скоростью что и на 64-процессорном SGI Origin 2000 (частота процессоров - 195 МГц). Если учесть, что в мае 1998 цена на 64-процессорный Origin 2000 с 250 Мгц процессорами и 8 Гбайт оперативной памяти составляла около 1,8 миллиона долларов, а стоимость проекта Avalon - только 313 тысяч долларов, то представленные результаты весьма впечатляют.

Интересная разработка появилась недавно в Университете штата Кентукки - кластер KLAT2 (Kentucky Linux Athlon Testbed 2). Система KLAT2 состоит из 64 бездисковых узлов с процессорами AMD Athlon/700 МГц и оперативной памятью 128 Мбайт на каждом. Программное обеспечение, компиляторы и математические библиотеки (SCALAPACK, BLACS и ATLAS) были доработаны для эффективного использования технологии 3DNow! процессоров AMD, что позволило увеличить производительность.

Значительный интерес представляет и использованное сетевое решение, названное "Flat Neighbourghood Network" (FNN). В каждом узле установлено четыре сетевых адаптера Fast Ethernet от Smartlink, а узлы соединяются с помощью девяти 32-портовых коммутаторов. При этом для любых двух узлов всегда есть прямое соединение через один из коммутаторов, но нет необходимости в соединении всех узлов через единый коммутатор. Благодаря оптимизации программного обеспечения под архитектуру AMD и топологии FNN удалось добиться рекордного соотношения цена/производительность - 650 долл. за 1 GFLOPS.

Нужно ли делать кластер?

Если вы не хакер и не фанат компьютерных технологий, то построение кластерного компьютера не самоцель, а средство достигнуть большей эффективности и продуктивности вашей научной работы. Существуют определенный класс задач, требующих производительности более высокой, нежели мы можем получить, используя обычные компьютеры. В этих случаях из нескольких мощных систем создают HPC (High Perfomance Computing) кластер, позволяющий разнести вычисления не только по разным процессорам (если используются многопросерросные SMP-системы), но и по разным компьютерам. Для задач, позволяющих очень хорошее распараллеливание и не предъявляющих высоких требований по взаимодействию параллельных потоков, часто принимают решение о создании HPC кластера из большого числа маломощных однопроцессорных систем. Зачастую подобные решения, при низкой стоимости, позволяют достичь гораздо большей производительности, чем производительность суперкомпьютеров.

Однако, создание такого кластера требует определенных знаний и усиилий, а использование его влечет за собой кардинальную смену используемой парадигмы программирования, что психологически достаточно трудно. Вы можете быть крутым специалистом в написании последовательных программ, но это не спасет вас от необходимости изучения методов

параллельного программирования, возможно, начиная даже с самых азов.

Часто можно встретить заблуждение, что только использование суперкомпьютера может само по себе дать прирост производительности. Это не верно. Если ваша задача не имеет внутреннего параллелизма и не адаптирована соответствующим образом, максимум, что вы можете получить от кластера -- это запуск на выполнение нескольких экземпляров программы одновременно, работающих с различными начальными данными. Это не ускорит выполнение одной конкретной программы, но позволит сэкономить много времени, если необходимо посчитать множество вариантов за ограниченное время. Можно привести следующую аналогию: один корабль переплывает море за 7 дней, но семь кораблей не смогут переплыть море за день. За то, они смогут перевезти за неделю в 7раз больше груза. Если объемы вашей задачи таковы, что только один прогон на однопроцессорной машине может длиться сутками, неделями и месяцами, то очевидно, следует приложить усилия по адаптации алгоритма. Следует разделить задачу на несколько (по числу процессоров) более мелких подзадач, которые могут выполняться независимо, а в тех местах, где независимое выполнение невозможно, явно вызывать процедуры синхронизации, для обмена данными через сеть. Например, если вы обрабатываете большой массив данных, то разумно будет разделить его на области и распределить их по процессорам, обеспечив равномерную загрузку всего кластера.

Поэтому прежде чем перейти к практической реализации кластерной технологии необходимо решить для себя несколько принципиальных вопросов.

Первый из них звучит так: "Необходим ли для решения моих задач кластер и параллельные вычисления?" Чтобы ответить на этот вопрос надо внимательно присмотреться к решаемым вами задачам. Параллельные вычисления - достаточно специфичная область математики и далеко не всегда параллельные вычисления могут быть вами применимы. Кластер вам скорее всего не нужен, если:

- Вы используете специализированные пакеты программ, которые не адаптированы для параллельных вычесление в средах MPI и PVM или не предназначенные для работы в UNIX. В этом случае вы просто не сможете задействовать долее одного процессора для выполнения задачи или вообще запустить вашу программу в чужой операционной среде.

- Программы, написанные вами для решения ваших задач, требуют не более нескольких часов процессорного времени на имеющемся оборудовании. Может случиться так, что время, потраченное вами на распараллеливание и отладку вашей задачи съест все преимущество в быстродействии, которое даст многопроцессорная обработка.

- Время жизни вашей программы сравнимо со временем ее разработки в параллельном варианте. Основными особенностями модели параллельного программирования являются высокая эффективность программ, применение специальных приемов программирования и, как следствие, более высокая трудоемкость программирования. Не имеет смысла тратить время на распараллеливание программы, которая будет считаться несколько часов один раз в жизни, после чего, получив результаты, вы про нее забудете.

Второй вопрос, который вы должны решить, это наличие принципиальной возможности "распараллелить" вашу задачу. Некоторые численные схемы в силу особенностей алгоритма не поддаются эфективной параллелезиции. Прежде чем ориентироваться на применение кластера для для решения вашей задачи, необходимо удостовериться в возможности

применения вами паралельных алгоритмов.

Приложение в параллельной архитектуре должно создаваться с расчетом на эффективное использование ресурсов этой архитектуры. Под этим понимается то, что приложение должно быть разделено на части, способные исполняться параллельно на нескольких процессорах, и разделено эффективно, чтобы отдельно исполняемые куски программы минимально влияли на исполнение остальных частей.

Предположим, что в вашей программе доля операций, которые нужно выполнять последовательно, равна f, где 0<=f<=1 (при этом доля понимается не по статическому числу строк кода, а по числу операций в процессе выполнения). Крайние случаи в значениях f соответствуют полностью параллельным (f=0) и полностью последовательным (f=1) программам. Так вот, для того, чтобы оценить, какое ускорение S может быть получено на компьютере из p процессоров при данном значении f, можно воспользоваться законом Амдала:

$$S \le \frac{1}{f+(1-f)/p}$$

Если вдуматься как следует, то закон, на самом деле, страшный. Предположим, что в вашей программе лишь 10% последовательных операций, т.е. f=0.1 . Что утверждает закон? А он говорит о том, что сколько бы вы процессоров не использовали, ускорения работы программы более, чем в десять раз никак не получите... да и то 10 - это теоретическая верхняя оценка самого лучшего случая, когда никаких других отрицательных факторов нет... :((

Отсюда первый вывод - прежде, чем основательно перепахивать код для перехода на параллельный компьютер надо основательно подумать, а стоит ли овчинка выделки?. Если оценив заложенный в программе алгоритм вы поняли, что доля последовательных операций велика, то на значительное ускорение рассчитывать явно не приходится и нужно думать о замене отдельных компонент алгоритма.

В дополнение к вышесказанному настоятельно рекомендуется прочесть и осознать вот эти замечания.

Трудовые затраты на распараллеливание или векторизацию программы

Самый простой вариант попробовать ускорить имеющуюся программу - это воспользоваться встроенными в транслятор (обычно с ФОРТРАНа или Си) средствами векторизации или распараллеливания. При этом никаких изменений в программу вносить не придется. Однако вероятность существенного ускорения (в разы или десятки раз) невелика. Трансляторы с ФОРТРАНа и Си векторизуют и распараллеливают программы очень аккуратно и при любых сомнениях в независимости обрабатываемых данных оптимизация не проводится. Поэтому, кстати, и не приходится ожидать ошибок от компиляторов, если программист явно не указывает компилятору выполнить векторную или параллельную оптимизацию какой-либо части программы.

Второй этап работы с такой программой - анализ затрачиваемого времени разными частями программы и определение наиболее ресурсопотребляющих частей. Последующие усилия должны быть направлены именно на оптимизацию этих частей. В программах наиболее затратными являются циклы и усилия компилятора направлены прежде всего на векторизацию и распараллеливание циклов. Диагностика компилятора поможет установить причины, мешающие векторизовать и распараллелить циклы. Возможно, что простыми действиями удастся устранить эти причины. Это может быть простое исправление стиля программы, перестановка местами операторов (цикла и условных), разделение одного цикла на несколько, удаление из критических частей программы лишних операторов (типа операторов отладочной печати). Небольшие усилия могут дать здесь весьма существенный

выигрыш в быстродействии.

Третий этап - замена алгоритма вычислений в наиболее критичных частях программы. Способы написания оптимальных (с точки зрения быстродействия) программ существенно отличаются в двух парадигмах программирования - в последовательной и в параллельной (векторной). Поэтому программа, оптимальная для скалярного процессора, с большой вероятностью не может быть векторизована или распараллелена. В то же время специальным образом написанная программа для векторных или параллельных ЭВМ будет исполняться на скалярных машинах довольно медленно. Замена алгоритма в наиболее критических частях программы может привести к серьезному ускорению программы при относительно небольших потраченных усилиях. Дополнительные возможности предоставляют специальные векторные и параллельные библиотеки подпрограмм. Используя библиотечные функции, которые оптимизированы для конкретной ЭВМ, можно упростить себе задачу по написанию и отладке программы. Единственный недостаток данного подхода состоит в том, что программа может стать не переносимой на другие машины (даже того же класса), если на них не окажется аналогичной библиотеки.

Написание программы "с нуля" одинаково сложно (или одинаково просто) для машин любых типов. Этот способ является идеальным для разработки эффективных, высокопроизводительных векторных или параллельных программ. Начинать надо с изучения специфики программирования для векторных и параллельных ЭВМ, изучения алгоритмов, которые наиболее эффективно реализуются на ЭВМ данных типов. После этого надо проанализировать поставленную задачу и определить возможность применения векторизуемых и распараллеливаемых алгоритмов для решения конкретной задачи. Возможно, что придется переформулировать какие-то части задачи, чтобы они решались с применением векторных или параллельных алгоритмов. Программа, специально написанная для векторных или параллельных ЭВМ, даст наибольшее ускорение при ее векторизации и распараллеливании.

Классы задач

В этой главе мы обсудим возможные классы задач, которые можно эффективно решать на кластерных компьютерах. Мы коснемся некоторых математических моделей, встречающихся во многих научных и инженерных задачах.

Одномерные массивы

Данные задачи встречаются довольно часто. Если значения элементов массива определяются довольно сложным выражением, а вычислять их надо многократно, то распараллеливание цикла для вычисления элементов массива может оказаться очень эффективным. В отдельный параграф мы вынесли решение систем дифференциальных уравнений, что по своей сути также является обработкой массивов функций, производных и т.д. Но на самом деле эффективными могут также быть вычисления сверток, сумм, функций от каждого элемента массива и т.п. Конечно, не имеет смысл распараллеливать действия над короткими массивами кроме тех случаев, когда собственно вычисления каждого элемента занимают большое время.

Двумерные массивы

При исполнении вложенных циклов обычно эффективно распараллеливаются самые внешние циклы. Однако практически все действия с матрицами (сложение, умножение, умножение на вектор, прямое произведение) могут быть выполнены на кластере. Многие алгоритмы линейной алгебры (но не все) могут быть эффективно распараллелены. Некоторые библиотеки подпрограмм (например, LAPACK) существуют для параллельных машин. Совершенно неэффективно использовать кластеры для работы с матрицами низкой размерности (например 3x3). Но можно переписать алгоритм для одновременной обработки

нескольких (к примеру 1000) матриц - обращение, поиск собственных чисел и т.д. При увеличении размера матриц растет эффективность работы программы, но растет и размер требуемой памяти для хранения матриц.

Клеточные автоматы

Во многих областях знания встречаются задачи, которые сводятся к вычислению эволюции объектов, расположенных в дискретных точках и взаимодействующих с ближайшими соседями. Простейшей и, наверно, наиболее широко известной такой задачей является игра "Жизнь". Можно так же привести в качестве примера модель магнетиков Изинга, представляющую собой набор спинов (элементарных магнитов), расположенных в узлах решетки и взаимодействующих только с ближайшими соседями. Алгоритм построения эволюции Изинговских магнетиков будет во многом идентичен алгоритму игры "Жизнь".

Системы дифференциальных уравнений

Решение систем дифференциальных уравнений встречается во многох инженерных и научных задачах. В большинстве случаев алгоритмы решения подобных задач можно эффективно распараллелить для обработки на кластерном компьютере. В качестве примеров можно упомянуть такие задачи, как молекулярные модели сплошных сред в статистической физике, инженерные расчеты по распределению нагрузок в сложных конструкциях, модели N тел (например расчеты движения космических аппаратов, динамика звездного диска Галактики), газодинамика сплошных сред (особенно, если исследуется многокомпонентная среда), электродинамика и др.

Видно, что класс задач, решать которые можно используя параллельные алгоритмы довольно широк и крайне важен. Однако следует учитывать, что параллельность задачи определяется не только ее физическим смыслом, но и выбранным численным алгоритмом. Например, всем известный метод прогонки практически не поддается распараллеливанию. Если единственный или предпочтительный метод решения вашей задачи - метод прогонки, то вам придется отказаться от применения кластерных компьютеров. С другой стороны, метод Монте-Карло идеально подходит для кластерного компьютера. Причем, чем больше процессоров будет в кластере, тем эфективнее будет решаться задача. Практически все варианты явных разностных схем решения дифференциальных уравнений успешно распараллеливаются.

С чего начать?

Итак, вы решили, что параллельный кластер - это то, что вам не хватало для вашей работы. Возникает вопрос, из чего его делать и сколько компьютеров необходимо связать в кластер, чтобы затраченные усилия дали ощутимый результат. Кроме того хотелось бы понять какие компьютеры необходимы для кластера.

Сразу скажу, что кластер Beowulf - гетерогенная структура. В него могут входить самые разнообазные по параметрам компьютеры, построенные на различных аппаратных платформах, например Intel Pentium различных версий, Alpha, RISC-процессоры, Transmeta, 32-х и 64-х битовые процессоры. Более того, на компьютерах в кластере могут быть установлены самые различные системы: Linux, Windows, OS/2 WARP. Нашей целью будет построение кластера с минимальными усилиями. Поэтому, если вы хотите заниматься делом (сиречь научной работой), а не повышать свой профессионализм в области информационных технологий, о возможной гетерогенности кластера я предлагаю забыть. Будем считать, что аппаратная платформа комьютеров нашего будущего кластера однообразна.

Что касается различия в параметрах (быстродействие, память, ...) у компьютеров, входящих в кластер, то это допустимо. Но в этом случае, вам придется учитывать эти различия при написании параллельных программ, распределяя объем счета в зависимости от возможности

каждого отдельного компьютера. В противном случае кластер будет работать как система, состоящая из машин с минимальными рабочими параметрами.

Как я уже говорил, построение кластера - не самоцель, а средство. Поэтому для минимизации усилий будем считать, что все компьютеры кластера одинаковы по своим рабочим характеристикам и управляются одной и той же операционной системой. За одним исключением, главный компьютер кластера, консоль кластера, может (но не должен) быть более мощной машиной.

Начнем с самого простого, с выбора размера кластера. Поскольку кластер Beowulf - масштабируемая система, то вопрос количества узлов не является жизненно важным. По мере роста ваших аппетитов вы можете произвольно добавлять количество узлов в любое время. Если же вы для узлов будете использовать удаленную загрузку операционной системы по сети (о чем мы еще поговорим позже), то работы по добавлению узла кластера не выйдут за рамки технического подключения новой машины в сеть. Естественно вам придется еще немного изменить ваши программы, разбив их на большее количество параллельных подзадач, с тем чтобы иметь возможность использовать в процессе счета большее количество процессоров.

> *Однако увлекаться количеством узлов не стоит. Самое узкое место в вашем кластере - это среда передачи данных между узлами, то есть пропускная способность используемой сети.*

Как видно из рисунка, эффективность вашего кластера зависит от числа узлов нелинейно. Собственно вид этой функции зависит от вида задачи, которая решается с помощью кластера. Приведенный рисунок более-менее корректно отображает положение дел с задачами типа решения уравнений газодинамики на больших сетках (но не только). В этом случае эффективность кластера растет до того момента, когда время передачи между узлами информации, необходимой для проведения одной итерации становится сравнимым со временем счета одной итерации.

В других случаях, например для решения системы уравнений методом Монте-Карло или методом перебора, функция эффективности принимает линейный вид. То есть, чем больше машин в кластере, тем быстрее работает программа. Если же говорить о методе прогонки, то функция эффективности имеет максимум при количестве узлов равном единице и спадает до нуля обратно пропорционально росту количества узлов.

Таким образом, можно порекомендовать при начальном построении кластера ограничится четырьмя узлами (одна консоль и три slave-ноды). С одной стороны, вы всегда при необходимости можете нарастить кластер, с другой стороны, меньшее количество узлов может дать не столь ощутимый результат, как ожидалось. Тем не менее, при пробемах с финансированием можно ограничится и двумя узлами. А если вы просто хотите попробовать, что такое есть кластер, можете обойтись вообще одним компьютером, с установленным на нем VirtualBox.

Что такое кластер?

Кластер Beowulf состоит из отдельных машин (узлов) и объединяющей их сети (коммутатора). Кроме ОС, необходимо установить и настроить сетевые драйверы, компиляторы, ПО поддержки параллельного программирования и распределения

вычислительной нагрузки.

Узлы кластера. Подходящим выбором в данный момент являются системы на базе процессоров Intel Core 2 Duo или Intel Core 2 Quad. Стоит установить на каждый узел не менее 1Gb оперативной памяти. Желательно 2-4Gb. Одну из машин следует выделить в качестве центральной (консоль кластера) куда можно (но не обязательно) установить достаточно большой жесткий диск, возможно более мощный процессор и больше памяти, чем на остальные (рабочие) узлы. Делать консоль кластера более мощной машиной имеет смысл, если вы захотите иметь на этом компьютере кроме интерфейса командной строки более удобное операционной окружение, например оконный менеджер (KDE, Gnome), офисные программы, программы визуализации данных и т.п..

Имеет смысл обеспечить (защищенную) связь этой машины с внешним миром. Другими словами, сеть кластера (сеть состоящая их консоли кластера и рабочих узлов) топологически не должна находится внутри корпоративной сети. Если необходимо обеспечить доступ к консоли кластера из корпоративной сети и/или Интернет, то в этом случае, связь должна идти через отдельную сетевую карту, установленную в главном компьютере, и отдельный коммутатор.

При комплектации рабочих узлов вполне возможно отказаться от жестких дисков - эти узлы будут загружать ОС через сеть с центральной машины, что, кроме экономии средств, позволяет сконфигурировать ОС и все необходимое ПО только один раз (на центральной машине). Если эти узлы не будут одновременно использоваться в качестве пользовательских рабочих мест, нет необходимости устанавливать на них видеокарты и мониторы. Возможна установка узлов в стойки (rackmounting), что позволит уменьшить место, занимаемое узлами, но будет стоить несколько дороже.

> ***Важное замечание.*** *Поскольку библиотеки для параллельных вычислений MPICH/MPI являются кроссплатформенными, то выбор операционной системы (Windows vs Linux) не важен. Однако следует учесть тот факт, что Linux является заметно менее ресурсоемкой системой. Например при использовании PelicanHPC GNU Linux система занимает в оперативной памяти не более 40Мб! Вся остальная память доступна параллельной программе. Это очень важный фактор в том случае, когда кластер используется с целью моделирования процессов на как можно более подробной сетке.*

Возможна организация кластеров на базе уже существующих сетей рабочих станций, то есть рабочие станции пользователей могут использоваться в качестве узлов кластера ночью и в нерабочие дни. Системы такого типа называют COW (Cluster of Workstations). В этом случае реальным представляется вариант, когда кластер строится на основе существующего компьютерного класса. Подобные классы уже имеются в большинстве учебных или научных учреждениях и обычно скомплектованы однотипными машинами, что и необходимо для кластера. Однако обычно такие компьютерные классы работают под операционной системой Windows и, вероятно, для замены ее на Unix придется решить вопросы административного плана и вопросы связанные с построением учебного процесса. Принципиальных препятствий для решения этих вопросов по-видимому нет, поскольку Unix (конкретно Linux) имеет все необходимое программное обеспечение для проведения учебного процесса или научной деятельности (компиляторы, средства разработки, офисные программы, программы работы с изображениями и визуализации данных, средства публикации (TeX)). Эта книга, например, большую часть времени писалась на консоли кластера в OpenOffice под управлением операционной системы Linux. Нельзя сказать, чтобы я испытывал при этом какую-либо ностальгию по старому доброму MS Word.

> *По большому счету отказываться от Windows не обязательно. Коммуникационные библиотеки PVM, MPI имеются не только для UNIX, но и для*

Windows. Если установка в компьютерном классе UNIX-сети вызывает непреодолимую аллергическую реакцию у админов или преподавателей, можно оставить ту операционную систему, к которой вы привыкли.

В принципе, для кластерных систем типа COW нет насущной необходимости останавливать кластер (и задачи на нем считаемые) на дневное (рабочее) время, когда за узловыми машинами работают пользователи. Работа параллельных программ конечно будет замедляться, но это не летально. Другое дело, если работа кластера будет заметно тормозить и затруднять работу пользователей.

Сеть. В простейшем случае для связи между узлами кластера используется один сегмент Ethernet (10Mbit/sec на витой паре). Однако дешевизна такой сети, вследствие коллизий оборачивается большими накладными расходами на межпроцессорные обмены, а хорошую производительность такого кластера можно ожидать только на задачах с очень простой параллельной структурой и при очень редких взаимодействиях между процессами (например, перебор вариантов).

Для получения хорошей производительности межпроцессорных обменов используют полнодуплексный Fast Ethernet на 100Mbit/sec или Gigabit Ethernet. При этом для уменьшения числа коллизий или устанавливают несколько "параллельных" сегментов Ethernet, или соединяют узлы кластера через коммутатор (switch). Под "параллельными" сегментами подразумевается такая структура сети, когда каждый узел кластера имеет более одной сетевой карты, которые с помощью специальных драйверов объединяются в один виртуальный сетевой интерфейс, имеющий суммарную пропускную способность. Для того, чтобы избежать проблем с конфигурированием такого виртуального интерфеса, следует использовать одинаковые сетевые карты на всех машинах кластера. Кроме того, каждая параллельная линия такого интерфеса должна представлять из себя Ethernet-сеть построенную на отдельном (от других параллельных ей линий) комутаторе.

Как построить кластер?

Хорошая новость заключается в том, что развертывание кластера как такового - задача экстремально простая. Причем, для этого подойдет **любой** дистрибутив по вашему выбору. Какой именно из дистрибутивов Linux ставить в качестве базовой ОС - не имеет значения. Ubuntu, Mandriva, Alt Linux, Red Hat, SuSE... Выбор зависит только от ваших предпочтений.

Попытаюсь по шагам объяснить, как развернуть кластер, используя дистрибутив общего назначения.

Итак... Вы должны будете:

1. Установить операционную систему на компьютер, который будет выступать в роли **консоли кластера**. То есть на этом компьютере будут компилироваться и запускаться параллельные программы. Другими словами, за этим компьютером будет сидеть человек, запускать программы и смотреть, что получилось.
2. После инсталлирования базовой ОС на консоли кластера, если это не сделано в процессе первоначальной установки, Вы должны будете установить необходимые компиляторы (фортран, C) и все необходимые библиотеки, desktop environment (GNOME или KDE по вашему выбору), текстовые редакторы и пр., то есть превратить этот компьютер в рабочую станцию разработчика.
3. Установить из репозитория или из исходников пакет MPICH или OpenMPI, если он Вам более по душе. Я лично предпочитаю OpenMPI.
4. Описать в /etc/hosts будущие узлы вашего кластера, в том числе и консоль

кластера.

5. Установить NFS и расшарить для **всех** узлов кластера некую директорию, в которой будут размещаться исполняемые модули параллельных программ и файлы данных, которыми эти программы будут пользоваться в процессе своей работы.
6. Установить на консоли кластера ssh-клиент (обязательно) и ssh-сервер (опционально, если Вы предполагаете давать доступ к консоли кластера по сети).
7. На всех узлах кластера установить операционную систему, библиотеки, необходимые для выполнения пользовательских параллельных программ, установить MPICH, NFS-client, ssh-server. Узлы кластера в целях экономии ресурсов должны грузиться в runlevel 3, так что ставить туда GNOME или KDE не надо. Максимум - поставить туда гномовские или кдешные библиотеки, если они нужны пользовательским программам.
8. Описать в /etc/hosts всех узлов кластера будущие узлы вашего кластера, в том числе и консоль кластера.
9. На всех узлах кластера необходимо автоматом при загрузке монтировать расшаренный в п. 5 ресурс. Причем, путь к этому ресурсу должен быть одинаков, как на консоли кластера, так и на его узлах. Например, если на консоли кластера Вы расшариваете каталог /home/mpiuser/data, то на узлах кластера этот ресурс также должен быть смонтирован в /home/mpiuser/data.
10. На всех узлах кластера обеспечить безпарольный доступ по ssh для консоли кластера. Как это сделать вы можете посмотреть на моем сайте.
11. Собственно, все. Кластер собран и полностью готов к использованию. Фактически для развертывания кластера нам потребовалось установить ОС, зайти в так сказать в "Установку и удаление программ", отметить для установки пакеты SSH и MPICH, запретить запрос пароля удаленного доступа к узлам кластера, расшарить на центральном узле каталог, где будут храниться наши параллельные программы и данные и поставить на узлах кластера автоматическое подключение к этому каталогу при загрузке. Как компилировать и запускать на исполнение параллельные программы Вы можете посмотреть в других разделах этого сайта и в документации к MPICH.

Как видите, все очень просто и ничего, кроме дистрибутива с репозиториями не нужно. Более подробно вопрос установки кластера на базе операционной системы Linux описан в разделе Ubuntu кластер.

Теперь я хотел бы обсудить другой вопрос, а именно построение сети кластера. Поскольку сеть - самое узкое место и от нее впрямую зависит эффективность работы кластера, то хотелось бы сделать следующее. Хотелось бы, чтобы функционирование сетевой файловой системы NFS не мешало обмену данными, который осуществляют между собой части параллельной программы, работающие на разных узлах. Чтобы это осуществить, необходимо физически разделить сеть на два сегмента. В одном сегменте будет работать NFS, в другом - будет происходить обмен данными между частями программы.

Таким образом и в консоли кластера и в его узлах необходимо иметь два сетевых интерфейса (две сетевые карты), Соответственно, **нужно два набора свитчей, не связанных друг с другом**, и два набора сетевых реквизитов для этих интерфейсов. То есть, NFS работает, например, в сети 192.168.1.0/24, а обмен данными происходит в сети 192.168.2.0/24. И соответственно, в файлах /etc/exports и /etc/fstab должны будут быть прописаны адреса из первой сети, а в файлах /etc/hosts и в файла machines.LINUX, описывающих кластер - адреса из второй. Что за файл machines.LINUX - смотрите в документации MPICH.

> ***Важное замечание.*** *Файлы, хранящиеся на диске, в условиях параллельной задачи, выполняемой на кластере, могут понадобиться только для сохранения состояния задачи в контрольных точках. Конечно, дисковые ресурсы можно использовать и для организации виртуальной памяти, подгружая по мере необходимости данные в оперативную память, увеличивая тем самым размер разностной сетки. Однако при наличии кластера, увеличение размера разностной сетки логичнее и эффективнее может быть выполнено посредством использования дополнительных вычислительных узлов кластера. Если же дисковые ресурсы используются только для сохранения контрольных точек и эти контрольные точки расположены не в каждой итерации (а в каждой десятой или сотой), то разделение локальной сети кластера на два независимых сегмента (NFS и сеть межпроцессорного обмена данных) является не обязательной. Вполне можно обойтись всего одним сегментом, используя его и для NFS и для обмена данными. Поскольку NFS будет использоваться достаточно редко, то и отрицательное влияние ее на эффективность кластера будет минимально.*

Еще одно. Настоятельно рекомендуется использовать гигабитную сеть, как наиболее доступную для университета (с точки зрения финансов). Строго говоря, Gigabit Ethernet - не лучший выбор в качестве сети кластера в силу того, что эта сеть обладает достаточно большой латентностью. Но это доступное решение. Если же финансы позволят, то лучше конечно обратить внимание на Myrinet или 10Gbit Ethernet.

Далее. Параметры сети никогда не бывают слишком хорошие. Поэтому, если есть возможность, надо стараться их улучшить. Если Myrinet или 10GbE для вас будут недоступны, то можно попытаться улучшить характеристики гигабитной сети. Собственно, сделать из нее двухгигабитку! Погуглите по ключевым словам **channel bonding**, кроме того, у меня на сайте немного про это написано. Суть дела в том, что вместо одной сетевой карты мы используем две, объединив их специальным драйвером в единый виртуальный канал с двойной пропускной способностью. В этом случае карты должны быть подключены к двум отдельным свитчам, то есть потоки по этим картам мы разделяем так же, как мы это делали раньше, разделяя NFS и передачу данных. Создание такого канала - немного геморройное занятие, поэтому только от вашего желания и энтузиазма зависит, делать это или нет. В принципе, это не обязательно, хотя эффект будет заметный.

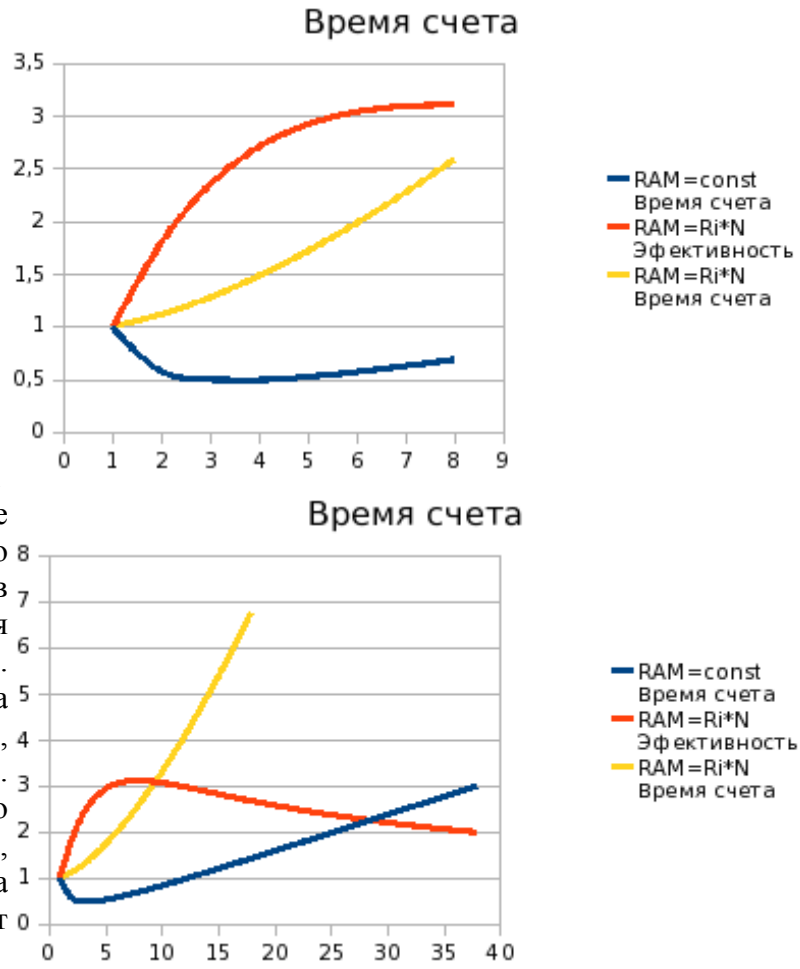
Теперь собственно о том, а **зачем вообще Вам нужен кластер**. Дело в том, что утверждение «чем больше узлов в кластере, тем быстрее он работает» - в общем случае не верно. Давайте посмотрим, в каких случаях нам захочется считать наши программы на кластере. Существует только две цели использования кластера.

1. Имеется разностная сетка размера R, вычисления на которой при использовании обычного компьютера занимают время T. Время T - критический параметр. Нам хочется существенно уменьшить время вычислений, имея R как константу.
2. Имеется разностная сетка размера R, вычисления на которой при использовании обычного компьютера занимают время T. Время T - не критично. Нас интересует увеличение размера сетки сверх имеющейся в одном компьютере памяти для более детального счета, возможного получения более тонких эффектов и т.п.

Все вычисления на разностной сетке имеют один общий и важный для нас параметр: время одной итерации. В случае использования кластера это время состоит из двух частей: время счета на сетке T_{iter} и время обмена данными между узлами T_{exch}. (Почитайте про **граничный обмен** на моем сайте.) T_{iter} зависит только от мощности процессора. А вот T_{exch} зависит уже, от размера разностной сетки, количества узлов кластера и пропускной

способности сети. Я не буду приводить формул, вы их сами можете при желании вывести. (Посмотрите еще вот этот файл. Именно по нему были построены графики.) Приведу окончательный результат для случая двухгигабитной сети, размера разностной сетки 64 гигабита и времени одной итерации 100 сек.

На графике ось ординат - временная характеристика, ось абсцисс - количество узлов кластера.



Обратите внимание на **синий график**. Это модель **первого случая**, когда разбиваем разностную сетку постоянного размера на несколько узлов. Как видно из графика, время счета вначале уменьшается, при увеличении количества узлов кластера. Что мы и хотели получить. Однако уменьшение происходит до определенного предела. При количестве узлов более четырех общее время счета снова начинает расти. Происходит это из-за увеличения объема данных, пересылаемых между узлами. Таким образом получается, что при постоянном размере сетки, увеличивать размер кластера свыше четырех узлов не имеет смысла.

Теперь рассмотрим **случай 2**, когда нам важен размер сетки, а со временем счета мы можем смириться.

Давайте представим, что у нас есть один компьютер с неограниченной памятью. Увеличивая размер разностной сетки, мы получаем линейное (с коэффициентом 1) увеличение время счета. Теперь сравним это время с тем, которое получится, если мы будем считать такую же сетку, но на кластере. Причем увеличивая размер сетки вдвое, мы увеличиваем вдвое и количество узлов кластера. Поскольку две чати сетки обсчитываются параллельно, то время итерации не увеличивается, но появляется время обмена данными. **Красный график** показывает отношение времени счета на одном компьютере (с неограниченной памятью) ко времени счета такой же сетки на кластере. **Желтый график** показывает рост времени счета при увеличении узлов кластера (и, соответственно, увеличении размера сетки). И рост этот, что важно, меньше, чем линейный.

Мы видим, что время счета на кластере существенно меньше, чем если бы мы считали сетку на одном компьютере. Причем, даже при увеличении размера сетки (и узлов кластера) в 40 раз, мы, тем не менее, получаем выигрыш во времени.

Для кластерных вычислительных систем одним из широко применяемых способов построения коммуникационной среды является использование концентраторов (hub) или коммутаторов (switch) для объединения процессорных узлов кластера в единую вычислительную сеть. В этих случаях топология сети кластера представляет собой полный

граф, в котором, однако, имеются определенные ограничения на одновременность выполнения коммуникационных операций. Так, при использовании концентраторов передача данных в каждый текущий момент времени может выполняться только между двумя процессорными узлами; коммутаторы могут обеспечивать взаимодействие нескольких непересекающихся пар процессоров.

Следует отметить, что приведенные выше расчеты сделаны в предположении, что для построения сети кластера используются концентраторы. В случае же использования коммутаторов **скоростные характеристики кластера могут заметно улучшиться**. Поэтому именно их рекомендуется использовать при проектировании кластера.

Резюме. Если мы имеем целью ускорить выполнение программы, то можно использовать кластер, но размером не больше двух-четырех узлов. Если же мы имеем целью получение доступа к памяти, большей, чем может нам обеспечить один компьютер, то кластер - именно то, что нам нужно.

Важное замечание. На приведенные на графиках зависимости очень сильно влияет такой параметр, как время счета одной итерации. То есть время между двумя моментами, когда возникает необходимость в граничном обмене. Чем оно меньше, тем хуже характеристика кластера. Если у вас достаточно простая задача, в которой время итерации порядка десятка секунд, то необходимость в кластеризации такой задачи весьма сомнительна. В любом случае, желательно, чтобы время обмена граничными данными между узлами кластера было много меньше времени итерации.

Ну и еще пару слов о целесообразности использования кластера. Дело в том, что не всякую задачу можно решать на кластере. Во-первых, как я уже говорил, если время итерации (время счета между моментами возникновения необходимости граничного обмена) достаточно мало и сравнимо с временем передачи данных, то эффективность кластера резко снижается. Во-вторых, не всякую задачу можно распараллелить. Например метод прогонки решения дифф. ур. вообще никак не распараллеливается и может быть решен только на одном процессоре.

Как быстро построить кластер?

Вполне представима ситуация, когда по каким-либо причинам развернуть стационарный кластер не представляется возможным. Ну, например, когда компьютерный класс, который вы собираетесь использовать в качестве вычислительного кластера, по каким-то причинам обязательно должен работать под операционной системой Windows. Ничего страшного! С помощью специализированного дистрибутива **PelicanHPC GNU Linux** вы в любой момент, например после окончания рабочего дня и занятий в компьютерном классе, можете запустить кластер и на приведение его в боевую готовность потребуется не более пяти минут. Причем исходная операционная система, программное обеспечение и данные на используемых в качестве кластера компьютерах не будет модифицировано. После выключения кластера компьютеры придут в то состояние, которое было до начала работы кластера.

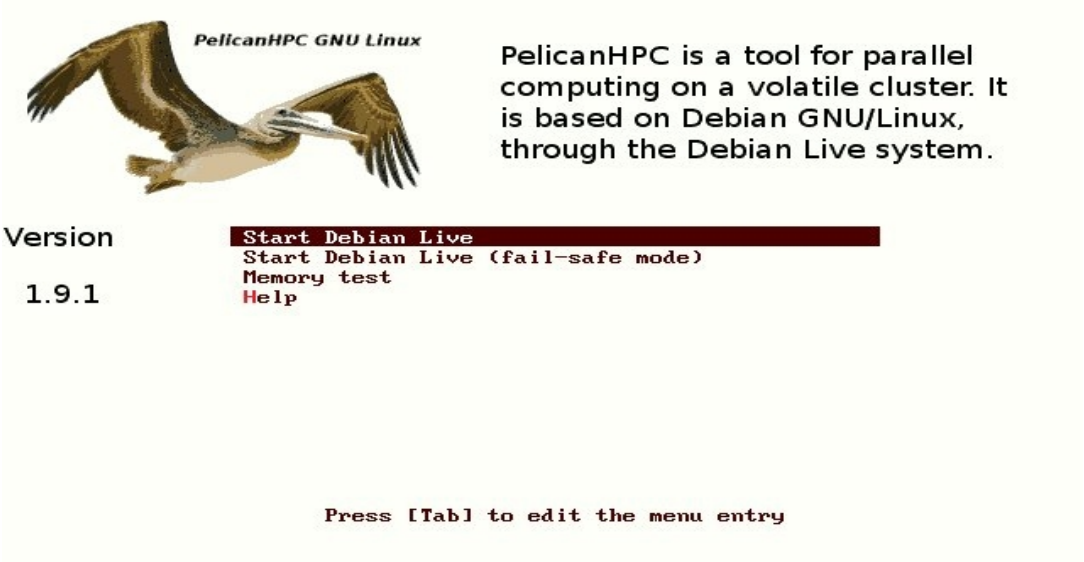
Для развертывания такого виртуального кластера вам потребуется один компакт-диск с последней версией дистрибутива PelicanHPC GNU Linux, iso-образ которого вы можете взять **отсюда**. С этого диска вы загружаете операционную систему кластера (не устанавливая ее на винчестер) на компьютере, который будет играть роль консоли кластера, то есть того компьютера, непосредсвенно за которым вы будете работать, компилируя и запуская ваши параллельные программы.

Остальные узлы кластера будут загружаться по сети. Для загрузки ОС вычислительных узлов кластера по сети необходимо, чтобы сетевые карты этих компьютеров умели выполнять

загрузку по сети. Большинство современных карт, в том числе встроенных, это делать умеют. Если же вам не повезло, то вы всегда можете сделать загрузочный CD из образа **gpxe.iso** и загрузить ваши вычислительные узлы с этого CD. Если же вам совсем не повезло и на предполагаемых вычислительных узлах отсутствуют и возможность загрузки по сети и CD-приводы, то и в этом случае отчаиваться не стоит. Посетите ресурс **www.rom-o-matic.net**, сгенерируйте и запишите на дискету загрузочный floppy-образ, соответствующий вашим сетевым картам. С этой дискеты и выполните загрузку ОС на остальных узлах кластера.

Теперь посмотрим, как на практике выполняется загрузка кластера.

1. Загружаем консоль кластера с PelicanHPC GNU Linux Live CD

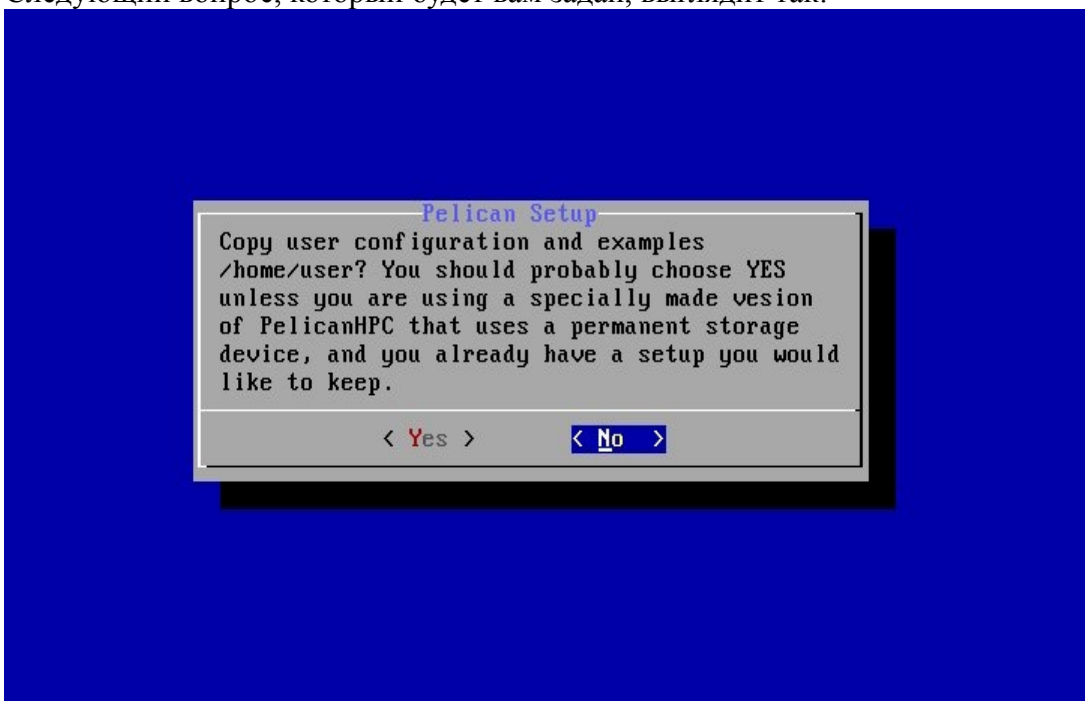


2. Через некоторое время повяляется следующий запрос:

Здесь вы должны будете указать устройство, на котором будет располагаться пользовательский каталог. То есть ваш рабочий каталог, где будут храниться ваши программы, исходники и файлы данных. По умолчанию вам предложен раздел на виртуальном диске, расположенном в оперативной памяти. Это самый простой вариант, однако не самый удобный в том смысле, что после выключения компьютера все данные на этом диске будут уничтожены. В этом случае вам придется каждый раз перед выключением сохранять ваши данные на внешний носитель, например на флешку. Более удобным будет, если вы выделите на винчестере компьютера отдельный

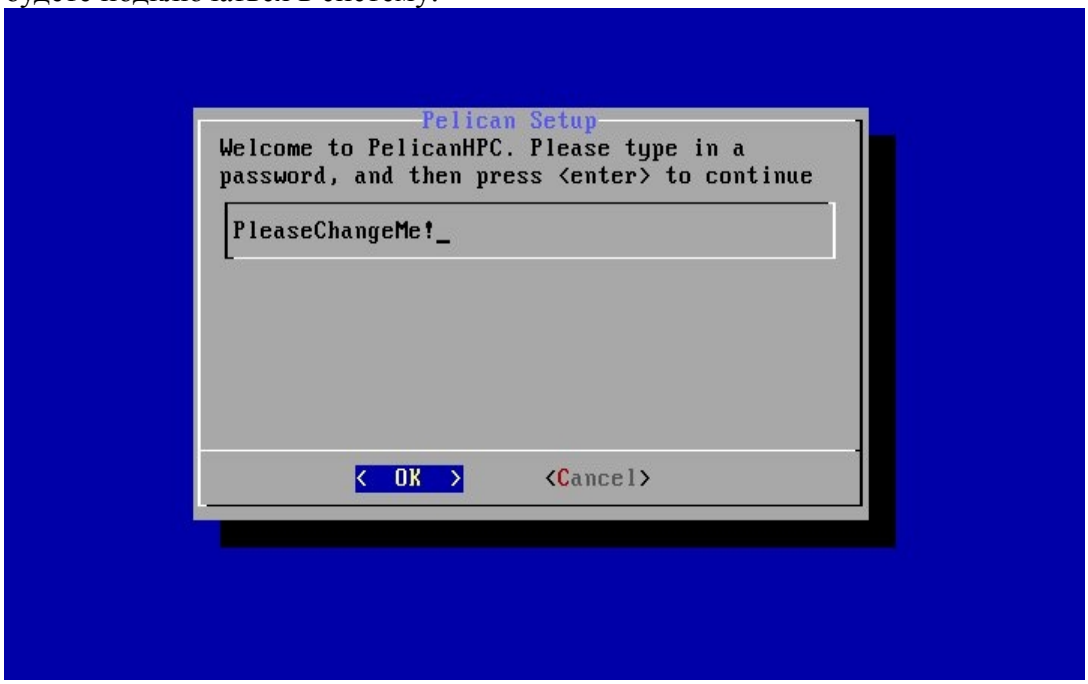
раздел для ваших данных. Как вариант может рассматриваться подключение внешнего носителя (флешки либо USB-винчестера). В таком случае вместо предложенного ram1 вы должны будуте указать что-то типа hda7, sda5, sdb1 и т.д. в зависимости от конфигурации вашей машины и выбранного варианта.

3. Следующий вопрос, который будет вам задан, выглядит так:

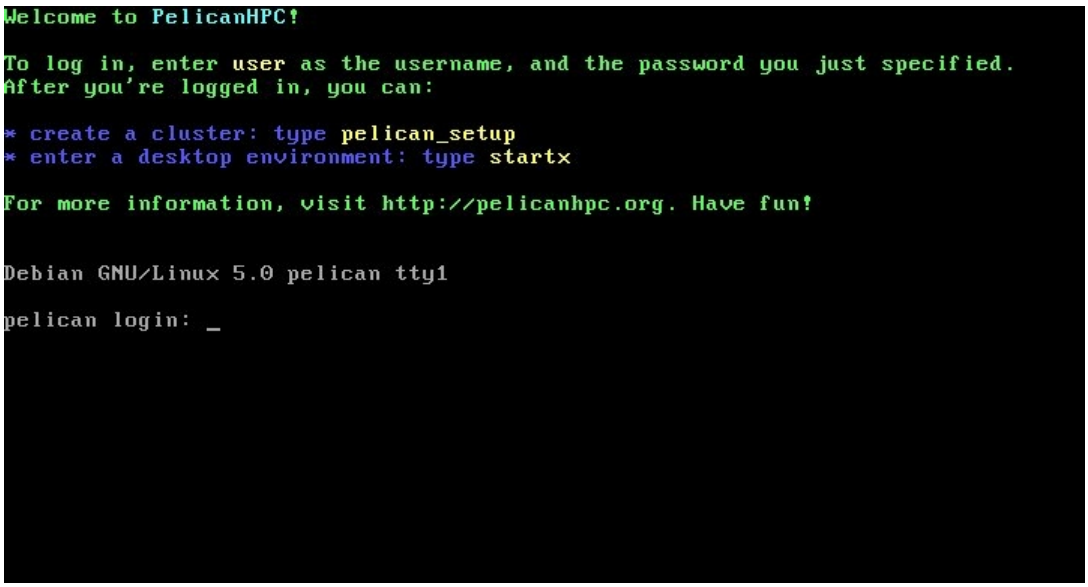


Система спрашивает, будет ли выполнена начальная конфигурация пользовательского каталога. В случае, когда вы используете в качестве пользовательского каталога виртуальный диск ram1, всегда отвечайте "Yes". Если же вы выбрали в качестве месторасположения пользовательского каталога постянный носитель (раздел винчестера компьютера, флешку или внешний USB-винчестер), тогда ответ "Yes" вы должны будете выбрать только в самый первый раз. Во все последующие загрузки кластера необходимо выбрать ответ "No".

4. На следующем шаге вы должны будете указать пароль пользователя, с которым вы будете подключаться в систему:

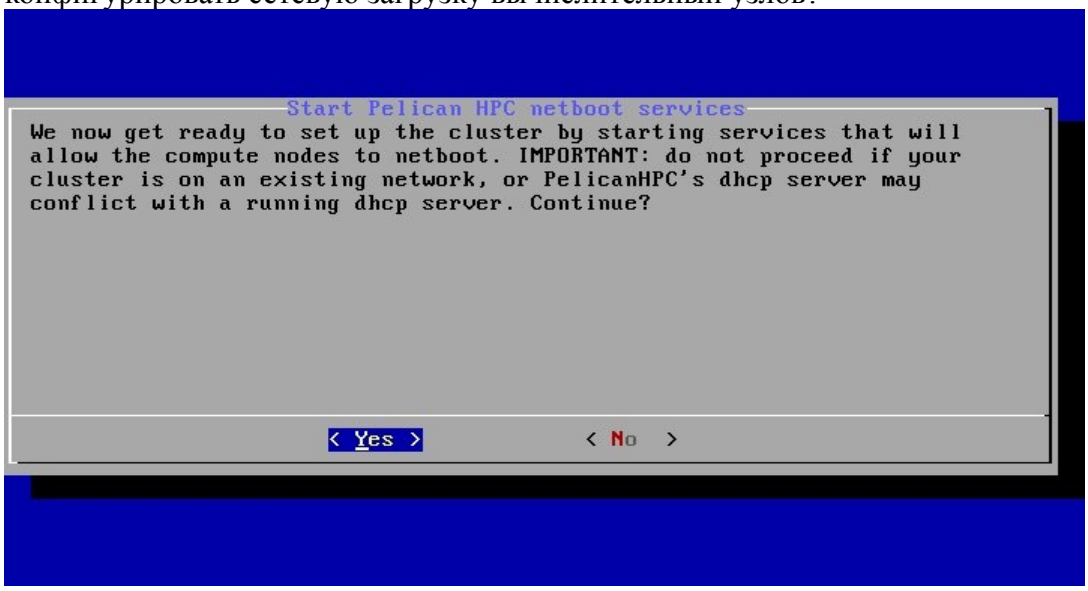


5. После задания пароля вам будет предложен стандартный экран входа в систему:



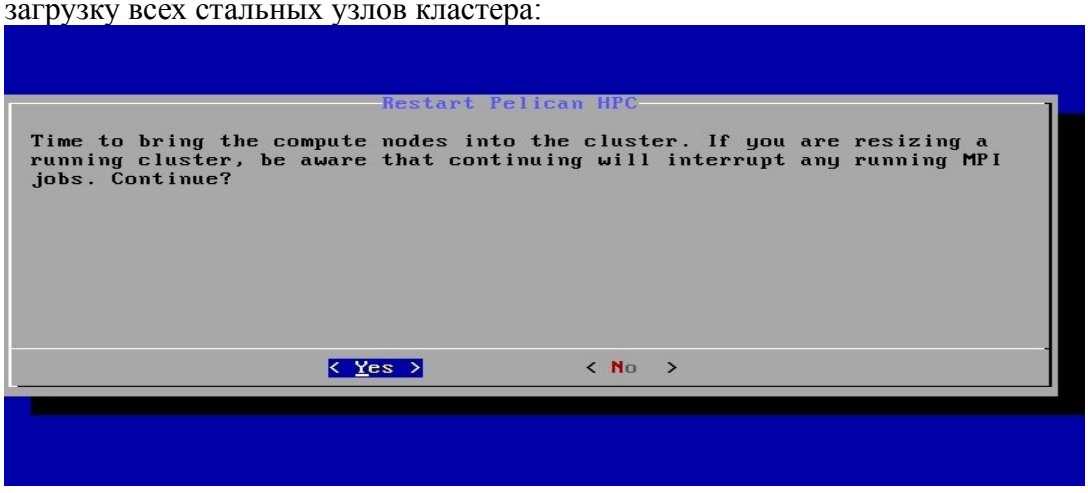
Для входа в систему вы можете использовать логин "user" и пароль, который вы определили на предыдущем шаге.

6. Итак, мы вошли в систему консоли кластера. Теперь нам необходимо подключить к класетру все наши вычислительные узлы. Для этого запустим команду конфигурации кластера **pelican_setup**. Первое, что спросит эта команда - будем ли мы конфигурировать сетевую загрузку вычислительный узлов?

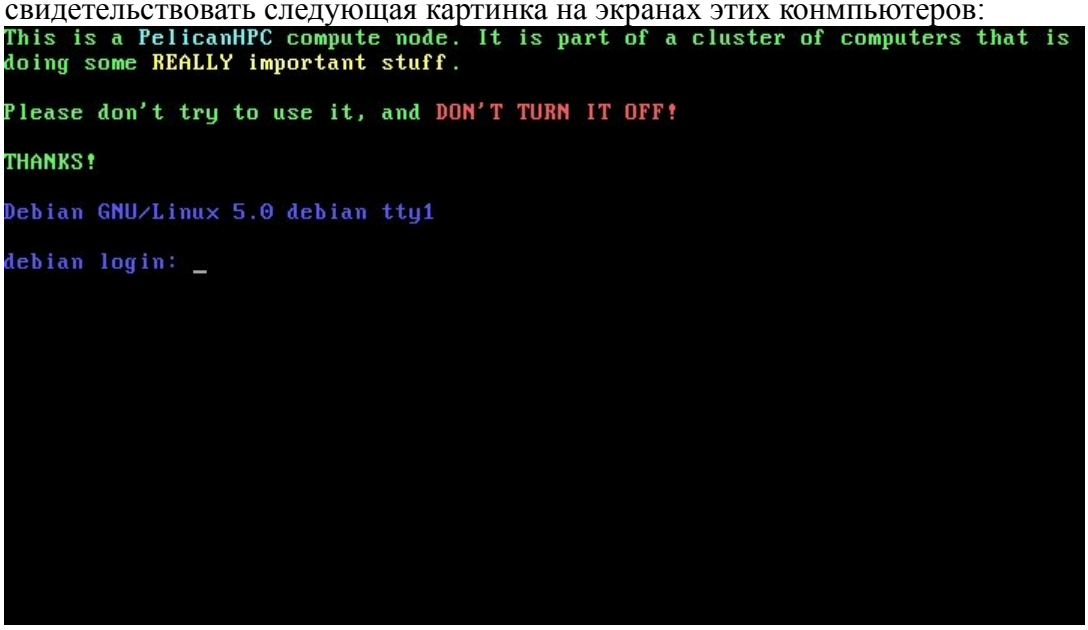


Естественно, мы говорим "Yes".

7. Сконфигурировав сервер сетевой загрузки, программа предложит нам выполнить загрузку всех стальных узлов кластера:

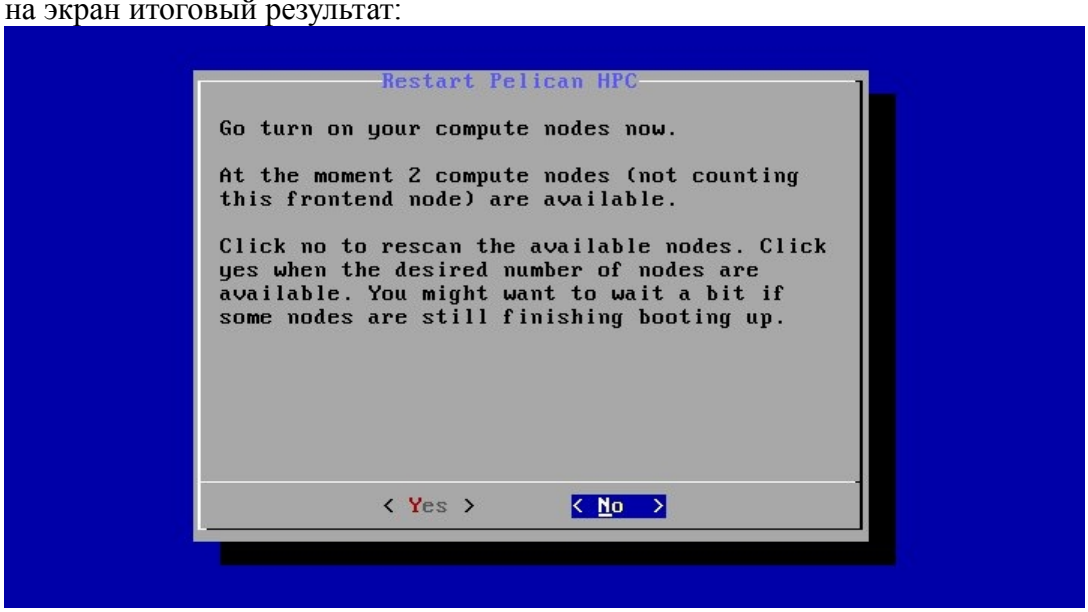


В этот момент мы должны включить все остальные компьютеры кластера, не забыв поправить настроики BIOS таким образом, чтобы они выполнили загрузку по сети. Вмешательства в процесс загрузки вычислительных узлов кластера не требуется. Надо просто дождаться, когда они все закончат процедуру загрузки, о чем будет свидетельствовать следующая картинка на экранах этих конмпьютеров:



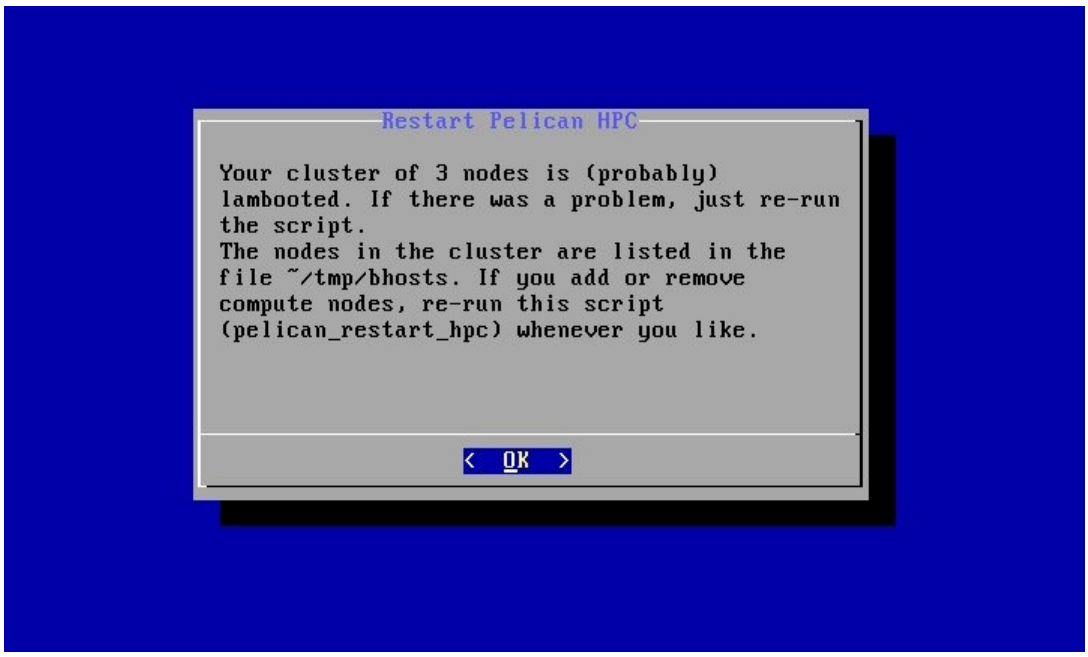
После того, как все компьютеры будут загружены, нажимаем кнопку "Yes".

8. Далее программа настройки попытается найти все загруженные компьютеры и включить их в конфигурацию кластера. После выполнения этого действия она выдаст на экран итоговый результат:



На этом экране программа сообщает нам, сколько было найдено вычислительных узлов (в данном случае два) кроме узла, который является консолью кластера. Если все нормально - нажимаем "Yes".

9. И, наконец, программа конфигурации кластера сообщает нам, что все настройки выполнены и кластер готов к эксплуатации:



Нам остается только завершить ее, нажав "ОК".

10. В некоторых случаях у меня конфигурация кластера завершалась с ошибкой, поэтому я рекомендую следующим шагом запустить скрипт реконфигурации: **pelican_restart_hpc**, тем самым повторив заново пункты 7-9.

Теперь кластер работоспособен. проверим его работу на тестовой программе. Возьмем в качестве пример программу вычисления числа π flops.f. Каким-либо способом копируем исходник этой программы в пользовательский каталог на консоли кластера. Далее выполняем следующие действия:

- Компилируем программу в параллельной среде MPI с помощью команды **mpif77 flops.f -o flops**
- Запускаем программу на одном процессоре командой **./flops**
- Запускаем программу на двух процессорах командой **mpirun n2,0 ./flops**

Как видно из результатов работы программы

```
user@pelican:~$ ./flops

  HPC Test ----------------------------------------
  Quantity of processors =   1
  Calculation time       =   1.66 seconds
  Cluster speed          =   1084 MFLOPS
  ------------------------------------------------
  Cluster node N00 speed =   1084 MFLOPS
  ------------------------------------------------

user@pelican:~$ mpirun n2,0 ./flops

  HPC Test ----------------------------------------
  Quantity of processors =   2
  Calculation time       =   0.89 seconds
  Cluster speed          =   2033 MFLOPS
  ------------------------------------------------
  Cluster node N00 speed =   1016 MFLOPS
  Cluster node N01 speed =   1016 MFLOPS
  ------------------------------------------------

user@pelican:~$ _
```

скорость вычисления на двух процессорах примерно в два раза больше, чем на одном. То есть кластер делает именно то, что мы и ожидали.

Важное замечание. По каким-то причинам конфигурация кластера оформляется таким образом, что в списке вычислительных узлов консоль кластера прописывается последней. Однако логично ожидать, что вывод программы будет идти на монитор той машины, с которой она запущена, то есть на консоль. В то же время параллельные программы обычно пишут таким образом, что весь вывод идет в процессе, который работает на самом первом узле. Поэтому при запуске программы приходится явно указывать последовательность узлов, на которых она будет выполнятся. И первым в этом списке должна быть именно консоль кластера. В нашем примере кластер состоял из трех машин. Нумерация их начинается с нуля. То есть 0, 1, 2. Поэтому мы явно указали, что первая машина - это машина номер 2 (последняя в конфигурационном списке). Указали мы это параметром **n2,0**, то есть программа запускалась на машине N2 и машине N0.

Если бы мы сделали кластер например из 16 машин, то их номера были бы 0, 1, 2, ... 14, 15. Для запуска нашей тестовой программы на таком кластере надо было бы использовать команду

mpirun n15,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14 ./flops
или в сокращенном варианте
mpirun n15,0-14 ./flops

Тест производительности

Тест Whetstone. Стандартная программа оценки и показатель производительности компьютера, построенная из смеси машинных команд и типов данных, типичных для научных приложений. Разработана подразделением фирмы ICL, расположенном в английском городке Whetstone.

Тест спроектирован для оценки характеристик систем при интенсивном использовании операций с плавающей запятой. Это смесь плавающих и целочисленных операций, вычисления трансцедентных функций, обработки массивов и вызовов процедур. Комплект тестов Whetstone состоит из нескольких модулей, имитирующих программную нагрузку в наиболее типичных режимах исполнения вычислительных задач.

Каждый модуль теста выполняется многократно, в соответствии с исходной статистикой Whetstone-инструкций (практически это реализуется с помощью заключения модулей в циклические конструкции с разным числом "оборотов" цикла - от 12 до 899), а производительность рассчитывается как отношение числа Whetstone-инструкций к суммарному времени выполнения всех модулей пакета. Этот результат представляется в KWIPS (Kilo Whetstone Instructions Per Second) или в MWIPS (Mega Whetstone Instructions Per Second). В известном смысле указанные единицы аналогичны MIPS, но с одной существенной оговоркой: Whetstone-инструкции не привязаны к системе команд какого-либо компьютера, т. е. оценка производительности в MWIPS является моделенезависимой.

Для оценки производительности нашего кластера я предлагаю использовать несколько модифицированный вариант теста, из которого убраны процедуры ввода/вывода, не относящиеся непосредственно к вычислительному процессу, и добавлен блок, учитывающий необходимость граничного обмена средствами MPI. О граничном обмене мы поговорим позже.

Тест написан на языке Fortran. Программа состоит из двух модулей, помещенных в отдельные исходные файлы: **whetstoned.f** - собственно сам тест Whetstone, оформленный в виде подпрограммы, и **mwips.f** - основная программа. Компиляция теста выполняется следующей командой:

```
mpif77 mwips.f whetstoned.f -o mwips
```

Программа в среднем работает от 30 до 60 секунд, в зависимости от используемых компьютеров. Тест производится над данными двойной точности. В граничном обмене участвуют 6 массивов действительных чисел двойной точности, размерностью 8800 элементов каждый. Такая модель граничного обмена примерно соответствует решению системы дифференциальных уравнений МГД на двумерной области на машинах с доступной памятью около 3.5 Гб. Результатом работы программы будет примерно следующее:

```
Cluster Double Precision Whetstone Test ---------
Quantity of processors =   3
Calculation time       =  44.55 seconds
Cluster speed          =  40404 MWIPS
-------------------------------------------------
Cluster node N00 speed =  28477 MWIPS (supergate)
Cluster node N01 speed =  17094 MWIPS (node01)
Cluster node N02 speed =  15719 MWIPS (node02)
-------------------------------------------------
Nodes minimal speed    =  15719 MWIPS
Acceleration factor    =   2.57
-------------------------------------------------
```

В проведенном тесте граничный обмен отрицательно влияет на скоростные характеристика кластера. Граничный обмен длится примерно 16% от времени, затраченного программой на вычислительные процедуры, что заметно замедляет весь процесс. В отсутствие граничного обмена, без которого в реальной задаче обойтись к сожалению нельзя, характеристики нашего тестового кластера выглядят так:

```
Cluster Double Precision Whetstone Test ---------
Quantity of processors =   3
Calculation time       =  38.55 seconds
Cluster speed          =  46693 MWIPS
-------------------------------------------------
Cluster node N00 speed =  28585 MWIPS (supergate)
Cluster node N01 speed =  17104 MWIPS (node01)
Cluster node N02 speed =  15756 MWIPS (node02)
-------------------------------------------------
Nodes minimal speed    =  15756 MWIPS
Acceleration factor    =   2.96
-------------------------------------------------
```

Для компиляции тестовой программы использовался компилятор **Intel Frotran ver. 11.1 20090630**.

Для сравнения, при использовании компилятора **gfortran ver. 4.3.3** из коллекции GCC этот же тест на том же оборудовании дает следующий результат:

```
Cluster Double Precision Whetstone Test ---------
Quantity of processors =   3
Calculation time       = 726.15 seconds
Cluster speed          =   2479 MWIPS
-------------------------------------------------
Cluster node N00 speed =   1504 MWIPS (supergate)
Cluster node N01 speed =   1010 MWIPS (node01)
Cluster node N02 speed =    833 MWIPS (node02)
-------------------------------------------------
Nodes minimal speed    =    833 MWIPS
Acceleration factor    =   2.98
-------------------------------------------------
```

Организация сети вычислительного кластера

Сеть - это модульная и адаптируемая коммутационная система, которую можно настроить в соответствии с самыми различными требованиями. Ее модульность облегчает добавление новых компонентов или перемещение существующих, а адаптивность упрощает внесение изменений и усовершенствований. Сеть кластера Beowulf ничем принципиально не отличается от сети рабочих станций, поэтому в самом простом случае для построения кластера необходимы обычные сетевые карты и хабы/коммутаторы, какие использовались бы при обустройстве какого-нибудь компьютерного класса. Однако, в случае кластера имеется одна особенность. Сеть кластера в первую очередь предназначена не для связи машин, а для связи вычислительных процессов. Поэтому чем выше будет пропускная способность вашей сети, тем быстрее будут считаться параллельные задачи, запущенные на кластере, следовательно рабочие характеристики сети приобретают первостепенное значение**.**

Для построения вычислительных кластеров используют самое разнообразное сетевое оборудование. При этом, так как характеристики стандартных сетевых устройств заметно уступают характеристикам специализированных коммуникаций в "нормальных" MPP компьютерах, пропускная способность сети, связывающей узлы кластера, во многих случаях оказывается решающей для производительности кластера. Используемое сетевое оборудование характеризуют обычно двумя параметрами:

Пропускная способность.

Это скорость передачи данных между двумя узлами после того, как связь установлена. Производитель обычно заявляет пиковую пропускную способность, которая в 1.5-2 раза выше реально наблюдаемой в приложениях.

Латентность.

Это среднее время между вызовом функции передачи данных и самой передачей. Время затрачивается на адресацию информации, срабатывание промежуточных сетевых устройств, прочие накладные расходы, возникающие при передаче данных.

Приведем для сравнения параметры некоторых наиболее популярных сетевых устройств.

Сетевое оборудование	Пиковая пропускная способность	Латентность
1. FastEthernet	12.5 Mbyte/sec	150 sec
2. GigabitEthernet	125 Mbyte/sec	150 sec
3. Myrinet	160 Mbyte/sec	5 sec
4. SCI	400 Mbyte/sec (реально 100)	2.3 sec
5. cLAN	150 Mbyte/sec	30 sec

Фактически пропускная способность и латентность не только характеризуют кластер, но и ограничивают класс задач, которые могут эффективно решаться на нем. Так, если задача требует частой передачи данных, кластер, использующий сетевое оборудование с большой латентностью (например GigabitEthernet), будет большую часть времени тратить даже не на передачу данных между процессами, а на установление связи, в то время как узлы будут простаивать, и мы не получим значительного увеличения производительности. Впрочем, если пересылаются большие объемы данных, влияние периода латентности на эффективность кластера может снижаться за счет того, что сама передача потребует достаточно большого времени, может быть даже в разы больше периода латентности.

Для малобюджетных кластеров использование супербыстрых Myrinet, SCI, cLAN скорее всего может оказаться нереальным с финансовой точки зрения. Поэтому рассмотрим более дешевые решения. Использование для кластера 10Mbit-сети хотя и возможно, но малоприятно. В результате вы рискуете получить от использования кластера больше разочарований, чем реального увеличения эффективности вашей работы. Далее мы будем рассматривать оборудование для сетей от 100Mbit и выше.

Сетевые карты. В качестве сетевых адаптеров можно использовать любые имеющиеся в продаже карты, поддерживающие работу в стандартах 100BaseTx и GigabitEthernet. Что касается списка предпочтений, то можно порекомендовать в первую очередь 3Com. Среди других вариантов можно назвать Compex, Intel, Macronix, другие карты, поддерживаемые драйвером tulip, например карты на чипсетах DC21xxx. Особенно популярными при построении кластеров являются платы на базе микросхем Intel 21142/21143. Популярность этих карт вызвана бытующим мнением об их высокой производительности, в то время как их цена по сравнению с конкурирующими предложениями обычно довольно невелика. Что касается сетевых карт фирмы 3Com, то они имеют некоторые преимущества, заметно влияющие на производительность сетевых коммуникаций. Приведем лишь несколько примеров возможностей аппаратного обеспечения карт 3Com.

Разгрузка процессора при вычислении контрольных сумм TCP/UDP/IP. Освобождает центральный процессор от интенсивных вычислений контрольных сумм, выполняя их в самой сетевой плате. Тем самым повышается производительность системы и время жизни процессора.

Освобождение ЦП при восстановлении сегментированных пакетов TCP. Снижает нагрузку на центральный процессор, повышая производительность системы.

Объединение прерываний. Позволяет группировать несколько полученных пакетов. Оптимизирует вычислительную эффективность хост-компьютера, сокращая число прерываний и максимально освобождая процессорные ресурсы для работы приложений.

Режим Bus mastering DMA. Обеспечивает более эффективный обмен данными для снижения загрузки центрального процессора.

В любом случае, если вы не предполагаете использовать технологию связывания каналов (channel bonding), которая позволяет объединять несколько сетевых адаптеров в один скоростной виртуальный канал, то вы можете себя чувствовать достаточно свободно выбирая для покупки ту или иную карту. Практически все современные сетевые карты, имеющиеся сейчас в продаже, без проблем распознаются Linux'ом и нормально работают.

Для организации связанного канала (channel bonding) лучше всего выбрать сетевые карты Intel EtherExpress PRO/100, 3Com FastEthernet (например 3c905B, 3c905C) или какие-либо карты GigabitEthernet от 3Com или Intel. Так же интересным вариантом являются специализированные серверные сетевые карты, в которых имеется более одного Ethernet-порта. Примерами таких адаптеров могут быть Intel EtherExpress PRO/1000 MF Dual Port или 3Com Fast EtherLink Server Dual Port 3c982C-TXM , которые я без труда нашел на http://www.price.ru. Использование таких карт позволит занимать в компьютере вдвое меньше PCI-слотов и, соответственно, устанавливать вдвое больше сетевых карт для объединения их в связанный канал.

Коммутаторы. Вторым важным элементом сети кластера являются устройства коммутации сетевых каналов. При выборе коммутирующих устройств так же следует учитывать возможность использования channel bonding. В зависимости от того, будет ли использоваться технология связывания каналов при построении кластера, можно остановить свой выбор на различном сетевом оборудовании.

Коммутаторы и другие элементы сетевой структуры используются для обеспечения коммуникаций между процессорами, для поддержки параллельного программирования и различных функций управления. Для параллельного программирования (организации межпроцессного взаимодействия (Inter Process Communication, IPC) широко используется коммутатор Myrinet-2000 компании Myricom (http://www.myri.com) - очень быстрое, хорошо масштабируемое широкополосное устройство. Считается, что при увеличении числа подключенных узлов общая ширина полосы пропускания - как у всех коммутаторов с настоящей масштабируемостью - растет пропорционально, а латентность остается

постоянной. Иными словами, полоса на каждом из путей одинакова, а число путей (направлений) зависит от количества узлов, при этом каждый узел имеет связь со всеми остальными узлами независимо от размера кластера. Например, полоса в расчете на направление может составлять 200 Мбайт/с в каждом направлении с латентностью в 6-8 мкс. Коммуникации между пользовательскими пространствами могут реализованы на основе протоколов IP или GM при помощи ПО пользовательского уровня Myricom.

Если среда параллельных вычислений не требует повышенной интенсивности коммуникаций между процессорами, то могут использоваться менее дорогостоящие средства, скажем, Ethernet. В индивидуальном заказном проекте могут также применяться технологии GigaNet, SCI или ServerNet, а в будущем и InfiniBand.

Выбор коммутатора осуществляется прежде всего на основе его характеристик. В самом простом случае для построения сети каластера можно использовать простые хабы. Это решение, наиболее выгодное по цене, явялется самым неудачным в технологическом смысле. При использовании хабов не происходит маршрутизации пакетов передаваемых данных. Любой пакет, переданный в сеть, направляется абсолютно всем участникам сети. Каждая машина "слышит" все передающиеся в сети пакеты данных, вне зависимости от того, предназначен ли конкретный пакет для нее или нет. При активном межпроцессорном обмене это может приводить к перегрузке сети, увеличении числа коллизий и, как следствие, к снижению эффективного быстродействия параллельной машины. Например, если две пары узлов кластера одновременно обмениваются данными посредством 100Мбит хаба, то скорость их обмена падает вдвое. Для решения этой проблемы следует использовать более "продвинутое" сетевое оборудование - коммутаторы, которые позволяют устанавливать своего рода каналы связи между парами машин.

Если говорить, к примеру, о 100Мбит сети, то задачей комутатора является обеспечение пропускной способности 100 Мбит/с одновременно для всех n/2 соединений между парами портов n-портового коммутатора. Теоретически коммутатор должен это гарантировать, но на практике производители оборудования весьма часто идут на упрощение электронной начинки своей продукции, как с целью удешевления, так и с целью максимального увеличения числа портов. В последнем случае при распараллеливании могут возникать конфликты на уровне Fast Ethernet, что снижает скорость обмена сообщениями и соответственно эффективность распараллеливания.

По моему личному опыту таблица приоритетов при выборе сетевого коммуникатора для построения сети кластера может выглядеть так: Cisco Catalist, 3Com SuperStack 3, Compex Switch. Ну и на последнем месте стоят самые дешевые хабы различных производителей, таких как Compex или 3Com.

Конечно, принимая решение о выборе коммутатора, необходимо учесть и другие их характеристики, в том числе цену. Хорошая продукция и стоит дороже. Так, отличные коммутаторы Cisco Catalyst (например, известная модель 5000, имеющая большее число портов и поддерживающая возможность связывания каналов) имеют более высокую цену, чем оборудование не столь "именитых" фирм.

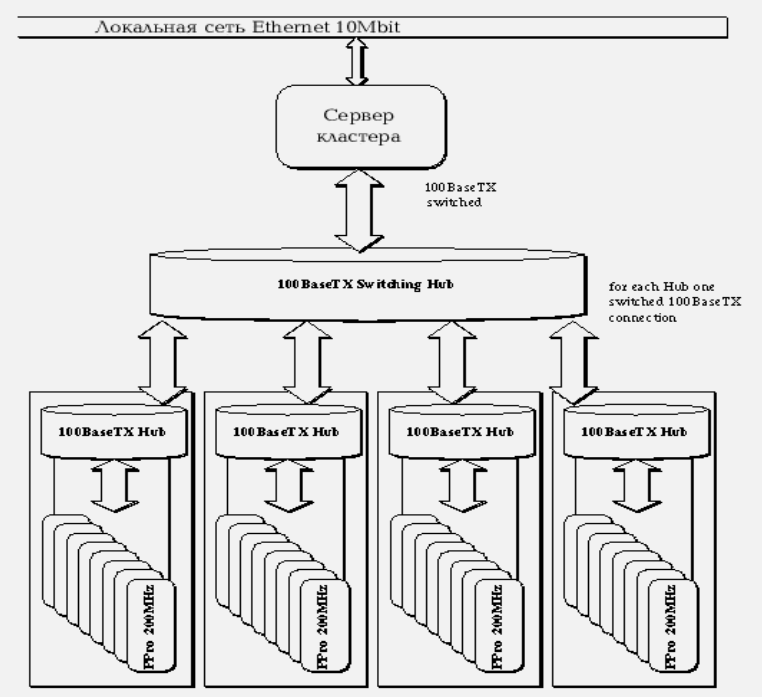
Не все коммутаторы могут обеспечить возможность применения связанных каналов. Если вы предполагаете использовать channel bonding для увеличения пропускной способности вашей сети, то необходимо с особой тщательностью подходить к выбору коммутатроа. Обычные хабы в этом случае отпадают сразу. Проблема в связывании каналов заключается в том, что пр наличии channel bonding у вас появляется две или несколько сетевых карт с одинаоквым MAC-адресом. В обычном режиме работы коммутатор либо просто "сойдет с ума", либо будет интенсивно перестраивать свои внутренние таблицы портов, переназначая ваш MAC-адрес с одного порта на другой. Это может привести либо к полной неработоспособности канала, либо к значительным потерям пакетов и существенному снижению производительности сети. Для обеспечения нормальной работы таких связанных каналов в

коммутаторе должны быть предусмотрены функции Link Aggrigation или, по другому, работа в стандарте IEEE 802.3ad. При покупке коммутатора внимательно читайте прилагаемые спецификаци и ищите эти магические словосочетания. Не все коммутаторы, имеющие функцию Link Aggrigation, позволяют применять ее для всех портов. Например, существуют модели, которые имеют 12/24 100Мбит и два гигабитных порта. В таких моделях Link Aggrigation можно настроить только для гигабитных портов, используя их для связи между двумя коммутаторами. Ясно, что такие модели не применимы для наших целей. Поэтому консультации со специалистами при покупке коммутатора обязательны.

В качестве примеров коммутатором, позволяющих настроить Link Aggrigation, можно упомянуть Cisco Catalist 2900 series, Cisco Catalist 3500 series, Cisco Catalist 5000 series, 3Com SuperStack 3 4950, 4400 и др. Следует отметить, что наличие или отсутствие функций Link Aggrigation зависит не только от модели коммутатора, но и от версии его программного обеспечения.

Сетевое обеспечение кластера

Как уже говорилось, узлы кластера можно связать обычным способом, используя Ethernet-адаптеры. Соединение машин кластера может выглядеть так, как это показано на рисунке.



Однако скорость доступа по сети никогда не бывает слишком большой. Поэтому, для получения от кластера максимальной эффективности необходимо по возможности увеличивать пропускную способность сети. Для этого можно в машинах кластера использовать не одну, а несколько сетевых карт, объединив их в один логический канал с большей пропускной способностью.

Интерфейс пользовательского уровня для такого "слияния" каналов состоит из двух программ: 'ifconfig' и 'ifenslave'. Первый сетевой интерфейс конфигурируется как обычно командой 'ifconfig'. Программа 'ifenslave' копирует установки первого интерфейса на все остальные дополнительные интерфейсы. Этой же командой можно при желании какие-либо интерфейсы сконфигурировать в режиме Rx-only.

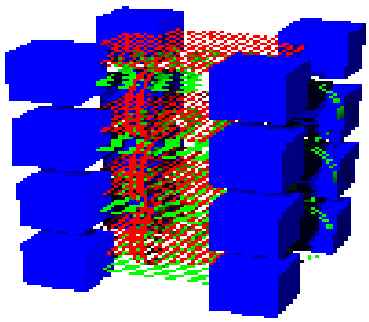
Этот метод называется "Beowulf Ethernet Channel Bonding". Для программиста и программ, выполняемых на кластере, метод абсолютно прозрачен. Единственное влияние, которое он оказывает - увеличение быстродействия.

Применение метода имеет некоторые ограничения: все присоединенные машины должны иметь одинаковый набор bonded networks, т.е. нельзя в одной машине использовать 2x100BaseTx, а в другой 10Base и 100BaseTx. Применение метода состоит из двух частей, необходимы изменения кернела для поддержки channel bonding, и программа ifenslave.

Channel bonding

Рассмотрим более подробно каким образом из нескольких сетевых интерфейсов можно

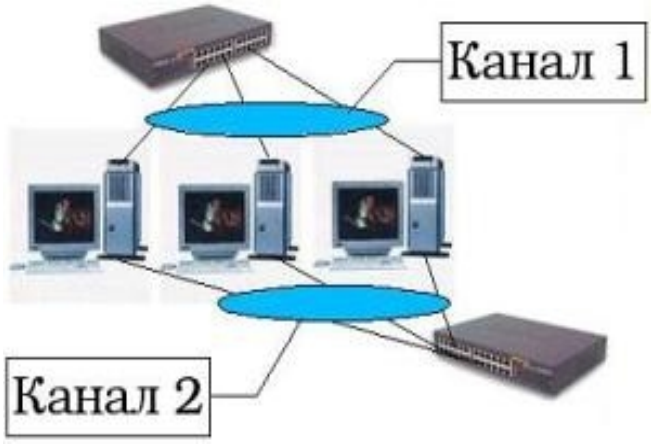
создать один виртуальный скоростной канал.



Для увеличения эффективной пропускной способности сети кластера рекомендуется использовать так называемое "связывание каналов" или channel bonding. Это такой способ объединения узлов кластера в сеть, когда каждый узел подсоединяется к коммутатору более чем одним каналом. Чтобы достичь этого, узлы надо оснастить либо несколькими сетевыми платами, либо многопортовыми платами Fast Ethernet. Связать можно и гигабитные каналы. Связывание каналов аналогично режиму транкинга при соединении коммутаторов, который используется для увеличения скорости передачи данных между двумя или несколькими коммутаторами. Применение связывания каналов в узлах под управлением ОС Linux позволяет организовать равномерное распределение нагрузки приема/передачи между соответствующими каналами.

Channel bonding пораждает некоторые проблемы связанные с выбором коммутаторов и их настройки. Коммутатор должен уметь работать со связанными каналами иначе могут происходить всевозможные ошибки при построении комутатором таблиц маршрутизации пакетов или таблиц MAC-адресов. То есть, как уже было упомянуто ранее, в качестве сетового оборудования надо выбирать такой ethernet switch, который поддерживает для своих портов функции Link Aggregation или IEEE 802.3ad. Другим решением проблемы я вляется выбор коммутатора с возможностью поддержки режима виртуальных локальных сетей (VLAN). Применение VLAN призвано помочь избежать "дублирования" во внутренних таблицах коммутаторов MAC-адресов многопортовых сетевых плат. Впрочем, есть сообщения, что и поддержка VLAN не всегда помогает, вы можете попробовать этот вариант, но на свой страх и риск.

> *Вместо использования специализированного сетевого оборудования, поддерживающего связывание каналов, можно разделить каналы с помощью двойного (тройного и т.д.) набора обычных хабов или свитчей на непересекающиеся сетевые сегменты таким образом, чтобы каждый канал образовывал свою собственную сеть, физически не связанную с сетями других каналов.*



Организация в системе сетевого итерфеса по методу channel bonding достаточно проста. Нужно только следовать одному правилу. Все присоединенные машины должны иметь одинаковый набор bonded networks, т.е. нельзя в одной машине использовать 2x100BaseTx, а в другой 10Base и 100BaseTx. Режим работы сетевых карт тоже должен быть однообразный. Другими словами, недопустим вариант, когда одна карта работает в full duplex, а другая в полудуплексном режиме. В каждой же отдельной машине можно устанавливать карты различных производителей, но работающие обязательно в одном стандарте. Channel bonding

требует наличия как минимум двух физических подсетей. Но, при желании связанный канал можно построить на основе трех или более сетевых карт.

Для связывания сетевых карт в один канал (одну виртуальную карту) необходимо либо скомпилировать ядро системы с поддержкой cannel bonding, либо загрузить в систему модуль ядра bonding.o.

> *В Linux начиная с ядер 2.4.x channel bonding является стандартной включаемой опцией. Например в дистрибутиве Alt Linux Master 2.2 channel bonding поставляется в виде загружаемого модуля ядра.*

Для конфигурации связанного канала вам потребуется стандартная команда ifconfig и, возможно, дополнительная команда ifenslave. Программа 'ifenslave' копирует установки первого интерфейса на все остальные дополнительные интерфейсы. Этой же командой можно при желании какие-либо интерфейсы сконфигурировать в режиме Rx-only.

Покажем процесс настройки channel bonding на примере использования двух сетевых карт. Сетевой интерфейс для первой карты должен быть заранее сконфигурен и полностью работоспособен. Для добавления в систему второй карты и объединения ее с первой в связанный канал требуется выполнить некоторые достаточно простые действия. Предварительно желательно остановть сетевые интерфейсы вашей системы выполнив команду

/etc/rc.d/init.d/network stop

После этого переходим собственно к конфигурации связанного канала. Для начала вам нужно изменить файл /etc/modules.conf, добавив в него следующую строчку.

alias bond0 bonding

Сделанное нами добавление говорит системе о том, что при загрузке необходимо загрузить модуль bonding.o, который узнается так же по алиасу bond0. Чтобы не перезагружать систему, вручную загрузим модуль:

modprobe bonding

Теперь идем в каталог /etc/sysconfig/network-scripts и переименовываем файл описания нашего первого интерфейса ifcfg-eth0 в ifcfg-bond0:

cp ifcfg-eth0 ifcfg-bond0

Полученный нами файл ifcfg-bond0 мы должны отредактировать так, чтобы он принял примерно следующий вид:

```
DEVICE=bond0
IPADDR=192.168.1.1
NETMASK=255.255.255.0
NETWORK=192.168.1.0
BROADCAST=192.168.1.255
ONBOOT=yes
BOOTPROTO=none
USERCTL=no
```

Естественно вы должны указать свои собственные ip-адрес, маску, адрес сети и broadcast вместо 192.168.1 и пр. Надо заметить, что мы не удаляли никакие строчки из этого файла, просто сделали изменения в нужных местах и может быть добавили что-то. Таким образом мы создали файл описания нашего виртуального сетевого интерфейса. Следующим шагом будет создание файлов описания для двух наших реальных физических интерфейсов eth0 и eth1, в которых мы укажем, что они входят в состав связанного канала. Файлы ifcfg-eth0 и ifcfg-eth1 у нас должны иметь следующее содержимое:

файл ifcfg-eth0	**файл ifcfg-eth1**
DEVICE=eth0	DEVICE=eth1
USERCTL=no	USERCTL=no
ONBOOT=yes	ONBOOT=yes
MASTER=bond0	MASTER=bond0
SLAVE=yes	SLAVE=yes
BOOTPROTO=none	BOOTPROTO=none

Теперь нам осталось только поднять сетевой интерфейс выполнив команду

/etc/rc.d/init.d/network start

Если дистрибутив вашей системы не позволяет применять master/slave нотификацию при конфигурации сетевых интерфейсов, то вам придется поднимать интерфейс связанного канала вручную, используя следующую последовательность команд:

/sbin/ifconfig bond0 192.168.1.1 up netmask 255.255.255.0
/sbin/ifenslave bond0 eth0
/sbin/ifenslave bond0 eth1

Соответственно вместо 192.168.1.1 вы должны использовать тот ip-адрес, который вам нужен, и указать правильную маску подсети; приведенные выше строчки только пример. Чтобы не выполнять эти команды вручную каждый раз, запишите их в какой-нибудь startup-скрипт, например в /etc/rc.d/rc.local, или замените ими ту часть скрипта /etc/rc.d/init.d/network, которая отвественна за поднятие сетевого интерфейса.

Как вы заметили, для ручного поднятия интерфейса мы использовали команду ifenslave. Это не стандартная системная команда. Программа ifenslave была разработана в рамках проекта Beowulf и вам придется скомпилировать ее из исходных кодов, которые вы можете взять непосредственно на сайте проекта http://beowulf.org/software/ifenslave.c или с сайта проекта Debian: ifenslave_0.07.orig.tar.gz, ifenslave_0.07-1.diff.gz. Естественно, все это вы можете найти в разделе Download этого сайта. Компиляция программы происходит следующей командой:

gcc -Wall -Wstrict-prototypes -O -I/usr/src/linux/include ifenslave.c -o ifenslave

Не забудьте только положить полученный исполняемый файл в /usr/sbin.

Если по каким то причинам вам нужно, чтобы все сетевые драйверы были загружены до загрузки bonding-драйвера, добавьте ниже приведенную строчку в файл /etc/modules.conf. Эта инструкция укажет системе, что в случае поднятия интерфейса bond0 утилита modprobe должна сначала загрузить драйверы для всех ваших сетевых интерфейсов.

probeall bond0 eth0 eth1 bonding

Собственно на этом настройка channel bonding закончена. Если сетевой интерфейс поднялся без ошибок, то проверить этот знаменательный факт можно используя обычную команду ifconfig. Запустив ее без параметров вы должны увидеть нечто подобное:

```
[root]# /sbin/ifconfig
bond0     Link encap:Ethernet  HWaddr 00:C0:F0:1F:37:B4
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MASTER MULTICAST  MTU:1500  Metric:1
          RX packets:7224794 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3286647 errors:1 dropped:0 overruns:1 carrier:0
          collisions:0 txqueuelen:0

eth0      Link encap:Ethernet  HWaddr 00:C0:F0:1F:37:B4
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500  Metric:1
```

```
          RX packets:3573025 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1643167 errors:1 dropped:0 overruns:1 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:10 Base address:0x1080

eth1      Link encap:Ethernet  HWaddr 00:C0:F0:1F:37:B4
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500  Metric:1
          RX packets:3651769 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1643480 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          Interrupt:9 Base address:0x1400

lo        Link encap:Local Loopback
                  inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:1110 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1110 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
```

Если вы увидели нечто подобное на экране монитора, можете себя поздравить, вы успешно сконфигурировали связанный канал. Как видите, ip- и MAC-адреса всех сетевых интерфейсов у нас получились одинаковыми. Чтобы switch мог нормально работать с таким каналом вам необходимо настроить Link Aggrigation. Как это делать вы можете прочитать в документации вашего коммутатора. Для разных моделей коммутатров и разных версий их программного обеспечения это может делаться по-разному. Поэтому в данной книге мы опустим вопросы настройки Link Aggrigation на коммутаторах.

В интернете встречаются сообщения, что в некоторых случаях, после поднятия виртуального сетевого интерфейса дополнительные каналы не могут сразу принимать входящие покеты. Это может произойти по причине того, что новый MAC-адрес дополнительных каналов не прописывается физически в EPROM сетевой карты, в результате чего при старте компьютера свитч не знает о том, что этот MAC-адрес присоединен к более чем одному порту. Для того, чтобы сообщить свитчу правильный набор MAC-адресов достаточно нпосредственно после поднятия интерфейса выполнить несколько пингов. После того, как ICMP-пакеты пройдут через коммутатор по всем виртуальным каналам, внутренняя таблица коммутатора примет правильный вид и в дальнейшем проблем с приемом пакетов не будет.

Коммуникационные технологии

На данной странице перечислены наиболее распространенные высокоскоростные сетевые и коммуникационные аппаратные технологии, которые используются для связи вычислительных узлов при построении кластеров. Для каждой технологии приведена следующая информация: название, компании-производители сетевого оборудования, показатели производительности и примерные цены на оборудование (по состоянию на 2003 год), программная поддержка, информация об архитектуре, дополнительные комментарии.

Fast Ethernet

Производители оборудования	Intel, CISCO, 3Com и др.
Показатели производительности	Пиковая пропускная способность - 100 Mbit/sec (12.5 MB/sec), полный дуплекс. В рамках MPI достигаются скорости порядка 6-7 MB/sec.
Примерные цены	Сетевой адаптер - $15, 16-портовый коммутатор Intel InBusiness Switch - $175
Программная поддержка	Драйверы для всех версий UNIX и Windows NT, протоколы TCP/IP, MPICH - бесплатная переносимая реализация MPI.
Комментарии	Преимуществами данной технологии являются: хорошая стандартизация и широкое распространение, а также низкие цены на

	оборудование и использование стандартных сетевых кабелей (UTP).

Gigabit Ethernet

Производители оборудования	Intel, 3Com и др.
Показатели производительности	Пиковая пропускная способность - 1 Gbit/sec (125 MB/sec), полный дуплекс. В рамках TCP/IP достигаются скорости порядка 500 Mbit/sec (60 MB/sec), в рамках MPI - до 45 MB/sec
Примерные цены	Сетевой адаптер Intel EtherExpress PRO/1000 MT - $50-$80. Коммутатор D-Link DGS-1016T 16-port UTP 10/100/1000 - $900.
Программная поддержка	Драйверы для многих версий UNIX и Windows NT, протоколы TCP/IP.
Комментарии	Преимуществом данной технологии является совместимость и возможность плавного перехода с технологий Ethernet/Fast Ethernet.

Myrinet 2000 / Myrinet-10G

Производители оборудования	Myricom
Показатели производительности Myrinet-2000 (в скобках - для 10G)	Пиковая пропускная способность - 2 Gbit/sec (10 Gbit/sec), полный дуплекс. В рамках TCP/IP достигаются скорости порядка 1.7-1.9 Gbit/sec (9.6 Gbit/sec). Латентность - порядка 2 мксек. На MPI-приложениях латентность составляет около 10 мксек, скорость передачи данных - до 200 MB/sec (до 400 MB/sec на дуплексных операциях).
Примерные цены	Адаптер 10G-PCIE-8A-S+MX1 для шины PCI-Express - $815
Программная поддержка	Драйвера для Linux (Alpha, x86, PowerPC, UltraSPARC), Windows (x86), Solaris (x86, UltraSPARC) и Tru64 UNIX. GM - интерфейс программирования на нижнем урове. Пакеты HPVM (включает MPI-FM, реализацию MPI для Myrinet), BIP-MPI и др.
Комментарии	Myrinet является открытым стандартом. Myricom предлагает широкий выбор сетевого оборудования по сравнительно невысоким ценам. На физическом уровне поддерживаются сетевые среды SAN (System Area Network), LAN (CL-2) и оптоволокно. Технология Myrinet-10G использует кабели стандарта 10Gbit-Ethernet и карты от Myricom могут работать как 10Gbit-Ethernet карты. Технология Myrinet дает высокие возможности масштабирования сети и в настоящее время очень широко используется при построении высокопроизводительных кластеров.

cLAN

Производители оборудования	Giganet
Показатели производительности	Пиковая пропускная способность - 1066 MB/sec.
Примерные цены	Сетевой адаптер cLAN1000-LNX - $795, 8-портовый коммутатор cLAN5000-LNX - $6250.
Программная поддержка	MPI/Pro, продукт компании MSTI - высокопроизводительная реализация MPI для cLAN. Поддерживаются платформы Linux и Windows NT.
Комментарии	Продукты семейства cLAN и MPI/Pro предлагаются в составе технологии кластеризации Gigacluster.

SCI (Scalable Coherent Interface)

Производители оборудования	Dolphin Interconnect Solutions и др.
Показатели производительности	Для продуктов Dolphin: пиковая пропускная способность - 10 GB/sec, полный дуплекс, в рамках MPI достигается около 700 MB/sec. Аппаратная латентность - 0.2 мксек, в рамках MPI - порядка 1,4 мксек.
Примерные цены	На начало 2008г. фиксированных цен не анонсировано.
Программная поддержка	Драйверы для Linux, Windows NT, Solaris. ScaMPI - реализация MPI компании Scali Computer для систем на базе SCI. SISCI API - интерфейс программирования нижнего уровня.
Комментарии	SCI - стандартизированная технология (ANSI/IEEE l596-1992). Кроме стандартной сетевой среды, SCI поддерживает построение систем с разделяемой памятью и когерентностью кэшей. На коммуникационной технологии SCI основаны кластерные системы компании SCALI Computer, системы семейства hpcLine компании Siemens, а также cc-NUMA-сервера Data General и Sequent. Технология SCI использовалась для связи гиперузлов в системах HP/Convex Exemplar X-class.

QsNetII

Производители оборудования	QSW (Quadrics Supercomputers World)
Показатели производительности	Пиковая пропускная способность каналов - 1064 MBytes/sec (достигается 900 MBytes/sec в одном направлении, и 1800 MBytes/sec в режиме Multi-rail). Латентность в рамках MPI - порядка 1,5 мксек. Максимальный размер системы - более 4000 узлов.
Программная поддержка	Поддержка интерфейсов MPI, Shmem, протоколов TCP/IP.
Примерные цены	Около 1700$ за порт.
Комментарии	Сетевое оборудование состоит из коммуникационных процессоров "Elan" и матричных коммутаторов 4x4 "Elite". На базе этих коммутаторов строится единая коммуникационная среда с топологией "fat tree", поддерживается глобальная адресация оперативной памяти. Коммуникационное оборудование QsNet используется в системах AlphaServer SC от Compaq.

MEMORY CHANNEL

Производители оборудования	Compaq Computer, разработано совместно с Encore Computer Corp.
Показатели производительности	Пиковая пропускная способность каналов более 100 MB/sec, латентность - 3 мксек.
Примерные цены	N/A
Программная поддержка	TruCluster Memory Channel Software, реализация MPI.
Комментарии	Технология MEMORY CHANNEL обеспечивает функциональность удаленного доступа к памяти других узлов в кластере (память других узлов может отображаться в локальное адресное пространство). Используется в кластерных системах AlphaServer Array и HPC320/HPC160 от Compaq. На данный момкнт эта технология не поддерживается производителем.

ServerNet II

Производители оборудования	Hewlett Packard

Показатели производительности	Пиковая пропускная способность каналов 180 MB/sec, латентность - 12-29 мксек.
Примерные цены	N/A
Программная поддержка	MPICH - бесплатная переносимая реализация MPI.
Комментарии	Данная технология послужила предтечей InfiniBand

InfiniBand

Производители оборудования	InfiniBand Trade Association
Показатели производительности	Пиковая пропускная способность каналов 10, 20, 30, 40 Gb/sec, латентность - 1,2 мксек.
Примерные цены	N/A
Программная поддержка	OFED - бесплатный пакет драйверов, библиотек, служебного ПО и MVAPICH - реализация MPI, оптимизированная для InfiniBand.
Комментарии	InfiniBand предлагает удалённый прямой доступ в память (remote direct memory access - RDMA), позволяющий доставлять данные непосредственно в память процесса, не вовлекая системные вызовы. Данные могут передаваться 1-о,4-х и 12-ти кратной скоростью. Анонсирован стандарт QDR 12x, позволяющий передевать данные со скоростью 120 GB/sec.

Flat Neighborhood

Производители оборудования	
Показатели производительности	
Примерные цены	N/A
Программная поддержка	
Комментарии	Данная технология является лишь оптимизацией существующих сетевых технологий, посторенных на базе свичей (как правило Ethernet). Её суть состоит в увеличении количества сетевых карт в каждом узле вычислительного кластера и соединении узлов таким образом, чтобы максимально снизить путь пакетов и нагрузку на свичи. Впервые применена в кластере KLAT2.

Сетевая файловая система

Одной из особенностей запуска MPI-программ является необходимость наличия копий программы на всех узлах кластера, на которых она исполняется. Например, если ваши программа *myprog* расположена в каталоге */home/mpiuser/program1*, то на всех узлах кластера должен присутствовать этот каталог и в него должна быть помещена ваша программа.

Это условие пораждает необходимость каким-либо образом распределить копии исполняемого модуля программы между узлами кластера. Аналогичное требование относится и к хранимым на диске данным, которые программа будет использовать.

Существуют различные механизмы, позволяющие выполнять подобное распределение. В большинстве случаев это разнообразные скрипты, осуществляющие синхронизацию каталогов на узлах кластера с помощью команд *scp* или *rsync*. Подобные способы синхронизации имеют свои недостатки. Например, в случае, когда различные копии программы должны обращаться к одним и тем же данным, хранимым на диске, и изменять их

определенным образом, возникает проблема, связанная с необходимостью **постоянной** синхронизации файлов на узлах кластера. Другая проблема возникает при использовании в качестве узлов кластера бездисковых станций. В этом случае вся файловая система таких узлов хранится в оперативной памяти копьютера и чем больше мы закачиваем данных на такой узел, тем меньше остается памяти для выполнения программы.

Для избавления от подобных проблем используются сетевые файловые системы. Существует большое количество реализаций таких систем, как платных, так и распространяемых под лицензией GPL. Мы с вами будем рассматривать сетевую файловую систему NFS, имеющуюся в любом Linux-дистрибутиве общего назначения. Файловая система NFS - это аналог того, что продуктах Майкрософт известно под названием windows share.

Сетевая файловая система NFS состоит из двух компонентов: сервера и клиента. Сервер осуществляет сетевой доступ к каталогам базовой файловой системы на основе определенных правил разграничения доступа. Клиент используется для подключения к расшаренным ресурсам. Установку NFS мы рассматривать не будем, поскольку она достаточно тривиальна. Вы либо в процессе исталляции операционной системы указываете необходимость установки NFS, либо вручную устанавливаете два пакета rpm: nsf-server и nfs_clients. Далее мы рассмотрим процесс конфигурации сетевой файловой системы.

Конфигурация сервера.

Для обеспечения сетевого доступа узлов кластера к расшаренным на сервере кластера ресурсам необходимо вначале разрешить подключение nfs-клиентам к nfs-серверу. Далее будем предполагать, что узлы кластера имеют ip-адреса в диапазоне 192.168.1.2-192.168.1.254. Консоль кластера, к каталогам файловой системы которой мы будем подключаться через NFS, имеет ip-адрес 192.618.1.1. Для разрешения сетевого доступа к NFS с этих адресов мы в файле **/etc/hosts.allow** прописываем следующую строчку:

portmap: 192.168.1.

Точка в конце строки обязательна! Далее мы должны определить к какому каталогу мы разрешаем сетевой доступ. То есть, какой каталог расшариваем. К примеру, мы хотим обеспечить узлам кластера доступ в каталог /home/mpiuser/data-and-progs. Для этого в файле **/etc/exports** прописываем строку:

/home/mpiuser/data-and-progs 192.168.1.0/255.255.255.0(rw,no_root_squash)

На этом настройка серверной части закончена. Чтобы изменения вступили в силу необходимо перезапустить службу NFS с помощью команды "**service portmap restart**".

Конфигурация клиентов.

Переходим от сервера кластара (консоли кластера) к остальным узлам. Все что будет описано ниже необходимо выполнить на каждом компьютере кластера кроме консольного.

Для подключения любой файловой системы (дискеты, раздела диска, сетевого ресурса) используется команда *mount*, если подключение происходит вручную, или запись в файле */etc/fstab*, если подключение происходит в момент загрузки системы. Нас будет интересовать последний случай.

Для обеспечения запуска mpi-программ нам нужно, чтобы содержимое каталога /home/mpiuser/data-and-progs на узлах кластера совпадало с содержимым этого же каталога на консоли кластера. Для этого мы должны в домашнем каталоге пользователь mpiuser (/home/mpiuser) создать пустой каталог data-and-progs. После чего прописать в файле **/etc/fstab** следующую строку:

192.168.1.1:/home/mpiuser/data-and-progs /home/mpiuser/data-and-progs nfs rw 0 0

Чтобы удаленный (сетевой) каталог монтировался автоматически при загрузке узла кластера,

сервис клиента NFS должен запускаться в процедуре начальной загрузки.

На этом установка сетевой кластерной файловой системы завершена. При включении кластера, консоль кластера должна быть загружена до того, как вы начнете включать остальные узлы.

Разделение сетей.

Поскольку сеть является самым узким местом кластера, желательно организовать работу так, чтобы операции межпроцессорной пересылки данных не пересекались с файловыми операциями NFS. Для этого необходимо компьютеры кластера оснастить дополнительными сетевыми картами, объединенными в отдельную сеть, физически не пересекающуюся с первыми картами. То есть использовать дополнительный набор хабов или свитчей. Сетевой интерфейс на вторых картах должен иметь ip-адреса, из другой, отличной от первого интерфейса сети. Например, на сервере кластера сетевая карта, через которую будет осуществляться доступ к расшаренному каталогу имеет адрес 192.168.1.1, а карта, через которую будет происходить межпроцессорное взаимодействие, имеет адрес 192.168.2.1.

Таким образом, NFS мы настраиваем так, как это было описано выше, а при конфигурировании MPI, список узлов кластера составляем из ip-адресов 192.168.2.*.

Естественно, разделение сетей не необходимо, но желательно.

> ***Важное замечание.*** *Файлы, хранящиеся на диске, в условиях параллельной задачи, выполняемой на кластере, могут понадобиться только для сохранения состояния задачи в контрольных точках. Конечно, дисковые ресурсы можно использовать и для организации виртуальной памяти, подгружая по мере необходимости данные в оперативную память, увеличивая тем самым размер разностной сетки. Однако при наличии кластера, увеличение размера разностной сетки логичнее и эффективнее может быть выполнено посредством использования дополнительных вычислительных узлов кластера. Если же дисковые ресурсы используются только для сохранения контрольных точек и эти контрольные точки расположены не в каждой итерации (а в каждой десятой или сотой), то разделение локальной сети кластера на два независимых сегмента (NFS и сеть межпроцессорного обмена данных) является не обязательной. Вполне можно обойтись всего одним сегментом, используя его и для NFS и для обмена данными. Поскольку NFS будет использоваться достаточно редко, то и отрицательное влияние ее на эффективность кластера будет минимально.*

SSH, беспарольный доступ

Рассмотрим процедуру создания беспарольного доступа пользователя user1 с консоли кластера на узлы кластера по протоколу SSH. Беспарольный доступ обеспечит нам более комфортную работу в параллельной виртуальной машине. Так, отпадет необходимость вводить пароли доступа при добавлении в виртуальную машину каждого нового узла и при копировании исполняемых модулей в локальные файловые системы узлов кластера.

Алгоритм обеспечения беспарольного доступа следующий:

1. Логинимся к консоли кластера: ssh user1@server
2. Переходим в каталог ssh: cd ~/.ssh
3. Генерируем rsa-ключи: ssh-keygen -t rsa
4. На вопрос задать имя файла жмем Enter - остается имя по умолчанию id_rsa.
5. На просьбу задать пароль жмем Enter два раза (второй раз для проверки).
6. Копируем публичный ключ на узел кластера: scp id_rsa.pub user1@node1:~/.ssh

7. Логинимся к узлу node1: ssh user1@node1
8. Переходим в каталог ssh: cd ~/.ssh
9. Копируем публичный ключ: cat id_rsa.pub >> authorized_keys2
10. Отключаемся от узла node1
11. Повторяем пункты 6-10 для остальных узлов кластера (node2 ... nodeN)

После проведения вышеописанной процедуры пользователь "user1" сможет подключаться к узлам кластера с консоли кластера не вводя свой пароль. Следует отметить, таким образом бы обеспечиваем беспарольный доступ только для одного пользователя и только в одном направлении: консоль кластера -> узлы кластера. И только для случая, когда user1 подключается к узлам кластера под своим именем.

Распараллеливание программ

После того, как мы воплотим в "железе" кластерную архитектуру, придется задуматься над вопросом а как же ее использовать. Старые линейные методы программирования уже не подходят для написания программ, эффективно использующих многопроцессорную технологию. Необходимо поменять стиль программирования задач. Но для этого надо иметь минимальное представление о том, какими способами можно превратить линейную программу в параллельную. Хотя существуют специальные трансляторы, которые автоматически, без участия програмиста, могут найти в программе куски параллельного кода и дать на выходе исполняемую на кластере задачу, добиться максимальных результатов с помощью таких трансляторов нельзя. О некоторых подобных трансляторах мы поговорим позже, когда будем обсуждать программное обеспечение кластера. Теперь же рассмотрим некоторые теоретические вопросы построения параллельных вычислений.

Распараллеливание программ - это процесс адаптации алгоритмов, записанных в виде программ, для их эффективного исполнения на вычислительной системе параллельной архитектуры. Заключается либо в переписывании программ на специальный язык, описывающий параллелизм и понятный трансляторам целевой вычислительной системы, либо к вставке специальной разметки (например, инструкций MPICH/MPI).

Распараллеливание может быть ручным, автоматизированным и полу-автоматизированным. При распараллеливании важно учитывать не только формальный параллелизм структуры алгоритма, но и то, что обменные операции в параллельных ЭВМ происходят, как правило, значительно медленнее арифметических. С этим связано существование львиной доли накладных расходов на организацию параллелизма.

Целью программиста не должно быть получение правильного результата вычислений любой ценой, но получение правильного результата наибыстрейшим, оптимальным способом. Если программа предназначена для однократного использования, то лучше написать ее как можно проще, не оптимизируя ее быстродействие и используемую память, чтобы потратить минимум усилий на тестирование и отладку. Если программа предназначена для частого использования или время ее работы будет гораздо больше времени ее написания и отладки, то не следует жалеть труда на оптимизацию ее быстродействия.

Для начала разберемся, что мы хотим получить от кластера. Как уже было сказано, использовать параллельные компьютеры имеет смысл только для "тяжелых" задач, которые требуют или большого времени счета или большого объема памяти.

Есть две проблемы, которые всегда встают перед нами, когда мы решаем подобные задачи. Первая: недостаток времени. Если наша задача выполняется в течение шести недель, было бы очень неплохо, если бы время ее счета сократилось до шести дней. Вторая: недостаток памяти. Предположим, к примеру, мы решаем численно систему дифференциальных уравнений на разностной сетке. Размерность сетки всегда ограничена объемом оперативной памяти компьютера. Нет ничего невероятного в том, что увеличивая размерность разностной

сетки (увеличивая детализацию) мы можем получить интересные тонкие эффекты, которые, хотя и описываются исходными уравнениями, но скрыты от нас слишком грубой сеткой.

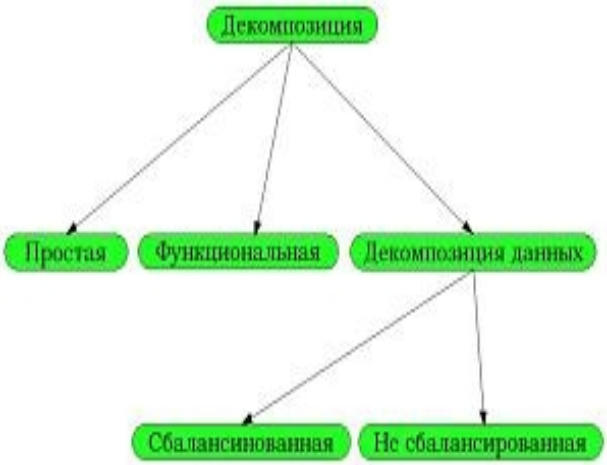
Решением обоих этих проблем является *декомпозиция* задачи. То есть, разделение задачи на части, которые могут быть параллельно исполнены на нескольких машинах кластера. С помощью декомпозиции можно как сократить общее время счета задачи, так и увеличить доступную для задачи оперативную память. Далее рассмотрим подробно, что есть декомпозиция.

Варианты декомпозиции

Распараллеливание программ сводится к структуризации, то есть процессу разделения задачи на независимые процессы, которые не требуют последовательного исполнения и могут, соответственно, быть выполнены на разных процессорах независимо друг от друга.

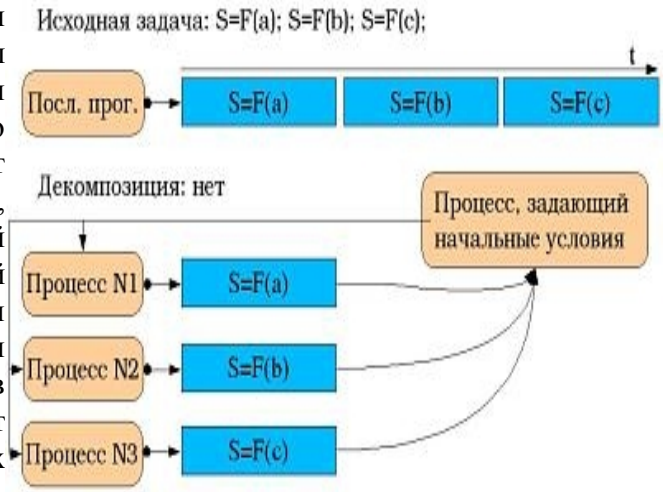
Одним из глобальных видов структурирования высокого уровня для решаемой проблемы является декомпозиция исследуемого физического процесса по составляющим его подпроцессам и, соответственно, сегментация общего алгоритма решения полной задачи на ряд алгоритмов решения составляющих подзадач.

Существует широкий спектр методов декомпозиции задачи. На следующем рисунке представлена классификация таких методов.



Как видно из рисунка, существует три основных варианта декомпозиции: простая декомпозиция (trival), функциональная (functional) и декомпозиция данных. Вопрос об использовании того или иного типа декомпозици при написании параллельной программы решается исходя из структуры самой задачи. Причем, в зависимости от условий, можно использовать сразу несколько типов.

Тривиальная декомпозиция



Как следует из названия, тривиальная декомпозиция наиболее простой тип декомпозиции. Применяется она в том случае, когда различные копии линейного кода могут исполняться независимо друг от друга и не зависят от результатов, полученных в процессе счета других копий кода. Проиллюстрировать подобный вариант можно на примере решения задачи методом перебора или Монте-Карло. В этом случае одна и та же программа, получив различные начальные параметры, может быть запущена на различных процессорах кластера. Как легко заметить, программирование таких параллельных процессов ничем не отличается обычного программирования на последовательном компьютере, за исключением маленького участка кода, отвечающего за запуск копий программы на процессорах кластера и затем ожидающего окончания счета запущенных программ.

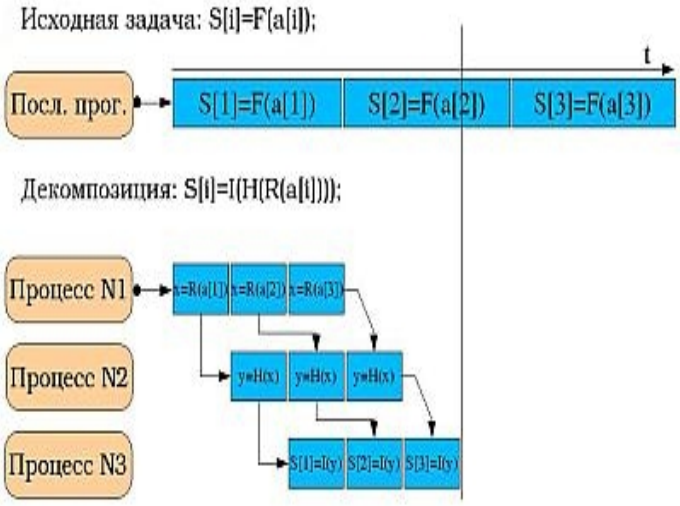
Функциональная декомпозиция

При функциональной декомпозиции исходная задача разбивается на ряд последовательных действий, которые могут быть выполнены на различных узлах кластера не зависимо от промежуточных результатов, но строго последовательно.

Предположим наша задача сводится к применению некоего функционального оператора к большому массиву данных: S[i]=F(a[i]). Предположим также, что выполнение функции F над одним элементом массива занимает достаточно большое время и нам хотелось бы это время сократить. В этом случае мы можем попытаться представить исходную функцию как композицию нескольких фунуций: S(a[i])=I(H(R(a[i]). Произведя декомпозицию мы получим систему последовательных задач:

x=r(a[i]);
y=h(x);
b[i]=i(y);

Каждая из этих задач может быть выполнена на отдельном узле кластера. Как можно заметить общее время обработки одного элемента массива a[i] в результате не изменяется, а скорее немного увеличивается за счет межпроцессорных пересылок. Однако общее время обработки всего массива заметно снижается за счет того, что в нашем примере одновременно идет обработка сразу трех элементов массива.



У данного метода декомпозиции есть пара особенностей, о которых надо помнить.

Первая особенность сосотоит в том, что выход кластера на максимальную эффективность происходит не сразу после запуска задачи, а постепенно, по мере того, как происходит частичная обработка первого элемента массива. Второй и третий процессоры в нашем примере, которые отвечают за выполнение функций g(x) и f(y), будут простаивать до тех пор, пока не закончится выполнение функции h(a[1]) на первом процессоре. Третий процессор будет простаивать до окончания выполнения функции g(a[1]). По аналогичному сценарию, только в зеркальном отображении, происходит окончание работы.

Вторая особенность заключается в выборе декомпозированных функций h,g,f. Для уменьшения времени простоя процессоров в ожидании следующей порции работы необходимо таким образом подбирать декомпозированные функции, чтобы время их работы было примерно одинаковым.

По приведенному нами сценарию данные обрабатываются в режиме конвейера. На программиста, выбравшего функциональный тип декомпозиции задачи, ложится обязанность не только по выбору декомпозированных функций, но и по организации работы параллельных частей программы в режиме конвеера, то есть правильно организовать процедуры получения исходных данных от предыдущего процесса и передачи обработанных данных следующему процессу.

Декомпозиция данных

В отличие от функциональной декомпозиции, когда между процессорами распределяются

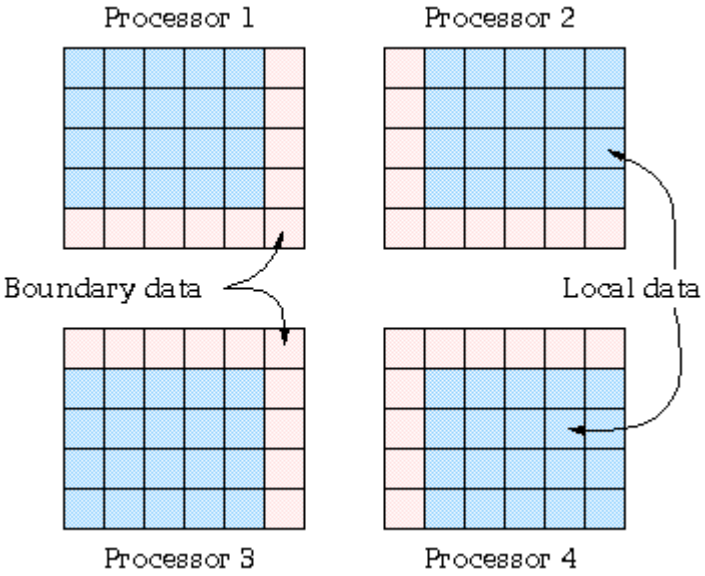
различные задачи, декомпозиция данных предполагает выполнение на каждом процессоре одной и той же задачи, но над разными наборами данных. Части данных первоначально распределены между процессорами, которые обрабатывают их, после чего результаты суммируются некоторым образом в одном месте (обычно на консоли кластера). Данные должны быть распределены так, чтобы объем работы для каждого процессора был примерно одинаков, то есть декомпозиция должна быть сбалансированной. В случае дисбаланса эффективность работы кластера может быть снижена.

В случае, когда область данных задачи может быть разбита на отдельные непересекающиеся области, вычисления в которых могут идти независимо, мы имеем регулярную декомпозицию.

Regular Domain Decomposition

Очень много задач базируются на обработке больших массивов данных, структурированных в регулярную решетку (матрицу). Как один из многочисленных примеров можно упомянуть численное моделирование газодинамических процессов. В том случае, когда обрабатываемая решетка данных может быть разбита на регулярный (не пересекающийся) массив подрешеток (областей), задача может быть распределена между процессорами кластера и решена в параллельном режиме. Это позволит или сократить время решения задачи, или поставить задачу для более крупного массива данных (например сделать разностную сетку более мелкой).

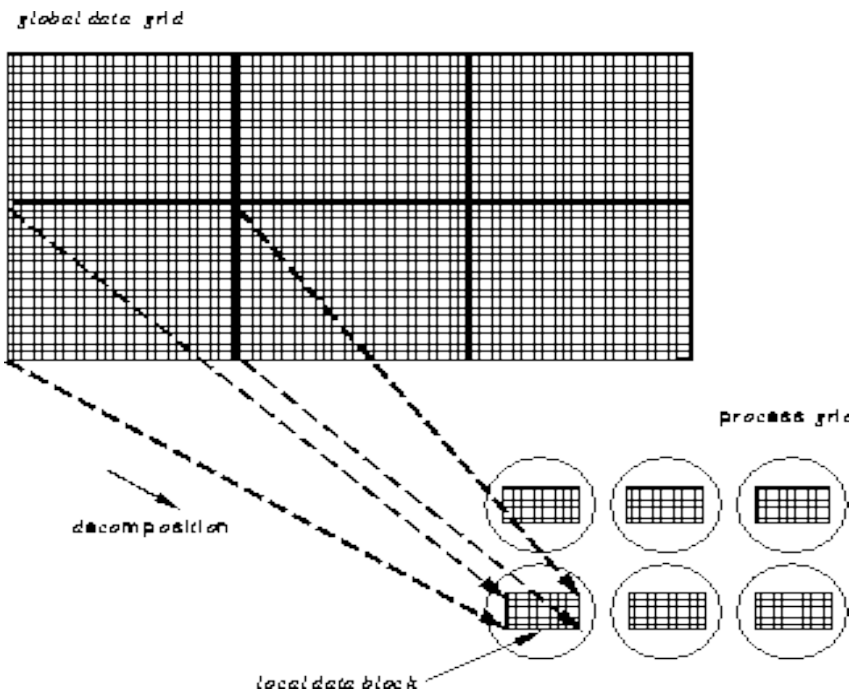
Для многих инженерных и научных задач декомпозиция данных является наиболее подходящим способом подготовить программу для исполнения на параллельной машине. Регулярная декомпозиция исходной решетки может быть проведена в виде смежных кусков, как это представлено на рисунке ниже или каким либо другим, подходящим для задачи образом.



Наибольшая эффективность достигается в случае, когда вычисления производятся в основном локально. Другими словами, когда для изменения ячейки данных требуются информация только из ближайшего окружения. В этом случае вычисления происходят полностью параллельно, а межпроцессорные взаимодействия требуются только для вычисления граничных данных.

Сетка процессов

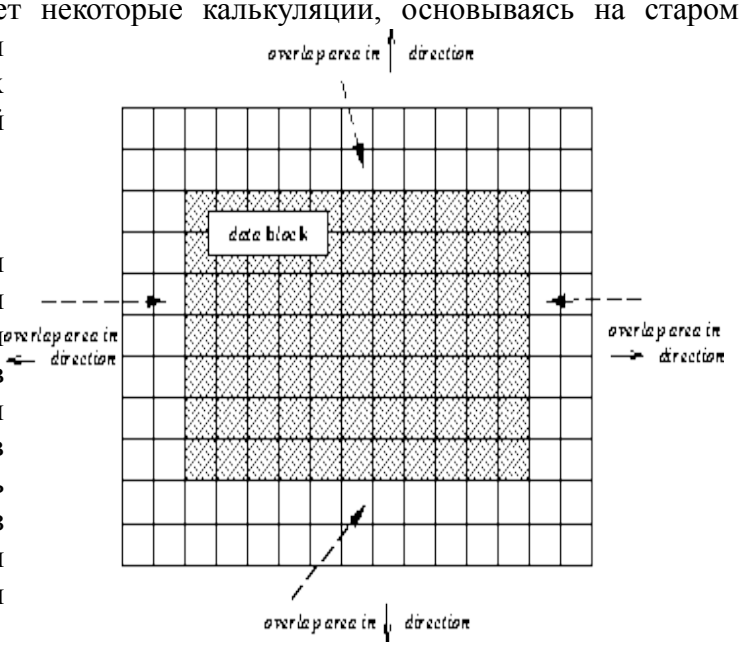
Метод регулярной декомпозиции сосотоит в разбиении исходной большой сетки данных на непересекающиеся регулярные области (подсетки) и распределении этих подсеток между процессорами, где эти части данных могут быть параллельно обработаны. Другими словами глобальный набор данных декомпозируется на секции и каждая секция передается под контроль отдельного процесса так, как это показано на следующем рисунке.



Перед тем, как сетка данных будет распределена между процессами, сами процессы должны быть организованы в логическую структуру, соответствующую структуре данных. Другими словами процессы не являются копиями одной и той же программы, хотя и могут выполнять одинаковые по сути действия. Каждый процесс должен, во-первых, учитывать какого типа данные он обрабатывает. В том же примере с газодинамикой процессы можно разделить на те, которые вычисляют плотность, скорость или гравитацию. Во-вторых, процесс должен учитывать откуда берутся данные, из центра глобальной сетки или с ее края. Набор данных, распределенных процессу в дальнейшем будем называть блоком данных этого процесса.

Изменение элементов данных

На каждой итерации, каждый процесс берет на себя работу по изменению значений элементов данных, которые содержатся в его локальном блоке данных. Вычисляя новое значение элемента, процесс выполняет некоторые калькуляции, основываясь на старом (старых) значении этого элемента и возможно также на старых значениях элементов, находящихся в некоторой близости от обрабатываемого элемента.



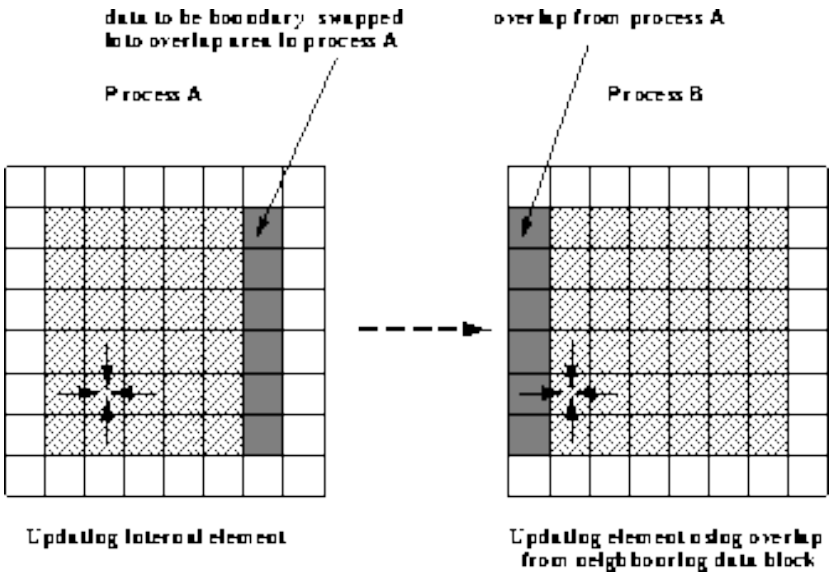
Область перекрытия

При вычислении нового значения элемента, может сложиться такая ситуация, когда не все необходимые для вычисления соседи находятся в локальном блоке данных процесса. Для доступа к элементам из других блоков данных каждый процесс должен иметь "теневую" область на которую из соседних процессов отображаются элементы, окружающие локальный для данного процесса блок данных.

Элементы в теневой области должны быть доступны для процесса в режиме "только чтение". Такие теневые области называются областями перекрытия. Локальный блок данных окруженный областью перекрытия мы будем называть массивом данных. Надо заметить, что массив данных процесса не обязательно должен иметь области перекрытия одинаковой толщины по всем направлениям. В некоторых случаях область перекрытия может отсутствовать по одному, нескольким или всем направлениям или иметь разную толщину по разным направлениям. Также, форма области перекрытия может отличаться у разных процессов. Толщина области перекрытия для каждого процесса определяется особенностями выбранного численного алгоритма.

Граничный обмен

Поскольку изменение значения пограничного эелемента локального блока данных процесса требует наличия элементов в области перекрытия, каждый процесс должен быть готов послать копию граничных элементов своего локального блока данных процессу, которому эти данные необходимы. Аналогично, каждый процесс должен быть в состоянии принять копии элементов от соседних процессов в свою область перекрытия.



На следующем рисунке показаны два процесса A и B. Процесс A выполняет изменение значения элемента (отмеченного стрелками), который находится далеко от границы и, таким образом, значения элементов области перекрытия для вычисления не используются. Процесс B выполняет обработку элемента, для изменения значения которого требуется значения элементов из области перекрытия с процессом A. Процесс A должен выполнить граничный обмен с процессом B до того момента, когда процессу B понадобится обратиться к элементам в области перекрытия.

Следует отметить, что чем шире оказываются области перекрытия, тем менее эффективным становится распараллеливание программ методом регулярной декомпозиции, поскольку процесс граничного обмена будет занимать все более длительный промежуток времени. Однако, это зависит от соотношения времен страничного обмена и собственно счета одной итерации. Если процесс итерации достаточно длителен, то граничный обмен даже при максимально больших областях перекрытия может слабо тормозить вычисления.

Детализация

Под детелизацией понимается степень востребованности данных из области перекрытия при обработке элементов глобального блока данных. В случае, когда вычисление каждого элемента глобального блок данных требует от процессов доступа к областям перекрытия, мы имеем высокую степень детализации (fine granularity). В случае, когда ни один элемент из глобального блока данных при его обработке процессами не требует доступа в области перекрытия или таковые отсутствуют, мы имеем грубую степень детализации (coarse granularity).

Чем выше степень детализации, тем менее эффективным становится параллельный процесс. Степень детализации определяется как типом выбранного для решения задачи численного алгоритма, так и количеством процессов на которое мы раздробим исходную задачу.

Если применяемый численный алгоритм требует при калькуляции доступа к соседним элементам, то увеличение количества процессов (увеличение степени дробления исходной сетки данных) увеличивает объем областей перекрытия и, соответственно, количество элементов, при обработке которых требуется доступ к этим областям. Таким образом, увеличение количества параллельных процессов увеличивает степень детализации. Наступает определенный момент, когда приращение эффективности параллельной задачи за счет увеличения количества процессов станет меньше, чем снижение эффективности из-за увеличения степени детализации.

Граничный обмен

Как уже кратко было упомянуто, часто бывает необходимо для вычисления нового значения элемента данных (проведения итерации) организовать в одном из процессов доступ к данным, полученным в результате работы другого процесса.

Когда значение какого-то элемента данных изменяются, новое значение этого элемента может зависить как от старого значения, так и от старых значений некоторго количества соседних элементов, часть из которых может находится в локальном блоке данных другого процесса. В этом случае копии таких элементов должны быть переданы в процесс из других процессов. С другой стороны, процесс должен быть готов к передаче результатов своей работы процессам, которым эти данные могут понадобится для проведения следующей итерации.

Таким образом для процессов можно предложить следующий алгоритм выполнения итераций:

- посылаем собственные данные процессам, которым они могут понадобится для проведения следующей итерации
- принимаем данные от соседних процессов для заполнения собственной области перекрытия новыми значениями.
- выполняем обработку данных (выполняем итерацию)

Процесс посылки и приема данных от других процессов называется "граничный обмен".

Интерфейс передачи сообщений (MPI)

MPI расшифровывается как "Message passing interface" ("Интерфейс передачи сообщений"). MPI - это стандарт на программный инструментарий для обеспечения связи между отдельными процессами параллельной задачи. MPI предоставляет программисту единый механизм взаимодействия процессов внутри параллельно исполняемой задачи независимо от машинной архитектуры (однопроцессорные, многопроцессорные с общей или раздельной памятью), взаимного расположения процессов (на одном физическом процессоре или на разных) и API операционной системы. Программа, использующая MPI, легко отлаживается и переносится на другие платформы, часто для этого достаточно простой перекомпиляции исходного текста программы.

Решение использовать в своих задачах MPI следует осторожно, после тщательного взвешивания своих сил и возможностей как программиста. Несмотря на то, что MPI представляет собой значительный шаг вперед по сравнению с предшествующим поколением библиотек передачи сообщений, а, возможно и вследствие этого, программировать на MPI достаточно сложно. Причиной тому явялется не недостаток стандарта, а в самой идеологии передачи сообщений. MPI можно рассматривать как уровень ассемблера для параллельных программ. Вы любите програмировать на ассемблере?

Основное отличие стандарта MPI от его предшественников (p4, PVM) - понятие коммуникатора. Все операции синхронизации и передачи сообщений локализуются внутри коммуникатора. С коммуникатором связывается группа процессов. В частности, все коллективные операции вызываются одновременно на всех процессах, входящих в эту группу. Поскольку взаимодействие между процессами инкапсулируется внутри коммуникатора, на базе MPI можно создавать библиотеки параллельных программ.

В настоящее время разными коллективами разработчиков написано несколько программных пакетов, удовлетворяющих спецификации MPI, в частности: MPICH, LAM, HPVM, OpenMPI и так далее. В двух словах охарактеризуем наиболее распространенные из этих пакетов. Если говорить о LAM, то основное достоинство этого пакета - богатые отладочные возможности. Трассировка обращений к MPI и анализ состояния параллельной программы после аварийного завершения делают процесс отладки менее тяжелым и более продуктивным. С другой стороны, пакет MPICH более мобилен, следуя простым инструкциям можно перенести MPI на новую платформу (например с Linux на Windows или наоборот). Для этого потребуется необходимо написать лишь несколько драйверов нижнего уровня. Установка библиотеки MPICH проходит несколько труднее, чем установка LAM MPI, поскольку приходится задавать гораздо большее число параметров, причем назначение некоторых из них известно только разработчикам. Если же говорить об эффективности, то есть мнение, что MPICH несколько эффективнее передает сообщения. Спорить с этим не буду, но мне лично убедиться в этом не привелось. Что касается отладки, то отлаживать программы в среде MPICH труднее.

MPI - это хорошо стандартизованный механизм для построения программ по модели обмена сообщениями. Существуют стандартные "привязки" MPI к языкам C/C++, Fortran 77/90. Имеются и бесплатные и коммерческие реализации почти для всех суперкомпьютерных платформ, а также для High Performance кластерных систем, построенных на узлах с ОС Unix, Linux и Windows. В настоящее время MPI - наиболее широко используемый и динамично развивающийся интерфейс из своего класса. За стандартизацию MPI отвечает MPI Forum (http://www.mpi-forum.org). В новой версии стандарта 2.0 описано большое число новых интересных механизмов и процедур для организации функционирования параллельных программ: динамическое управление процессами, односторонние коммуникации (Put/Get), параллельные I/O. Но к сожалению, пока нет полных готовых реализаций этой версии стандарта, хотя часть из нововведений уже активно используется.

Основные понятия MPI. Парадигма SPMD

При запуске задачи создается группа из P процессов. Группа идентифицируется целочисленным дескриптором (коммуникатором). Внутри группы процессы нумеруются от 0 до P-1. В ходе решения задачи исходная группа (ей присвоено имя MPI_COMM_WORLD) может делиться на подгруппы, подгруппы могут объединяться в новую группу, имеющую свой коммуникатор. Таким образом, процесс имеет возможность одновременно принадлежать нескольким группам процессов. Каждому процессу доступен свой номер myProc внутри любой группы, членом которой он является.

Поведение всех процессов описывается одной и той же программой. Межпроцессные коммуникации в ней программируются явно с использованием библиотеки MPI, которая и диктует стандарт программирования. Квазиодновременный запуск исходной группы процессов производится средствами операционной системы. При этом P определяется желанием пользователя, а отнюдь не количеством доступных процессоров!

Итак, все P процессов асинхронно выполняют одну и ту же программу. Но у каждого из них свой номер myProc. Поэтому в программе, естественно, будут такие фрагменты:

```
if (myProc.eq.0) then
    < делать что-то одно >
else if (myProc.eq.1) then
    < делать что-то другое >
          .  .  .
else
    < делать что-то P-е >
endif
```

Таким образом, в программе "под одной крышей" закодировано поведение всех процессов. В этом и заключена парадигма программирования Single Program - Multiple Data (SPMD).

Обычно поведение процессов одинаково для всех, кроме одного, который выполняет координирующие функции, не отказываясь, впрочем, взять на себя и часть общей работы. В качестве координатора обычно выбирают процесс с номером 0.

Установка системы MPI

Установка системы MPI на компьютерах кластера аналогична установки PVM, в том смысле, что установка сводится к компиляции системы из исходников. В вопросе выбора версии MPI вы можете чувствовать себя свободными и руководствоваться собственными пристрастиями. С моей точки зрения наиболее простыми в использовании являются пакеты MPICH и OpenMPI, которые в отличие от LAM/MPI не трубуют запуска дополнителных демонов и требуют минимальной настройки. Моя личная рекомендация - OpenMPI. Этот пакет в настоящее время активно развивается и меет хорошую интеграцию с системами управления очередями и grid-системами. Кроме того пакет MPICH перестал развиваться с 2005 года.

Что же касается LAM/MPI, то этот пакет LAM/MPI является реализацией протокола, ориентированного на архитектуру параллельного компьютера, основанную на сети рабочих станций. Установка LAM/MPI требует немного меньше усилий, по сравнению с MPICH. Что касается программ, написанных с использованием стандарта передачи сообщений MPI, то они без изменения исходного кода будут одинаковым образом работать в среде обоих пакетов. Далее мы обсудим вопросы установки, администрирования и использования обоих пакетов.

Первым шагом в установке MPI является получение исходных кодов пакета. Взять исходники можно либо непосредственно на сайтах разработчиков: http://www.mcs.anl.gov/mpi/mpich/, http://www.lam-mpi.org/, либо в разделе Download этого сайта. Текущая версия пакета MPICH: 1.2.7p1, LAM/MPI: 7.1.3, OpenMPI: 1.3.3.

Получив архив mpich.tar.gz, lam-7.1.3.tar.gz или openmpi-1.3.3.tar.bz2, вы должны распаковать его в каком-либо каталоге вашей файловой системы и запустить скрипт конфигурации configure:

MPICH

```
./configure -with-arch=LINUX -with-device=ch_p4 -rsh=/usr/bin/ssh \
        --prefix=/usr/local/mpich-1.2.6/ch_p4
```

LAM/MPIH

```
./configure --prefix=/usr --with-rsh="/usr/bin/ssh -x"
```

OpenMPI

```
./configure --prefix=/usr
```

В параметрах скрипта configure вы определяете тип архитектуры машины (только для MPICH), на которой будет установлен пакет MPI (в данном случае LINUX) и путь к каталогу, в который пакет будет установлен (/usr/local/mpich-1.2.6/ch_p4 или /usr). Следует отметить, Что на всех узлах кластера вы должны установить MPI в один и тот же каталог. Будучи запущеным, скрипт configure обследует вашу операционую систему и подготовит пакет MPI к компиляции с учетом ее особенностей.

По умолчанию MPI использует rsh в качестве средства межузловых коммуникаций. Как уже прежде говорилось, по некоторым причинам предпочтительнее заменить rsh на более комфортный в администрировании ssh, обеспечив при этом беспарольный доступ к узлам кластера с консольной машины. Для этого при запуске скрипта configure мы используем параметр -rsh=/usr/bin/ssh для MPICH и --with-rsh="/usr/bin/ssh -x" для LAM/MPI. Если программа ssh находится в вашей системе в другом месте, то значение параметра -rsh или --with-rsh должно быть соответствующим образом изменено.

Как можно заметить, параемтр --prefix, определяющий каталог, куда будет установлен пакет, указывает для LAM/MPI на системную область, а для MPICH на отдельный каталог. Сделано это потому, что пакет MPICH по какой-то причине не поддерживает команду деинсталляции "make uninstall". В случае, когда вам по какой-то причине надо будет удалить из системы пакет MPICH, сделать это будет гораздо проще, когда он находится в каком-то одном своем каталоге, вместо того, чтобы долго и нудно вычищать системную область.

Следующим шагом является собственно компиляция и установка MPI. Выполняются эти действия последовательным запуском двух команд, одинаковых для обоих пакетов:

```
make
make install
```

Первая из команд компилирует пакет MPI, вторая - устанавливает пакет в систему, перемещая полученные на этапе компиляции бинарные и другие необходимые файлы в область файловой системы, определенную параметром --prefix команды configure.

На этом собственно установка системы MPI заканчивается.

Конфигурация кластера MPICH

В отличие от PVM описание кластера выполняется не командами системы, а посредством редактирования соответствующего конфигурационного файла. Для Linux-системы это файл /usr/local/mpich-1.2.6/ch_p4/share/machines.LINUX. Этот файл содержит простое перечисление компьютеров, входящих в кластер и может выглядеть следующим образом:

```
server
node1
node2.mydomain.com
192.168.1.33
node4:2
```

То есть, может использоваться либо краткое имя узла, либо доменное имя узла, либо его ip-адрес. Правило: одни узел - одна строчка. В описании узла "node4" в примере был использован модификатор ":2". Это означает, что в качестве четвертого узла используется двухпроцессорная (SMP) машина.

Файл machines.LINUX должен быть одинаковым на всех узлах кластера. (В действительности это не так. Различием этого файла можно задавать более сложную структуру кластера, но в данной книге мы будем рассматривать наиболее простые варианты использования виртуальных параллельных машин.)

Для проверки работоспособности MPI необходимо запустить скрипт

/usr/local/mpich-1.2.6/ch_p4/sbin/testmachines:

/usr/local/mpich-1.2.6/ch_p4/sbin/testmachines -v LINUX

В качестве единственного параметра должна быть указана архитектура проверяемых машин кластера (в нашем случае LINUX). В случае нормального завершения всех тестов вывод на консоль должен иметь примерно следующий вид:

```
[yuri@server sbin]# ./tstmachines -v LINUX
Trying true on server ...
Trying ls on   server   ...
Trying user program on   server   ...
Trying true on node1 ...
Trying ls on  node1  ...
Trying user program on  node1  ...
Trying true on node2 ...
Trying ls on  node2  ...
Trying user program on  node2  ...
[yuri@server sbin]#
```

Скрипт tstmachines, пытаясь проверить доступность узла кластера, последовательно пытается запустить на нем программы /bin/true, /bin/ls и некую пользовательскую программу /usr/local/mpich-1.2.6/ch_p4/sbin/tstfoo. Когда я устанавливал MPI на своих компьютерах, у меня неожиданно возникли проблемы с этой программой. Система ее попросту не находила. Мне пришлось самостоятельно скомпилировать эту программу и положить полученный исполняемый файл в каталог /usr/local/mpich-1.2.6/ch_p4/sbin/ на всех узлах кластера. Исходный текст программы tstfoo на языке C выглядит так:

```
main() {return 0;}
```

Конфигурация кластера LAM/MPI

Аббревиатура LAM в названии пакета расшифровывается как "Local Area Machine", что указывает на изначальную ориентацию пакета на использование его для кластера, построенного из сети рабочих станций.

Прежде чем MPI программы могут быть запущены на исполнение в систему кластера должна быть виртуальная машина LAM. Обычно этот процесс называется "загрузка LAM". Успешно выполненный процесс загрузки виртуальной машины устанавливает в системе операционную среду окружения, называемую "вселенная LAM" (LAM universe). Вселенная LAM может состоять как из нескольких машин, так и из одной машины. В последнем случае LAM может быть использована для симуляции параллельной машины, например для отладки программ.

Вселенная LAM описывается в ***файле схемы загрузки*** (boot schema file), который содержит информацию о том, какие рабочие станции входят в виртуальную машину. Файл схемы загрузки, которому в дальнейшем мы дадим имя *hostfile*, является простым текстовым файлом, содержащем адреса машин, один адрес в одной строке. Место расположения этого файла может быть любое. Содержание его может быть например таким:

```
# My boot schema
node1.cluster.example.com
192.168.1.123
node3.cluster.example.com cpu=2
```

Первая строчка - это комментарий. Остальные строчки - это перечисление машин, входящих в кластер. Первая машина задана доменным именем. Ссылка на вторую машину задана ее ip-

адресом. Третья машина также описана доменным именем с параметром "cpu=2". Параметр этот означает, что машина node3 является двупроцессорным SMP компьютером.

Для загрузки вселенной LAM используется команда lamboot, запуск которрой выглядит следующим образом:

```
[yuri@server yuri]$ lamboot -v -ssi boot rsh ./hostfile

LAM 7.0.6/MPI 2 C++/ROMIO - Indiana University

n-1<29699> ssi:boot:base:linear: booting n0 (node1.cluster.example.com)
n-1<29699> ssi:boot:base:linear: booting n1 (192.168.1.123)
n-1<29699> ssi:boot:base:linear: booting n2 (node1.cluster.example.com)
n-1<29699> ssi:boot:base:linear: finished
```

Для успешного запуска LAM должны быть выполнены следующие условия:

- все машины, описанные в hostfile должны быть включены и доступны по сети
- пользователь должен иметь беспарольный доступ к этим машинам по протоколу SSH
- бинарники системы LAM на этих машинах должны находится в каталогах, указанных в переменной окружения PATH
- если машина описана доменным именем, то она должна быть прописана в системе DNS или в системном файле hosts

Посмотреть текущую конфигурацию кластера можно с помощью команды lamnodes:

```
[yuri@server yuri]# lamnodes
n0 node1.cluster.example.com:1
n1 192.168.1.123:1
n2 node3.cluster.example.com:2
```

Остановить работы LAM-вселенной можно командой lamhalt

Конфигурация кластера OpenMPI

Конфигурация OpenMPI заключается всего лишь в редактирования соответствующего файла, описывающего узлы кластера. Название файла и его местоположение может быть произвольным. Этот файл содержит простое перечисление компьютеров, входящих в кластер и может выглядеть следующим образом:

server
node1
node2.mydomain.com
192.168.1.33

То есть, может использоваться либо краткое имя узла, либо доменное имя узла, либо его ip-адрес. Правило: одни узел - одна строчка.

Для проверки работоспособности OpenMPI необходимо на параллельное исполнение любую простую программу, например **hostname**, которая покажет имя хоста, на котором она запущена. Делается это следующей командой:

mpirun -hostfile mpi.host -np 4 hostname

Команда **mpirun** имеет три параметра. Первый (*-hostfile*) указывает на файл, содержащий список узлов кластера. Второй (*-np*) задает количество процессоров (узлов кластера), на которых эта программа будет запущена. И третий параметр - собственно сама программа, которая будет запущена на параллельное исполнение.

В результате выполнения этой команды вы увидите нечто вроде этого:

```
[yuri@server sbin]# mpirun -hostfile mpi.host -np 4 hostname
node1.cluster.org
node2.cluster.org
node3.cluster.org
node4.cluster.org
[yuri@server sbin]#
```

Естественно, названия узлов у вас будут другие, в зависимости от того, как названы ваши машины.

Компиляция и выполнение

Процесс компиляции и выполнения параллельных программ, написанных с использованием MPI, примерно одинаков в MPICH, LAM/MPI и OpenMPI. Оба пакета содержат в себе специализированные скрипты (wrappers) облегчающие вызов компиляторов. Для языка FORTRAN такой скрипт называется mpif77. Компиляция исходного текста программы, написанной на FORTRANe выполняется следующим образом:

mpif77 myprog.f -o myprog

Здесь *myprog.f* - исходный текст программы, *myprog* - исполняемый модуль, полученный в результате компиляции.

Следующий этап работы с кластером - запуск параллельных программ на исполнение. В обоих версиях MPI, которые мы рассматриваем, запуск программы происходит с помощью команды *mpirun*:

MPICH, OpenMPI
mpirun -np 4 -machinefile ~/machines /tmp/prog1/myprog

LAM/MPI
mpirun -np 4 /tmp/prog1/myprog

Параметр -np задает количество процессоров кластера, на которых будет запущена программа. Для MPICH используется дополнительный параметр -machinefile, который указывает на файл (~/machines), содержащий список машин клсатера. Естественно, здесь приведены простейшие варианты запуска. Команда mpirun имеет гораздо больше параметров, позволяющих оператору кластера произвольно сложным образом формировать задание на счет.

Следует отметить, что запускаемая программа должна быть скопирована с консоли кластера на все узловые машины в одно и тоже место. Так, если мы в передыдущих примерах запускали программу /tmp/prog1/myprog, то на узлах кластера по этому же пути должен существовать исполняемый файл с тем же названием, хотя и не обязательно тот же самый.

Общая организация MPI

Если сформулировать коротко, MPI - это библиотека функций, обеспечивающая взаимодействие параллельных процессов с помощью механизма передачи сообщений. Это достаточно объемная и сложная библиотека, состоящая примерно из 130 функций, в число которых входят:

- функции инициализации и закрытия MPI процессов;
- функции, реализующие коммуникационные операции типа точка-точка;
- функции, реализующие коллективные операции;

- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

Набор функций библиотеки MPI далеко выходит за рамки набора функций, минимально необходимого для поддержки механизма передачи сообщений, в вычислительных задачах малой и средней сложности. Однако сложность этой библиотеки не должна пугать пользователей, поскольку, в конечном итоге, все это множество функций предназначено для облегчения разработки эффективных параллельных программ. В конце концов, пользователю принадлежит право самому решать, какие средства из предоставляемого арсенала использовать, а какие нет. В принципе, любая параллельная программа может быть написана с использованием всего 6 MPI функций, а достаточно полную и удобную среду программирования составляет набор из 24 функций.

Каждая из MPI функций характеризуется способом выполнения:

1. Локальная функция - выполняется внутри вызывающего процесса. Ее завершение не требует коммуникаций.
2. Нелокальная функция - для ее завершения требуется выполнение MPI-процедуры другим процессом.
3. Глобальная функция - процедуру должны выполнять все процессы группы. Несоблюдение этого условия может приводить к зависанию задачи.
4. Блокирующая функция - возврат управления из процедуры гарантирует возможность повторного использования параметров, участвующих в вызове. Никаких изменений в состоянии процесса, вызвавшего блокирующий запрос, до выхода из процедуры не может происходить.
5. Неблокирующая функция - возврат из процедуры происходит немедленно, без ожидания окончания операции и до того, как будет разрешено повторное использование параметров, участвующих в запросе. Завершение неблокирующих операций осуществляется специальными функциями.

В языке FORTRAN большинство MPI-процедур являются подпрограммами (вызываются с помощью оператора CALL), а код ошибки возвращают через дополнительный последний параметр процедуры. Несколько процедур, оформленных в виде функций, код ошибки не возвращают. Не требуется строгого соблюдения регистра символов в именах подпрограмм и именованных констант. Массивы индексируются с 1. Ниже приведено соответствие предопределенных в MPI типов стандартным типам языка FORTRAN.

Тип MPI	**Тип языка FORTRAN**
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Изучение MPI начнем с рассмотрения базового набора из 6 функций, образующих минимально полный набор, достаточный для написания простейших программ.

Базовые функции MPI

Любая прикладная MPI-программа должна начинаться с вызова функции инициализации

MPI: функции MPI_Init. В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором MPI_COMM_WORLD. Эта область связи объединяет все процессы-приложения. Процессы в группе упорядочены и пронумерованы от 0 до groupsize-1, где groupsize равно числу процессов в группе. Кроме этого, создается предопределенный коммуникатор MPI_COMM_SELF, описывающий свою область связи для каждого отдельного процесса.

Синтаксис **функции инициализации MPI_Init**:

- MPI_INIT(IERROR)
- INTEGER IERROR

В программах на языке FORTRAN параметр IERROR является выходным и возвращает код ошибки.

Функция завершения MPI программ MPI_Finalize

- MPI_FINALIZE(IERROR)
- INTEGER IERROR

Функция закрывает все MPI-процессы и ликвидирует все области связи.

Функция определения числа процессов в области связи MPI_Comm_size

- MPI_COMM_SIZE(COMM, SIZE, IERROR)
- INTEGER COMM, SIZE, IERROR

IN comm - коммуникатор;
OUT size - число процессов в области связи коммуникатора comm.
Функция возвращает количество процессов в области связи коммуникатора comm.

До создания явным образом групп и связанных с ними коммуникаторов единственно возможными значениями параметра COMM являются MPI_COMM_WORLD и MPI_COMM_SELF, которые создаются автоматически при инициализации MPI. Подпрограмма является локальной.

Функция определения номера процесса MPI_Comm_rank

- MPI_COMM_RANK(COMM, RANK, IERROR)
- INTEGER COMM, RANK, IERROR

IN comm - коммуникатор;
OUT rank - номер процесса, вызвавщего функцию.
Функция возвращает номер процесса, вызвавшего эту функцию. Номера процессов лежат в диапазоне 0..size-1 (значение size может быть определено с помощью предыдущей функции). Подпрограмма является локальной.

В минимальный набор следует включить также две функции передачи и приема сообщений.

Функция передачи сообщения MPI_Send

- MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
- <type> BUF(*)
- INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

IN buf - адрес начала расположения пересылаемых данных;
IN count - число пересылаемых элементов;
IN datatype - тип посылаемых элементов;

IN dest - номер процесса-получателя в группе, связанной с коммуникатором comm;
IN tag - идентификатор сообщения (аналог типа сообщения функций nread и nwrite PSE nCUBE2);
IN comm - коммуникатор области связи.
Функция выполняет посылку count элементов типа datatype сообщения с идентификатором tag процессу dest в области связи коммуникатора comm. Переменная buf - это, как правило, массив или скалярная переменная. В последнем случае значение count = 1.

Функция приема сообщения MPI_Recv

- MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,
- STATUS, IERROR)
- <type> BUF(*)
- INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
- STATUS(MPI_STATUS_SIZE), IERROR

OUT buf - адрес начала расположения принимаемого сообщения;
IN count - максимальное число принимаемых элементов;
IN datatype - тип элементов принимаемого сообщения;
IN source - номер процесса-отправителя;
IN tag - идентификатор сообщения;
IN comm - коммуникатор области связи;
OUT status - атрибуты принятого сообщения.
Функция выполняет прием count элементов типа datatype сообщения с идентификатором tag от процесса source в области связи коммуникатора comm.

Более детально об операциях обмена сообщениями мы поговорим в следующем разделе, а в заключение этого раздела рассмотрим функцию, которая не входит в очерченный нами минимум, но которая важна для разработки эффективных программ. Речь идет о функции получения отсчета времени - таймере. С одной стороны, такие функции имеются в составе всех операционных систем, но, с другой стороны, существует полнейший произвол в их реализации. Опыт работы с различными операционными системами показывает, что при переносе приложений с одной платформы на другую первое (а иногда и единственное), что приходится переделывать - это обращения к функциям учета времени. Поэтому разработчики MPI, добиваясь полной независимости приложений от операционной среды, определили и свои функции отсчета времени.

Функция отсчета времени (таймер) MPI_Wtime

- DOUBLE PRECISION MPI_WTIME()

Функция возвращает астрономическое время в секундах, прошедшее с некоторого момента в прошлом (точки отсчета). Гарантируется, что эта точка отсчета не будет изменена в течение жизни процесса. Для хронометража участка программы вызов функции делается в начале и конце участка и определяется разница между показаниями таймера.

Коммуникационные операции типа точка-точка

К операциям этого типа относятся две представленные в предыдущем разделе коммуникационные процедуры. В коммуникационных операциях типа точка-точка всегда участвуют не более двух процессов: передающий и принимающий. В MPI имеется множество функций, реализующих такой тип обменов. Многообразие объясняется возможностью организации таких обменов множеством способов. Описанные в предыдущем разделе функции реализуют *стандартный режим с блокировкой.*

Блокирующие функции подразумевают выход из них только после полного окончания

операции, т.е. вызывающий процесс блокируется, пока операция передачи или приема данных не будет завершена.

Неблокирующие функции подразумевают совмещение операций обмена с другими операциями, поэтому неблокирующие функции передачи и приема по сути дела являются функциями инициализации соответствующих операций. Для опроса завершенности операции (и завершения) вводятся дополнительные функции.

Как для блокирующих, так и неблокирующих операций MPI поддерживает четыре режима выполнения. Эти режимы касаются только функций передачи данных, поэтому для блокирующих и неблокирующих операций имеется по четыре функции посылки сообщения. Ниже перечислены имена базовых коммуникационных функций типа точка-точка, имеющихся в библиотеке MPI.

Режимы выполнения	**С блокировкой**	**Без блокировки**
Стандартная посылка	MPI_Send	MPI_ISend
Синхронная посылка	MPI_SSend	MPI_ISSend
Буферизованная посылка	MPI_BSend	MPI_IBSend
Согласованная посылка	MPI_RSend	MPI_IRsend
Прием информации	MPI_Recv	MPI_IRecv

Из таблицы хорошо виден принцип формирования имен функций. К именам базовых функций Send/Recv добавляются различные префиксы.

- Префикс S (synchronous) -
- означает синхронный режим передачи данных. Операция передачи данных заканчивается только тогда, когда заканчивается прием данных. Функция нелокальная.
- Префикс B (buffered) -
- означает буферизованный режим передачи данных. В адресном пространстве передающего процесса с помощью специальной функции создается буфер обмена, который используется в операциях обмена. Операция посылки заканчивается, когда данные помещены в этот буфер. Функция имеет локальный характер.
- Префикс R (ready) -
- согласованный или подготовленный режим передачи данных. Операция передачи данных начинается только тогда, когда принимающий процессор выставил признак готовности приема данных, инициировав операцию приема. Функция нелокальная.
- Префикс I (immediate) -
- относится к неблокирующим операциям.

Все функции передачи и приема сообщений могут использоваться в любой комбинации друг с другом. Функции передачи, находящиеся в одном столбце, имеют совершенно одинаковый синтаксис и отличаются только внутренней реализацией. Поэтому в дальнейшем будем рассматривать только стандартный режим, который в обязательном порядке поддерживают все реализации MPI.

В стандартном режиме выполнение операции обмена включает три этапа:

1. Передающая сторона формирует пакет сообщения. Этот пакет передается отправителем в системный буфер, и на этом функция посылки сообщения заканчивается.
2. Сообщение системными средствами передается адресату.
3. Принимающий процессор извлекает сообщение из системного буфера, когда у

него появится потребность в этих данных. Содержательная часть сообщения помещается в адресное пространство принимающего процесса (параметр buf), а служебная - в параметр status.

Приведем пример программы обмена данными, выполняемой на двух процессорах с использование блокирующих функций.

```
C определяем свой идентификатор процесса
      CALL MPI_COMM_RANK(comm, rank, ierr)
      IF (rank.EQ.0) THEN
C если это первый процесс, то сначала посылаем данные, ждем завершения операции,
потом принимаем
          CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
          CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
      ELSE IF (rank.EQ.1) THEN
C если это второй процесс, то сначала принимаем данные, ждем завершения
операции, потом отсылаем
          CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
          CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
      END IF
```

В ситуациях, когда требуется выполнить взаимный обмен данными между процессами, безопаснее использовать совмещенную операцию MPI_Sendrecv.

```
    MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,
                 RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)
    <type> SENDBUF(*), RECVBUF(*)
    INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT, RECVTYPE,
            SOURCE, RECV TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR
```

Тип	Параметр	Описание
IN	sendbuf	адрес начала расположения посылаемого сообщения
IN	sendcount	число посылаемых элементов
IN	sendtype	тип посылаемых элементов
IN	dest	номер процесса-получателя
IN	sendtag	идентификатор посылаемого сообщения
OUT	recvbuf	адрес начала расположения принимаемого сообщения
IN	recvcount	максимальное число принимаемых элементов
IN	recvtype	тип элементов принимаемого сообщения
IN	source	номер процесса-отправителя
IN	recvtag	идентификатор принимаемого сообщения
IN	comm	коммуникатор области связи
OUT	status	атрибуты принятого сообщения

Функция MPI_Sendrecv совмещает выполнение операций передачи и приема. Обе операции используют один и тот же коммуникатор, но идентификаторы сообщений могут различаться. Расположение в адресном пространстве процесса принимаемых и передаваемых данных не должно пересекаться. Пересылаемые данные могут быть различного типа и иметь разную длину.

Неблокирующие коммуникационные операции

Использование неблокирующих коммуникационных операций повышает безопасность с точки зрения возникновения тупиковых ситуаций, а также может увеличить скорость работы программы за счет совмещения выполнения вычислительных и коммуникационных операций. Эти задачи решаются разделением коммуникационных операций на две стадии: инициирование операции и проверку завершения операции.

Функция передачи сообщения без блокировки MPI_Isend

```
MPI_ISEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Тип	Параметр	Описание
IN	buf	адрес начала расположения передаваемых данных
IN	count	число посылаемых элементов
IN	datatype	тип посылаемых элементов
IN	dest	номер процесса-получателя
IN	tag	идентификатор сообщения
IN	comm	коммуникатор
OUT	request	"запрос обмена"

Возврат из подпрограммы происходит немедленно (immediate), без ожидания окончания передачи данных. Этим объясняется префикс I в именах функций. Поэтому переменную buf повторно использовать нельзя до тех пор, пока не будет погашен "запрос обмена". Это можно сделать с помощью подпрограмм MPI_Wait или MPI_Test, передав им параметр request.

Функция приема сообщения без блокировки MPI_Irecv

```
MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR
```

Тип	Параметр	Описание
OUT	buf	адрес для принимаемых данных
IN	count	максимальное число принимаемых элементов
IN	datatype	тип элементов принимаемого сообщения
IN	source	номер процесса-отправителя
IN	tag	идентификатор сообщения
IN	comm	коммуникатор
OUT	request	"запрос обмена"

Возврат из подпрограммы происходит немедленно, без ожидания окончания приема данных. Определить момент окончания приема можно с помощью подпрограмм MPI_Wait или MPI_Test с соответствующим параметром request.

Функция ожидания завершения неблокирующей операции MPI_Wait

```
MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

INOUT       request         - "запрос обмена";
OUT         status  - атрибуты сообщения.
```

Это нелокальная блокирующая операция. Возврат происходит после завершения операции, связанной с запросом request. В параметре status возвращается информация о законченной операции.

Функция проверки завершения неблокирующей операции MPI_Test

```
MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

INOUT       request         -       "запрос обмена";
OUT         flag    -       признак завершенности проверяемой операции;
OUT         status  -       атрибуты сообщения, если операция завершилась.
```

Это локальная неблокирующая операция. Если связанная с запросом request операция

завершена, возвращается flag = true, а status содержит информацию о завершенной операции. Если проверяемая операция не завершена, возвращается flag = false, а значение status в этом случае не определено.

Рассмотрим пример использования неблокирующих операций и функции MPI_Wait.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
   CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
    **** Выполнение вычислений во время передачи сообщения ****
   CALL MPI_WAIT(request, status, ierr)
ELSE
   CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
    **** Выполнение вычислений во время приема сообщения ****
   CALL MPI_WAIT(request, status, ierr)
END IF
```

И, напоследок, опишем структуру переменной *status*.

Назначение полей переменной status:

Поля status	**Обращение к полю из FORTRANa**
Процесс-отправитель	status(MPI_SOURCE)
Идентификатор сообщения	status(MPI_TAG)
Код ошибки	status(MPI_ERROR)

Коллективные операции

Набор операций типа точка-точка является достаточным для программирования любых алгоритмов, однако MPI вряд ли бы завоевал такую популярность, если бы ограничивался только этим набором коммуникационных операций. Одной из наиболее привлекательных сторон MPI является наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий. Например, часто возникает потребность разослать некоторую переменную или массив из одного процессора всем остальным. Каждый программист может написать такую процедуру с использованием операций Send/Recv, однако гораздо удобнее воспользоваться коллективной операцией MPI_Bcast. Причем гарантировано, что эта операция будет выполняться гораздо эффективнее, поскольку MPI-функция реализована с использованием внутренних возможностей коммуникационной среды.

Главное отличие коллективных операций от операций типа точка-точка состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Рассмотрим основные функции, определенные для коллективных операций.

Функция синхронизации процессов MPI_Barrier блокирует работу вызвавшего ее процесса до тех пор, пока все другие процессы группы также не вызовут эту функцию. Завершение работы этой функции возможно только всеми процессами одновременно (все процессы "преодолевают барьер" одновременно).

```
MPI_BARRIER(COMM, IERROR)
INTEGER COMM, IERROR

IN comm - коммуникатор.
```

Синхронизация с помощью барьеров используется, например, для завершения всеми процессами некоторого этапа решения задачи, результаты которого будут использоваться на следующем этапе. Использование барьера гарантирует, что ни один из процессов не приступит раньше времени к выполнению следующего этапа, пока результат работы предыдущего не будет окончательно сформирован. Неявную синхронизацию процессов выполняет любая коллективная функция.

Широковещательная рассылка данных выполняется с помощью функции MPI_Bcast. Процесс с номером root рассылает сообщение из своего буфера передачи всем процессам области связи коммуникатора comm.

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

Тип	Параметр	Описание
INOUT	buffer	адрес начала расположения в памяти рассылаемых данных
IN	count	число посылаемых элементов
IN	datatype	тип посылаемых элементов
IN	root	номер процесса-отправителя
IN	comm	коммуникатор

После завершения подпрограммы каждый процесс в области связи коммуникатора comm, включая и самого отправителя, получит копию сообщения от процесса-отправителя root.

Пример использования функции MPI_Bcast.

```
...
IF ( MYID .EQ. 0 ) THEN
PRINT *, 'ВВЕДИТЕ ПАРАМЕТР N : '
READ *, N
END IF
CALL MPI_BCAST(N, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, IERR)
```

Функция MPI_Gather производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root. Длина блоков предполагается одинаковой. Объединение происходит в порядке увеличения номеров процессов-отправителей. То есть данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf процесса root. Длина массива, в который собираются данные, должна быть достаточной для их размещения.

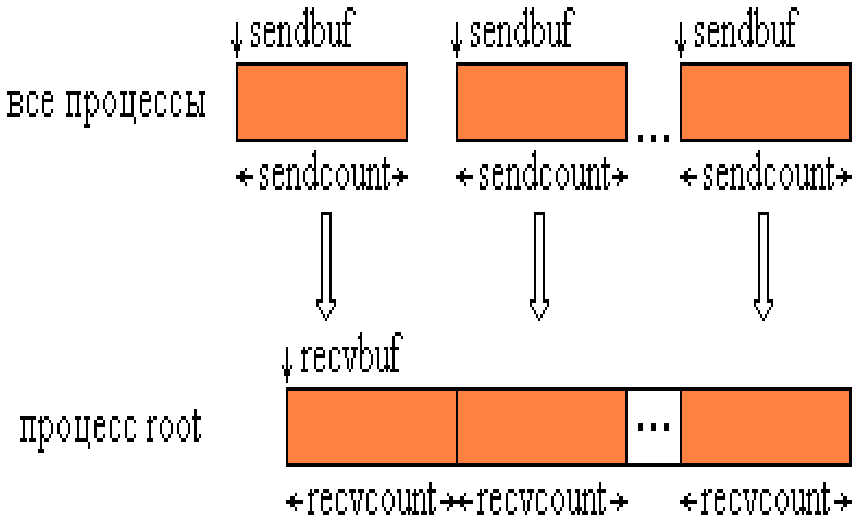
```
MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
           RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR
```

Тип	Параметр	Описание
IN	sendbuf	адрес начала размещения посылаемых данных
IN	sendcount	число посылаемых элементов
IN	sendtype	тип посылаемых элементов
OUT	recvbuf	адрес начала буфера приема (используется только в процессе-получателе root)
IN	recvcount	число элементов, получаемых от каждого процесса (используется только в процессе-получателе root)
IN	recvtype	тип получаемых элементов
IN	root	номер процесса-получателя
IN	comm	коммуникатор

Тип посылаемых элементов sendtype должен совпадать с типом recvtype получаемых

элементов, а число sendcount должно равняться числу recvcount. То есть, recvcount в вызове из процесса root - это число собираемых от каждого процесса элементов, а не общее количество собранных элементов.

Графическая интерпретация операции Gather представлена на следующем рисунке:

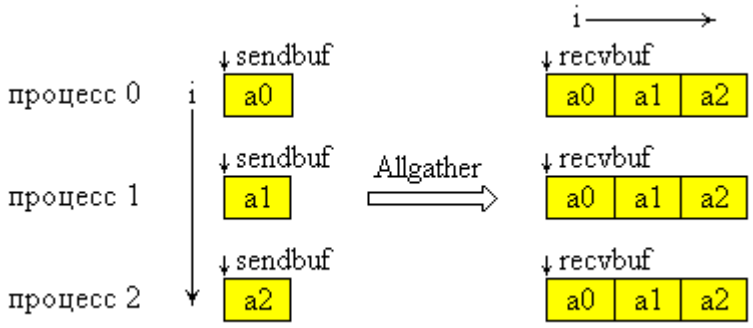


Функция MPI_Allgather выполняется так же, как MPI_Gather, но получателями являются все процессы группы. Данные, посланные процессом i из своего буфера sendbuf, помещаются в i-ю порцию буфера recvbuf каждого процесса. После завершения операции содержимое буферов приема recvbuf у всех процессов одинаково.

```
MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
              RECVCOUNT, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM, IERROR
```

Тип	Параметр	Описание
IN	sendbuf	адрес начала буфера посылки
IN	sendcount	число посылаемых элементов
IN	sendtype	тип посылаемых элементов
OUT	recvbuf	адрес начала буфера приема
IN	recvcount	число элементов, получаемых от каждого процесса
IN	recvtype	тип получаемых элементов
IN	comm	коммуникатор

Графическая интерпретация операции Allgater представлена на следующем рисунке. На этой схеме ось Y образуют процессы группы, а ось X блоки данных.



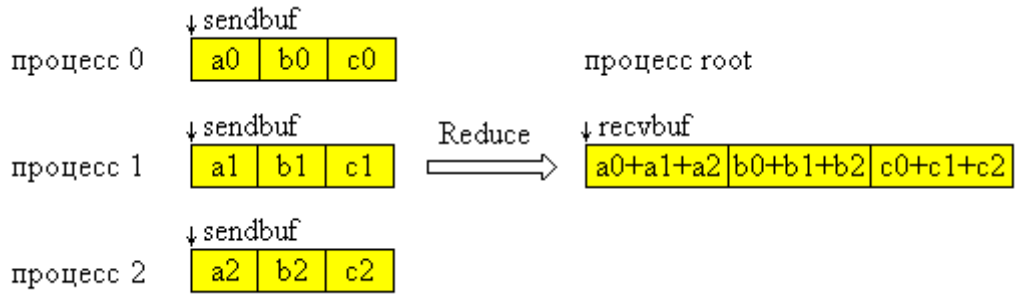
Функция MPI_Reduce выполняется следующим образом. Операция глобальной редукции, указанная параметром op, выполняется над первыми элементами входного буфера, и

результат посылается в первый элемент буфера приема процесса root. Затем то же самое делается для вторых элементов буфера и т.д.

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT,
           COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

Тип	Параметр	Описание
IN	sendbuf	адрес начала входного буфера
OUT	recvbuf	адрес начала буфера результатов (используется только в процессе-получателе root)
IN	count	число элементов во входном буфере
IN	datatype	тип элементов во входном буфере
IN	op	операция, по которой выполняется редукция
IN	root	номер процесса-получателя результата операции
IN	comm	коммуникатор

Ниже на рисунке представлена графическая интерпретация операции Reduce. На данной схеме операция "+" означает любую допустимую операцию редукции.



Менеджер ресурсов Torque

Несмотря на то, что средства MPI сами по себе позволяют осуществлять запуск параллельных задач, обычно для этих целей используются различные менеджеры ресурсов. Одним из таких менеджеров является система Torque - один из наиболее популярных и простых в использовании менеджеров.

Система управления заданиями Torque предназначена для управления запуском задач на многопроцессорных вычислительных установках (в том числе кластерных). Она позволяет автоматически распределять вычислительные ресурсы между задачами, управлять порядком их запуска, временем работы, получать информацию о состоянии очередей. При невозможности запуска задач немедленно, они ставятся в очередь и ожидают, пока не освободятся нужные ресурсы.

Torque главным образом используется на многопроцессорных вычислительных установках. Объединение ресурсов в вычислительных установках обычно уменьшает необходимость в постоянном управлении ресурсами для пользователей. Настроенная однажды правильно вычислительная установка абстрагируется от многих деталей, связанных с запуском и управлением заданиями. Пользователю обычно надо установить в параметрах лишь минимальные требования к задаче, и ему нет необходимости знать даже имена вычислительных узлов, на которых задача выполняется.

В том случае, когда вы являетесь единоличным пользователем кластера, особой нужды в менеджере ресурсов нет. Эффективно управлять собственной задачей вы можете самостоятельно. Однако, если кластером пользуются несколько человек, то неизбежно

возникают задачи административного характера - кто, когда и на сколько времени может занимать ресурсы кластера. Кроме того, если у вас нет выделенного кластера, но вместо этого для решения параллельных задач используется компьютерный класс, который в дневное время предназначен например для обеспечения учебного процесса, то проблема остановки "тяжеловесных" параллельных задач и освобождения ресурсов в 8:00 утра, может быть легко решена с помощью Torque.

Менеджер Torque может быть загружен с официального сайта проекта.

Система Torque состоит из нескольких демонов, выполняющих различные функции по управелению потоком заданий. Вычислительная установка обязана иметь главный узел (консоль кластера), на котором запущен демон **pbs_server**. Это основной демон - менеджер, собирающий информацию о структуре кластера и запущенных заданиях. В зависимости от необходимости или параметров системы главный узел может быть предназначен только для этого или же также исполнять роль других компонент системы. Например, он может быть так же вычислительным узлом кластера.

Роль вычислительных узлов - выполнять поставленные задачи. На каждом из них работает демон **pbs_mom** для того, чтобы начинать, прекращать и управлять поставленными в очередь задачами. Это единственный демон, который должен быть запущен на вычислительном узле кластера.

И наконец, демон **pbs_sched**. Этот демон занимается собственно планированием запуска и остановки задач. Он должен быть запущен на главном компьютере кластера.

Torque: установка

Установка системы Torque может быть выполнена путем компиляции из исходников, которые доступны для загрузки с официального сайта проекта. В операционной системе Ubuntu Linux Torque имеется в репозиториях пакетов и может быть установлен в систему обычным для Ubuntu образом - с помощью команды apt-get или менеджера пакетов Synaptic.

Установка из исходников системы Torque на главном компьютере кластера достаточно проста. Для этого скачанный архив программы должен быть развернут в какой-либо каталог, после чего, внутри этого каталога необходимо выполнить обычную для данного действия последовательность команд:

```
./configure --prefix=/usr
make
sudo make install
```

После установки Torque требуется перекомпиляция системы MPI для того, чтобы она знала о существовании менеджера ресурсов и могла взаимодействовать с ним. Компиляция MPI осуществляется точно так же, как это было ранее описано на этом сайте.

Настройка начальной очереди заданий будет нашим следующим шагом. Сделать это можно выполнив (в каталоге с исходниками) команду **./torque.setup <ИмяПользователя>**. Где <ИмяПользователя> - это имя пользователя, который будет администратором Torque, например root:

```
./torque.setup root
```

Установка менеджера ресурсов на вычислительных узлах кластера не требует полной установки системы Torque. Вместо этого на главном компьютере необходимо создать саморазворачивающийся архив системы с помощью команды

```
make packages
```

запущеной в каталоге с исходниками Torque. В результате будет создано несколько

исполняемых скриптов. Нас будет интересовать скрипт **torque-package-mom-linux-i686.sh**. Этот скрипт мы должны скопировать на все вычислительные узлы кластера и запустить его на исполнение на этих узлах:

./torque-package-mom-linux-i686.sh --install

Конфигурация Torque на вычислительных узлах кластера в минимальном варианте заключается в указании доверенного сервера, который будет иметь право управлять заданиями на узле. Выполнить это можно записав в файле /var/spool/torque/server_name имя хоста вашего сервера (консоли кластера). Обычно это не требуется, поскольку делается автоматически. Единственно, что я рекомендую, так это проверить, правильно ли резольвится это имя. Посмотрите, что записано в этом файле (например там будет записано "supergate") и пропингуйте это имя. Если что-то не так, то пропишите это имя в файле /etc/hosts.

Настройка Torque-сервера заключается в задании списка вычислительных узлов кластера в файле /var/spool/torque/server_priv/nodes (который надо создать). В этом файле должны быть перечислены имена всех вычислительных узлов, один узел - одна строчка. После этого необходимо остановить сервер, чтобы при следующем его запуске список узлов вступил в силу. Для этого запускаем команду **sudo qterm** без параметров.

Проверка системы. Для проверки работоспособности системы на всех вычислительных узлах запускаем демон **pbs_mom**. Далее на главном компьютере запускаем сервер Torque - демон **pbs_server**. Проверка узлов осуществляется командой **pbsnodes -a**, которая выдасть на консоль нечто вроде этого:

```
mpiuser@supergate:~$ pbsnodes -a
supergate
     state = free
     np = 1
     ntype = cluster
     status = opsys=linux,uname=Linux supergate 2.6.28-14-generic #47-Ubuntu SMP Sat Jul 25 00:28:35
     UTC 2009 i686,sessions=2524 13213 9091 9136 9353 9473 9521 8948 9868 9899 10149 10692 10101
13099
     13242 13259 13268 13277 13288 13341 13359 13383 28122 32075 32076 32077 32078 32079 32080 32081
     32082 32083 32084 32085 32086 32087 32091
32093,nsessions=38,nusers=8,idletime=380,totmem=3123024kb,
     availmem=2583100kb,physmem=1018552kb,ncpus=2,loadave=0.00,netload=2253226200,state=free,jobs=,v
arattr=,
     rectime=1250488593

LENA
     state = free
     np = 1
     ntype = cluster
     status = opsys=linux,uname=Linux Lena2 2.6.28-14-generic #47-Ubuntu SMP Sat Jul 25 00:28:35
     UTC 2009 i686,sessions=14835 2176 18833 2603 2455 3053 14938 14949 15015 15024 15034 15103
     15126 15170 15181
15603,nsessions=16,nusers=4,idletime=1437,totmem=3026692kb,availmem=2362384kb,
     physmem=1026640kb,ncpus=2,loadave=0.23,netload=1604969560,state=free,jobs=,varattr=,rectime=125
0488595

mpiuser@supergate:~$
```

Состояние узлов **state = free** говорит о том, что узлы работоспособны и готовы к принятию заданий.

Проверка очереди заданий может быть осуществлена путем добавления в очередь простой задачи:

echo "sleep 30" | qsub

Задача эта, как видите, ничего не делает, кроме как ждет 30 секунд, по завершении которых благополучно умирает. Проверим состояние очереди командой **qstat**, которая покажет нам что-то вроде этого:

```
mpiuser@supergate:~$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
0.mail                    STDIN            mpiuser                0 Q batch
dima@supergate:~$
```

Статус задания (предпоследняя колока) имеет значение **"Q"**, что говорит нам о том, что задание находится в очереди и ждет, когда ей будет разрешено начать работу. Ждать оно будет долго, до тех пор пока планировщик не решит, что пришло ее время. Однако планировщик в системе отсутствует. Чтобы он присуствовал, запустим его командой **sudo pbs_sched**. Через какое-то время мы увидим, что задание начало работать:

```
mpiuser@supergate:~$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
0.mail                    STDIN            mpiuser                0 R batch
dima@supergate:~$
```

То есть статус задания измеился с **"Q"** (Queued) на **"R"** (Run), а еще немного погодя, статус изменится на **"C"** (Completed):

```
mpiuser@supergate:~$ qstat
Job id                    Name             User            Time Use S Queue
------------------------- ---------------- --------------- -------- - -----
0.mail                    STDIN            mpiuser         00:00:00 C batch
dima@supergate:~$
```

Если все прошло именно так, как описано, то значит система Torque у нас работает и мы можем ее использовать для запуска наших параллельных задач.

Torque: использование

Каждая параллельная задача, которую мы собираемся ставить в очередь на исполнение должна быть оформлена в виде пакета. Пакет представляет собой набор параметров запуска и инструкции что конкретно мы хотим запустить. Здесь мы рассмотрим самые необходимые параметры и команды, которые потребуются в большинстве случаев.

Предположим, что мы собираемся запускать тестовую программу **./flops**, которую мы упоминали, когда рассматривали установку Ubuntu кластера. Для запуска мы хотим, чтобы выполнялись слкдующие условия:

1. Использовались три вычислительных узла.
2. На каждом из которых мы займем по одному доступному процессору.
3. Задача должна выполнятся не более 10 часов (по истечении которых она должна быть снята со счета).
4. На вычислительных узлах должно быть доступно не менее 100Мб оперативной памяти.
5. О событиях запуска задачи и ее завершения (с ошибкой или без) мы должны быть уведомлены сообщением, направленным на нашу электронную почту.

Для описания такой задачи создадим следующий скрипт (назовем его flops.pbs):

```
#!/bin/sh
#
#PBS -l nodes=supergate:ppn=1+2:ppn=1
#PBS -N Flops_TEST
#PBS -m abe
#PBS -M yuri@linux-ekb.info
#PBS -l pmem=100mb
#PBS -l pcput=10:00:00
cd /home/mpiuser/mpi
mpirun ./flops
```

Количество задействованных процессоров мы описываем параметром **nodes=supergate:ppn=1+2:ppn=1**. Причем указываем (это важно!) что первым узлом нашей параллельной задачи будет хост **supergate**. Именно так мы назвали нашу консоль кластера. Кроме него задаче будет выделено еще 2 узла. Для каждого из этих трех узлом мы запрашиваем по одному процессору (ppn=1).

Параметр **-N Flops_TEST** - это просто название нашей задачи. Так она будет отображена в очереди задач.

Параметры **-m abe** и **-M yuri@linux-ekb.info** указывают, какие должны быть уведомления и куда их посылать.

Последние два параметра (**pmem=100mb** и **pcput=10:00:00**) должны быть понятны. Это запрашиваемый размер оперативной памяти и максимальное время исполнения программы.

Остальные строчки скрипта - это собственно задача, которая должна быть исполнена на кластере: переход в каталог с программой и запуск ее на параллельное исполнение скриптом OpenMPI.

Запсук задания, точнее размещение его в очереди на исполнение, осуществляется командой **qsub**, единственным параметром которой будет имя нашего скрипта:

qsub flops.pbs

Ранее мы рассматривали способ запуска параллельной программы в среде OpenMPI, и там в качестве одного из параметров команды запуска указывался файл со списком вычислительных узлов кластера. При использовании менеджера ресурсов **Torque** этот параметр не нужен. Конфигурацию кластера и список узлов диспетчер OpenMPI берет непосредственно из Torque.

Стандартные потоки вывода (STDOUT, STDERR) после завершения программы вы можете найти в том каталоге, из которога программа была запущена, в файлах с именами **Flops_TEST.o45** и **Flops_TEST.e45**. Цифры - это номер очереди для нашей задачи, а название перед расширением - это то имя, которое мы указали в нашем скрипте запуска.

Практика параллельного программирования

Спецификация MPI (Message Passing Interface - интерфейс передачи сообщений) предлагает модель программирования, в которой программа порождает несколько процессов, взаимодействующих между собой с помощью обращения к подпрограммам передачи и приема сообщений. Ее реализации представляют собой библиотеки подпрограмм, которые могут использоваться в программах на языках программирования C/C++ и Fortran.

Написание подобной программы заставляет программиста полностью перестроить привычный способ мышления. По сравнению с "обычной" последовательной программой появляется новое измерение - появляется необходимость уметь видеть, какие части программы не зависят друг от друга в плане исполнения кода, но требуют взаимодействия на уровне данных и синхронизации во времени.

Звучит ужаснее, чем есть на самом деле. Тем более, что преимущества, получаемые в результате такой перестройки, с лихвой окупают неизбежные трудности.

Далее мы на конкретных примерах рассмотрим технологию написания параллельных программ.

Структура параллельной программы

Структура параллельной программы в большой степени зависит от численных методов,

используемых для ее решения, выбора языка программирования, компиляторов, а так же от самой задачи. Однако можно выделить независящие от этих факторов структурные особенности. Любая программа, выполняемая на кластерном компьютере, должна выполнять определенную последовательность действий, которую (последовательность) можно представить в виде виде отдельных алгоритмических блоков. В общем виде эта последовательность может выглядеть примерно таким вот образом:

```
1.              program test
2.              <инициализация параллельной среды>
3.              N=<номер текущего процессора>
4.              <загрузка массивов данных для процеса N>
5.              while(<задача не решена>) do
6.                      <получение и передача граничных данных>
7.                      <выполнение очередной итерации>
8.                      if(<контрольная точка>) then
9.                              <сохранение данных на диск>
10.             done
11.             <деинициализация параллельной среды>
12.             end
13.             stop
```

Более подробно прокоментируем каждый из шагов программы.

Строка N1. Начало программы.

Строка N2. Действие, производимое на этом шаге вполне понятно. Здесь мы выполняем некоторые операции для подготовки к работе параллельной вычислительной среды. Конкретный программный код этого шага зависит от языка программирования и выбранной параллельной среды (MPICH/MPI/PVM).

Строка N3. Важный шаг. Перед выполнением основного кода мы должны понять, на каком процессоре выполняется данная программа, поскольку от этого зависит, что именно будет делаться дальше. Ведь каждый отдельный процесс параллельной программы выполняет либо собственные специфические действия, либо обрабатывает собственную порцию общего массива данных. Либо специфичным для процесса является и то и другое. В зависимости от выбранного варианта декомпозации исходной задачи.

Строка N4. На этом этапе программа должна подготовить для дальнейшей обработки массивы данных, загрузив в них информацию, специфичную для процесса, номер которого мы получили в предыдущем шаге.

Строка N5. Это начало нашего итерационного процесса. Здесь же мы по каким-то критериям решаем, достигнут ли желаемый результат, и не пора ли завершить программу.

Строка N6. Перед выполнением собственно итерации, мы должны получить данные от соседних процессов для граничных областей локальных массивов.

Строка N7. Собственно обработка массивов данных в соответствии с задачами, возложенными на процесс номер N. То есть очередной шаг итерации.

Строка N8. После завершения очередной порции вычислений мы должны решить, не наступил ли момент сохранить промежуточные результаты на диск. То есть, не достигли ли мы контрольной точки. Как часто это делать - решать программисту. Однако разумным будет сохранять данные не на каждой итерации, а гораздо реже. Так, чтобы время, необходимое для записи на диск, было много меньше, чем длительность итераций между контрольными точками. Таким образом мы снизим отрицательный влияние на быстродействие кластера такой медленной операции, как сохранение данных на винчестер.

Строка N9. Если мы решили, что достигли контрольной точки, то в этом месте программы мы начинаем процедуру сохранения данных. Причем, выполнять запись данных на диск

необходимо (по мере возможности) в **асинхронном** режиме. Действительно, если мы, например, решаем систему ДУ газодинамики, мы имеем (опять же к примеру) два массива, соответствующих двум временным слоям. Предыдущий слой по времени никак не меняется в процессе вычислений. Поэтому сохранять его на диск имеет смысл параллельно вычислениям, производимым над текущим временным слоем. Естественно, не надо забывать проверять, закончился ли процесс записи перед тем, как возникнет необходимость как-либо изменить массив, который мы сохраняем на диск.

Строка N10. Конец итерационного цикла. Ничего более.

Строка N11. В самом конце работы программы мы должны корректно завершить работу параллельной среды. Чем мы и занимаемся на этом шаге. Здесь же мы можем сохранить на диск финальные данные, которые мы мы можем как-то анализировать или использовать в качестве начальных данных при последующем запуске нашей программы.

Простейшие параллельные программы

Рассмотрим классический пример задачи: программу "Hello world". Программа может быть запущена в среде MPI на произвольном количестве процессоров. Каждая копия программы, будучи запущена на своем узле кластера, перчатает приветствие, номер процесса, вызвавшего функцию печати и общее количество процессов.

Текст программы следующий:

```
program hello
include 'mpif.h'
integer rank, size, ierr

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,size,ierr)

print *,"Hello world! I am ",rank," of ",size

call MPI_FINALIZE(ierr)
stop
end
```

Теперь рассмотрим более осмысленную задачу. Возьмем за основу вычисление числа π . Число π будем вычислять как определенный интеграл :

$$\int_0^1 \frac{4}{1+x^2} dx = 4 \cdot \operatorname{arctg}(x)\Big|_0^1 = \pi$$

Согласно правилу прямоугольников интеграл можно заменить суммой:

$$\pi \approx h \cdot \sum_{i=1}^{n} \left(\frac{4}{1+x_i^2} \right) \; ; \; h = \frac{1}{n} \; ; \; x_i = \left(i - \frac{1}{2} \right) \cdot h$$

Теперь посмотрим как этот алгоритм можно реализовать на языке FORTRAN в многопроцессорном параллельном варианте, с использованием системы MPI:

```
      program calc_pi
      include 'mpif.h'
      integer i, n
      double precision w, gsum, sum
      double precision v
      integer np, myid, ierr
      real*8 time, mflops, time1, time2, dsecnd
c Инициализация MPI и определение процессорной конфигурации
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, np, ierr )
c Информацию с клавиатуры считывает 0-й процессор
      if ( myid .eq. 0 ) then
      n = 400000000
      time1 = MPI_Wtime()
      endif
c Рассылка числа точек разбиения всем процессорам
      call MPI_BCAST(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
c Вычисление частичной суммы на процессоре
      w = 1.0 / n
      sum = 0.0d0

c Вычисление процессом своей части суммы
      do i = myid+1, n, np
      v = (i - 0.5d0 ) * w
      v = 4.0d0 / (1.0d0 + v * v)
      sum = sum + v
      end do
c Суммирование частичных сумм с сохранением результата в 0-м
c процессоре
      call MPI_REDUCE(sum, gsum, 1, MPI_DOUBLE_PRECISION,
     $                MPI_SUM, 0, MPI_COMM_WORLD, ierr)
c Печать выходной информации с 0-го процессора
      if (myid .eq. 0) then
      time2 = MPI_Wtime()
      time   = time2 - time1
      mflops = 9 * n / (1000000.0 * time)
      print *, 'pi is approximated with ', gsum *w
      print *, 'time = ', time, ' seconds'
      print *, 'mflops = ', mflops, ' on ', np, ' processors'
      print *, 'mflops = ', mflops/np, ' for one processor'
      endif
c Закрытие MPI
      call MPI_FINALIZE(ierr)
      end
```

Уравнение теплопроводности

Распределение температуры в области D в стационарном случае при отсутствии притока тепла во внутренних точках области удовлтворяет однородному уравнению Лапласа:

$$\Delta\Phi(x,y) = \frac{\partial^2\Phi(x,y)}{\partial x^2} + \frac{\partial^2\Phi(x,y)}{\partial y^2} = 0.$$

Очевидно, что решение стационарной задачи не зависит от коэффициента теплопроводности.

Одним из подходов к решению стационарного уравнения теплопрводности является так называемый метод установления. В этом случае решается нестационарное уравнение теплопроводности с некоторым начальным распределением температуры и стационарными грничными условиями. В решении время устремляется к бесконечнсти, когда решение "устанавливается", то есть перестает меняться с течением времени. Это решение совпадает с решением стационарного уравнения.

Нестационарное уравнение теплопроводности имеет вид

$$\frac{\partial \Phi}{\partial t} - a\Delta\Phi = 0.$$

Коэффициент α теплопроводности влияет только на скорость установления решения, поэтому положим его равным единице.

Не вдаваясь в теорию численных методов (подробности вы можете найти в соответствующей литературе), запишем разностное представление решения этого уравнения.

$$\Phi_{i,j}^{k+1} = \Phi_{i,j}^{k} + \Delta t \left(\frac{\Phi_{i+1,j}^{k} - 2\Phi_{i,j}^{k} + \Phi_{i-1,j}^{k}}{\Delta x^2} + \frac{\Phi_{i,j+1}^{k} - 2\Phi_{i,j}^{k} + \Phi_{i,j-1}^{k}}{\Delta y^2} \right)$$

Здесь Δt - шаг по времени, Δx и Δy - шаги по пространственным координатам X и Y соответственно. Индексы k,i и j - индексы по времени и координатам X и Y соответственно.

В качестве области вычислений мы для простоты берем прямоугольную сетку, значения на границе которой мы задаем некоей функцией

$$\Phi_{L}^{k+1} = f(s).$$

Для нахождения решения этого уравнения мы проводим последовательные итерации на сетке, а в качестве критерия установления решения используем критерий

$$\max_{i,j} \left| \Phi_{i,j}^{k+1} - \Phi_{i,j}^{k} \right| < \varepsilon,$$

где ε - заданное малое положительное число.

При выполнении расчетов необходимо провести исследование устойчивости нашей разностной схемы, а именно проверить условие сходимости:

$$\Delta t \leqslant \frac{1}{2} \left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} \right)^{-1}.$$

В следующей главе мы для начала рассмотрим обычную последовательную программу, реализующую решение этого уравнения.

Последовательное решение уравнения

```
      program diff
      integer xmax,ymax
c     Размер сетки по оси X
      parameter (xmax=4000)
c     Размер сетки по оси Y
      parameter (ymax=6000)
      real*8 f0(xmax,ymax), f1(xmax,ymax)
      real*8 df
      integer x,y,n
c     Задаем начальные значения
      call initial(f0,xmax,ymax)
      call initial(f1,xmax,ymax)
c     Задаем граничные значения
      call boundary(f0,xmax,ymax)
      call boundary(f1,xmax,ymax)
c     Начинаем основной итерационный цикл
      df=0.0
      n=0
    1 continue
c     При смене итераций меняем местами массивы
      if ( ((n/2)*2) .EQ. n ) then
c       Обрабатываем f0 для каждой четной итерации
        call iter(f0,f1,xmax,ymax,df)
      else
c       Обрабатываем f1 для каждой нечетной итерации
        call iter(f1,f0,xmax,ymax,df)
      endif
c     Увеличиваем номер цикла на единицу
      n=n+1
c     Выводим на экран разницу между итерациями
      write(*,*) 'Diff:',df
c     Если разница больше заданной, то решение не найдено
c     и мы идем на следующий цикл
      if (df > 1e-2) goto 1
c     Печатаем количество итераций, потребовавшихся
c     для нахождения решения
      write(*,*) 'Iteration count:', n
      stop
      end

c     Подпрограмма вычисления искомой функции
      subroutine iter(f0,f1,xmax,ymax,df)
      integer xmax,ymax
      real*8 f0(xmax,ymax), f1(xmax,ymax)
      real*8 dt,dx,dy
      real*8 df,dff,df1
      integer x,y,n
      dt=0.01
      dx=0.5
      dy=0.5
      df1=0.0
c     Вычисляем функцию в ячейках сетки
      do y=2,ymax-1
        do x=2,xmax-1
        dff=dt*(
     x    (f0(x+1,y)-2*f0(x,y)+f0(x-1,y))/(dx*dx)
     x    +
     x    (f0(x,y+1)-2*f0(x,y)+f0(x,y-1))/(dy*dy)
     x    )
```

```
        f1(x,y)=f0(x,y)+dff
c     Находм максимальную дельту
        if( df1 < abs(dff) ) df1=abs(dff)
        end do
      end do
      df=df1
      return
      end

c     Подпрограмма задания начальных значений
      subroutine initial(f0,xmax,ymax)
      integer xmax,ymax,x,y
      real*8 f0(xmax,ymax)
c     Задаем начальные значения массива
      do x=1,xmax
        do y=1,ymax
          f0(x,y)=0.0
        end do
      end do
      return
      end

c     Подпрограмма задания граничных условий
      subroutine boundary(f0,xmax,ymax)
      integer rank,xmax,ymax,x,y
      real*8 f0(xmax,ymax)
      common myid,np
c     Задаем граничные значения массива
      do y=1,ymax
        f0(1,y)=0.0
        f0(xmax,y)=0.0
      end do
      do x=1,xmax
        f0(x,1)=sin((x*1.0)/(xmax/2))
 f0(x,ymax)=0.0
      end do
      f0(xmax/2,ymax)=-5.0
      return
      end
```

Как видите, программа достаточно проста. Здесь я ограничусь всего несколькими пояснениями. В качестве разностной сетки мы берем прямоугольник со сторонами 4000 ячеек по оси X и 6000 ячеек по оси Y. Размерность сетки задается в строка 4 и 6.

Далее, в строках 11 - 15 мы задаем начальные и граничные условия посредством вызова подпрограмм **initial** (строка 69) и **boundary** (строка 82) соответственно, заполняющих по некоторому правилу ячейки сетки.

Цикл с 19 по 34 строчку является основным итерационным процессом. В строке 34 принимается решение о продолжении итераций на основе максимальной разницы между значениями искомой функции на текущем и предыдущем слое по времени. Если разница больше некоторого заранее заданного значения, то итерации продолжаются. Само вычисление функции в ячейках оформлено в виде подпрограммы **iter**, код которой начинается со строки 42. Эта же подпрограмма вычисляет максимальное значение Δf.

В самом начале в качестве исходных данных берется массив **f0**, а вычисленные значения функции на следующем шаге по времени заносятся в массив **f1** - строка 23. На следующей итерации за основу берется уже массив **f1**, то есть массивы меняются местами - строка 26. И так далее.

Теперь о вычислении самой функции. Выполнение этого действия выделено в подпрограмму

iter (строка 42). Функция на k+1 слое по времени вычисляется на основе значений этой функции на k-ом слое. Делается это в два этапа. Сначала вычисляются значения внутренних ячеек на основе разностной формулы (строки 55-60), а затем находим максимальную разницу значений функции между слоями (строка 61).

В следующем разделе мы с вами посмотрим, как можно эту программу превратить в параллельную, что для этого надо и какие изменения в исходный код должны быть внесены.

Параллельное решение уравнения

```
      program diff
      include 'mpif.h'
      integer xmax,ymax
      parameter (xmax=4000)
      parameter (ymax=1500)
      real*8 f0(xmax,ymax), f1(xmax,ymax)
      real*8 df,gdf,ldf
      integer x,y,n,myid,np,ierr,ierr1,status(MPI_STATUS_SIZE)
      common myid,np
      double precision tstart, tstop, tdiff1, tdiff2
c     Инициализация MPI и определение процессорной конфигурации
      call MPI_INIT( ierr )
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
      call MPI_COMM_SIZE( MPI_COMM_WORLD, np, ierr )
      if ( myid .eq. 0 ) write(*,*) 'Starting!'
c     Задаем начальные значения
      call initial(f0,xmax,ymax)
      call initial(f1,xmax,ymax)
c     Задаем граничные значения
      call boundary(f0,xmax,ymax)
      call boundary(f1,xmax,ymax)
c     Начинаем основной итерационный цикл
      df=0.0
      n=0
      if ( myid .eq. 0 ) write(*,*) 'Processing...'
    1 continue
      tstart=MPI_WTIME();
c     При смене итераций меняем местами массивы
      if ( ((n/2)*2) .EQ. n ) then
c       Обрабатываем f0 для каждой четной итерации
        call iter(f0,f1,xmax,ymax,df)
      else
c       Обрабатываем f1 для каждой нечетной итерации
        call iter(f1,f0,xmax,ymax,df)
      endif
      ldf=df
c     Увеличиваем номер цикла на единицу
      n=n+1
      tstop=MPI_WTIME();
      tdiff1=tstop-tstart
c     Получаем разницу значений функции на слоях
c     с каждого узла кластера и находим максимальное значение
c     этой разницы для всей разностной сетки
      call MPI_REDUCE(df, gdf, 1, MPI_REAL8,
     x    MPI_MAX, 0, MPI_COMM_WORLD, ierr1)
c     Сообщаем найденную максимальную разницу
c     всем узлам кластера
      if(myid .eq. 0) df=gdf
      CALL MPI_BCAST(df, 1, MPI_REAL8, 0, MPI_COMM_WORLD, ierr)
      tstart=MPI_WTIME();
      tdiff2=tstart-tstop
```

```
c     Выводим на экран разницу между итерациями
      if ( myid .eq. 0 )
     x write(*,*) 'D:',df,'R:',tdiff1,ierr1,'B:',tdiff2
c     Если разница больше заданной, то решение не найдено
c     и мы идем на следующий цикл
      if (df .GT. 0.01) goto 1
c     Печатаем количество итераций, потребовавшихся
c     для нахождения решения
      if ( myid .eq. 0 ) write(*,*) 'Iteration count:', n
c     Заканчиваем работу с MPI
      call MPI_FINALIZE(ierr)
      stop
      end

c     Подпрограмма вычисления искомой функции
      subroutine iter(f0,f1,xmax,ymax,df)
      include 'mpif.h'
      integer xmax,ymax
      real*8 f0(xmax,ymax), f1(xmax,ymax)
      real*8 dt,dx,dy
      real*8 df,dff,df1
      integer x,y,n,myid,np,ierr,status(MPI_STATUS_SIZE)
      common myid,np
      double precision tstart, tstop, tdiff
      dt=0.01
      dx=0.5
      dy=0.5
c     Обмениваемся границами с соседом
c      tstart=MPI_WTIME();
      if ( myid .gt. 0 )
     x call MPI_SENDRECV(
     x f0(1,2), xmax, MPI_REAL8, myid-1, 1,
     x f0(1,1), xmax, MPI_REAL8, myid-1, 1,
     x MPI_COMM_WORLD, status, ierr)
      if ( myid .lt. np-1 )
     x call MPI_SENDRECV(
     x f0(1,ymax-1), xmax, MPI_REAL8, myid+1, 1,
     x f0(1,ymax), xmax, MPI_REAL8, myid+1, 1,
     x MPI_COMM_WORLD, status, ierr)
c      tstart=MPI_WTIME();
c     Вычисляем функцию в ячейках сетки
      df1=0.0
      do y=2,ymax-1
        do x=2,xmax-1
        dff=dt*(
     x    (f0(x+1,y)-2*f0(x,y)+f0(x-1,y))/(dx*dx)
     x    +
     x    (f0(x,y+1)-2*f0(x,y)+f0(x,y-1))/(dy*dy)
     x    )
        f1(x,y)=f0(x,y)+dff
c     Находим максимальную дельту
        if( df1 < abs(dff) ) df1=abs(dff)
        end do
      end do
      df=df1
      call MPI_BARRIER(MPI_COMM_WORLD, ierr)
      return
      end

c     Подпрограмма задания начальных значений
      subroutine initial(f0,xmax,ymax)
      integer rank,xmax,ymax,x,y
```

```
      real*8 f0(xmax,ymax)
      common myid,np
c     Задаем начальные значения массива
      do x=1,xmax
        do y=1,ymax
          f0(x,y)=0.0
        end do
      end do
      return
      end

c     Подпрограмма задания граничных условий
      subroutine boundary(f0,xmax,ymax)
      integer rank,xmax,ymax,x,y
      real*8 f0(xmax,ymax)
      common myid,np
c     Задаем граничные значения массива
c     на границах с X=1 и X=xmax
      do y=1,ymax
        f0(1,y)=0.0
        f0(xmax,y)=0.0
      end do
c     Если мы - первый процесс, то
c     задаем граничные условия массива
c     на границе с Y=1
      if ( myid .eq. 0 ) then
        do x=1,xmax
          f0(x,1)=sin((x*1.0)/(xmax/2))
        end do
      endif
c     Если мы - последний процесс, то
c     задаем граничные условия массива
c     на границе с Y=ymax
      if (myid .eq. np-1 ) f0(xmax/2,ymax)=-5.0
      return
      end
```

Изменения, внесенные в нашу программу для распараллеливания, достаточно просты. Рассмотрим их последовательно.

Первое. Предположим, что мы готовим программу для расчетов на кластере, состоящем из 4х узлов. Поскольку массивы Фортрана хранятся в памяти по столбцам, то есть a(1,1),a(2,1),a(3,1)...a(xmax,1),a(1,2),a(2,2),a(3,2) и т.д., то нашу разностную сетку мы поделим между узлами, разрезав ее вдоль оси X. Другими словами, размерность X массивов останется прежней (4000), а размерность Y мы уменьшим в 4 раза (до 1500), что и было сделано в строке 5.

Второе. Добавим файл с описаниями MPI во все подпрограммы (в том числе и в основную программу), где используются методы MPI. Это сделано в строках 2 и 68.

Третье. Добавим описание некоторых переменных, которые будут использоваться в вызовах MPI - строчки 7,8,9,73,74,113,114,127,129.

Четвертое. В начале программы проинициализируем среду MPI, запросив номер текущего процесса и общее количество процессов - строки 12,13 и 14 соответственно.

Пятое. Все операторы печати, которые имеются в программе мы оставим работать только в корневом процессе. Это было сделано путем проверки равенства значения переменной **myid** нулю: строки 15, 25, 53 и 60.

Шестое. Поскольку максимальная разница между значениями функции в ячейках на разных

временных слоях, необходимая нам для принятия решения о продолжении итераций, вычисляется отдельно на каждом узле кластера на своем собственном наборе данных, то в строке 44 мы запрашиваем все узлы и находим максимальное значение разницы по всей разностной сетке.

Седьмое. После этого в строке 49 мы рассылаем всем узлам значение этой глобальной разницы, тем самым, дав им возможность принять решение о продолжении вычислений.

Восьмое. И наконец в строке 62 мы завершаем работу среды MPI.

Изменение основной программы на этом закончено. Дополнению подверглась так же подпрограмма нахождения значений искомой функции (строка 67). Поскольку для вычислений этой функции на границах той области, которую обсчитывает узел кластера, необходимо знать значения ячеек из соседних областей. Возникает необходимость в граничном обмене.

Девятое. Для обеспечения граничного обмена непосредственно перед началом очередной итерации каждый процесс обменивается своими границами с соседними процессами справа и слева (строки 81-90). За исключением первого и последнего процесса - у них только один сосед, поэтому первый процесс обменивается границами с соседом справа (строки 81-85), а последний обменивается только с соседом слева (строки 86-90). Алгоритм такого обмена достаточно прост. Для каждого узла кластера первый и последний столбец разностной сетки, в данном случае f0(*,1) и f0(*,ymax), заполнены данными из соседних областей, а граница собственной области проходит по второму и предпоследнему столбцу - f0(*,2) и f0(*,ymax-1). Поэтому процесс обмена между узлами A (левый) и B (правый) происходит по следующей схеме:

- $f0_A(*,ymax-1) \rightarrow f0_B(*,1)$
- $f0_A(*,ymax) \leftarrow f0_B(*,2)$

Для оптимизации обмена вместо последовательного вызова функций передачи (**MPI_SEND**) и приема (**MPI_RECV**) данных мы использовали совмещенную функцию **MPI_SENDRECV**.

На этом изменение подпрограммы вычисления функции завершено. Последнее, что нам осталось сделать, так это - модифицировать подпрограммы задания начальных и граничных условий с учетом параллелизма. Процедуру задания начальных значений **initial** мы менять не стали, поскольку в нашем варианте вся разностная сетка заполняется единообразно (нулями). В случае же граничных условий заполнение разностной сетки разнится в зависимости от узла на котором выполняется процедура.

Десятое. Верхняя и нижняя граница сетки f0(1,*) и f0(xmax,*) заполняется единообразно нулями и не зависит от номера процесса. Левая граница f0(*,1) заполняется только первым процессом, а правая f0(*,ymax) - только последним. Поэтому в подпрограмме **boundary** предусмотрена проверка номера процесса (строки 139 и 147) и три блока: первый (строки 132-135) выполняется всеми процессами, второй (строки 140-142) выполняется только первым процессом и третий (строка 147) выполняется только последним процессом.

Как видите, процесс распараллеливания программы требует в сущности минимальных действий.

Сравнение быстродействия

После того, как мы написали два варианта программы попробуем сравнить их быстродействие и посмотреть, стоила ли овчина выделки. Ниже приведена сводная таблица тестов последовательной программы, параллельной программы, выполняемой на одном компьютере и параллельной программы, выполняемой на двух и четырех компьютерах. Во всех случаях параметры задачи и общий размер разностной сетки были одинаковы.

Вариант запуска	Кол-во CPU	Время счета
Последовательная программа	1	3 мин. 35 сек.
Параллельная программа *запуск без MPI (./diffp)*	1	3 мин. 36 сек.
Параллельная программа *запуск с MPI (mpirun ./diffp)*	1	4 мин. 36 сек.
Параллельная программа	2	2 мин. 13 сек.
Параллельная программа	4	0 мин. 56 сек.

Тест показал, что запуск в среде MPI параллельной программы на одном процессоре занимает больше времени. Это вполне ожидаемый результат, поскольку в этом случае появляются накладные расходы на взаимодействия между процессами, хотя в действительности процесс был один. Выполнение параллельной программы без использования среды MPI показывает примерно то же быстродействие, что и обычная последовательная программа.

Данный тест проведен в предположении неизменности размера нашей разностной сетки. Другими словами, мы стремились на одних и тех же данных получить наибольшее быстродействие. Во всех тестах общая расчетная область имела размер 4000x6000 ячеек.

Как уже говорилось ранее, кластер может использоваться не только для увеличения скорости счета, но и для увеличения доступной оперативной памяти. Поэтому имеет смысл протестировать параллельную задачу в условиях, когда размер локальной сетки на каждом из узлов кластера будет равен исходному, то есть 4000x6000, то есть мы увеличиваем доступную память в N раз, где N - количество вычислительных узлов.

На момент написания этого раздела мне был доступен тестовый кластер, состоящий из трех компьютеров с двухядерными процессорами и 1 Гб памяти на борту. Поскольку два массива REAL*8 размером 4000x6000 занимают в памяти примерно 366 Мб, то, несмотря на двухядерность, во избежание проблем с памятью (свопинг и все такое...), я не стал запускать по два процесса на одной машине, и сравнил быстродействие последовательной программы, запущенной на одном процессоре и быстродействие параллельной программы с троекратным увеличением общего объема данных, запущенной на трех процессорах. Результаты теста получились такими:

Размер матрицы (общий)	Кол-во CPU	Время счета
4000x6000	1	3 мин. 35 сек.
4000x18000	3	4 мин. 37 сек.

Результат тоже впечатляющий. Действительно, увеличение объема данных (размера разностной сетки) в 3 раза при использовании трехпроцессорного кластера потребовало расчетного времени всего в 1.3 раза больше.

Эффект, как говорится, на лицо!

Возможные недопонимания

В некоторых случаях может сложиться впечатление, что кластер не работает, или работает совсем не так, как это следует из теории. Мне несколько раз приходилось общаться с людьми, которые попробовали развернуть тестовый кластер и прогнать на нем пример из предыдущего раздела (уравнение теплопроводности). Результаты получались неожиданные:

параллельная программа работала в лучшем случае не медленнее, чем обычная последовательная. Сразу скажу, теория, описанная в предыдущих разделах, работает, когда кластер состоит из одинаковых узлов с одинаковым быстродействием. Однако, часто случается, что для тестирования параллельных программ используются доступные на момент проведения эксперимента компьютеры, обладающие различным быстродействием. Это и есть источник неожиданных разочарований.

Кластер из N узлов работает с эффективным быстродействием, равным быстродействию самой медленной машины, умноженному на N. Если не предпринять дополнительных действий.

Для того, чтобы понять, что в реальности происходит с параллельной программой, работающей на кластере, состоящим из неодинаковых узлов, возьмем нашу программу расчета уравнения теплопроводности diffp.f. Размер разностной сетки для нашего эксперимента возьмем равным 3000x3000 элементов. Кластер для эксперимента будет построен из трех машин, обладающих различным быстродействием.

	Node1	Node2	Node3	Node1,Node2	Node1,Node3	All
Размер сетки	3000x3000	3000x3000	3000x3000	3000x1500	3000x1500	3000x1000
t, Node1	34,04 сек.			14,24 сек.	14,24 сек.	9,09 сек.
t, Node2		11,92 сек.		6,07 сек.		3,95 сек.
t, Node3			63,33 сек.		23,28 сек.	15,22 сек.
Время исполнения **теоретическое**	34,04 сек.	11,92 сек.	63,33 сек.	14,24 сек.	23,28 сек.	15,22 сек.
Время исполнения **реальное**	32,76 сек.	12,15 сек.	62,23 сек.	13,86 сек.	23,97 сек.	15,81 сек.

Экспериментальный кластер, с которым мы будем работать, состоит из трех узлов с различным быстродействием. Самый быстрый узел - Node2. Самый медленный - Node3. Это видно в первых трех колонках приведенной таблицы: параллельная программа с сеткой 3000x3000, запущенная в виде одного процесса, показывает заметно разное время исполнения на различных узлах.

Далее мы разбиваем сетку пополам и пытаемся выполнить параллельную программу, задействовав по два разных узла - четвертая и пятая колонки сводной таблицы. И, наконец, делим сетку на три равных части (шестая колонка) и запускаем программу на всех трех узлах кластера.

Теоретически общее быстродействие кластера не превышает быстродействия самого медленного узла кластера. При условии, что объем вычислений на всех узлах одинаков, что имеет место быть в нашем эксперименте. Результаты, собранные в таблице, демонстрируют именно такое поведение кластера. Плюс-минус небольшие накладные расходы на передачу данных по сети. Мы видим, что время, потраченное программой непосредственно на вычисления, различается в соответствии с быстродействием узла. Но, поскольку кроме собственно вычислений каждый процесс параллельной программы вынужден синхонизироваться с остальными процессами, то на быстрых узлах после каждой итерации имеется период ожидания, необходимый для того, чтобы медленные узлы "догнали" более быстрые.

Другими словами, если мы производим вычисления над разностной сеткой размером 3000x3000 элементов с помощью кластера, состоящего из трех узлов, и равномерно

распределяем сетку между узлами, то есть по 3000x1000 элементов, то время выполнения параллельной программы 3000x3000 на трех узлах будет примерно равно времени выполнения параллельной программы 3000x1000 на самом медленном узле кластера. Именно такой результат получился в нашем эксперименте.

Для получения же максимального быстродействия кластера при различии мощности узлов, необходимо разностную сетку поделить между узлами НЕ равномерно, а в соответствии со скоростными характеристиками узлов.

В нашем случае сетка должна быть поделена примерно следующем образом:
Node1: 3000x683
Node2: 3000x1950
Node2: 3000x367

При таком распределении сетки общее время выполнения параллельной программы в моем эксперименте было равно 8,28 сек., что почти вдвое лучше, чем при равномерном распределении области вычислений. При таком "правильном" разбиении сетки таблица результатов будет выглядеть уже так:

	Node1	Node2	Node3	All, optimal
Размер сетки	3000x3000	3000x3000	3000x3000	3000x(838|1950|367)
t, Node1	34,04 сек.			6,36 сек.
t, Node2		11,92 сек.		7,81 сек.
t, Node3			63,33 сек.	4,60 сек.
Время исполнения **теоретическое**	34,04 сек.	11,92 сек.	63,33 сек.	7,81 сек.
Время исполнения **реальное**	32,76 сек.	12,15 сек.	62,23 сек.	8,28 сек.

Работа с файлами

Важным моментом при написании параллельной программы является работа по сохранению данных в файлы или восстановлению данных из них. Примерами необходимости таких действий могут служить задачи загрузки начальных данных в массивы при запуске программы и сохранение промежуточных данных и/или результатов счета.

Если говорить о загрузке начальных данных или сохранении результатов счета, то эта процедура происходит в самом начале работы программы или в самом ее конце, и никаких особенностей или отличий о того, как это может быть сделано в последовательной программе, для параллельной программы нет.

Однако, в том случае, когда речь идет о сохранении промежуточных данных, появляются некоторые нюансы, связанные со структурой параллельной программы. Дело в том, что файловые операции - очень медленные операции. Если сохранение данных происходит в каждом итерационном цикле, то такое действие может существенно снизить быстродействие программы, что кстати верно и в случае обычного последовательного алгоритма. Поэтому при проектировании параллельной программы следует придерживаться двух принципов:

- Сохранение данных должно происходить не в каждом итерационном цикле, а, скажем, в каждом десятом. В любом случае время между моментами сохранения болжно быть много больше времени, требующемся для записи данных в файл.
- Сохранение данных должно быть реальзовано с использованием неблокирующих

операторов ввода-вывода, позволяющих совместить по времени файловые операции и вычисления.

Рекомендации, данные на этой странице, проверялись на компиляторе Intel Fortran. Проверьте, поддерживает ли ваш компилятор процедуры асинхронной записи. Если нет - ну что ж, вам придется отказаться от распараллеливания процедур вычисления и сохранения данных. Постарайтесь делать запись на диск как можно реже, чтобы это минимальным образом сказалось на быстродействии вашего кластера.

Посмотрим на конкретном примере, как можно реализовать эти два принципа. Возьмем нашу программу вычисления уравнения теплопроводности, а именно - блок, описывающий итерационный цикл:

```
 55  c     Подпрограмма вычисления искомой функции
 56        subroutine iter(f0,f1,xmax,ymax,df)
 57        include 'mpif.h'
 58        integer xmax,ymax
 59        real*8 f0(xmax,ymax), f1(xmax,ymax)
 60        real*8 dt,dx,dy
 61        real*8 df
 62        integer x,y,n,myid,np,ierr,status(MPI_STATUS_SIZE)
 63        common myid,np
 64        dt=0.01
 65        dx=0.5
 66        dy=0.5
 67  c     Обмениваемся границами с соседом
 68        if ( myid .gt. 0 )
 69       x call MPI_SENDRECV(
 70       x f0(1,2), xmax, MPI_REAL8, myid-1, 1,
 71       x f0(1,1), xmax, MPI_REAL8, myid-1, 1,
 72       x MPI_COMM_WORLD, status, ierr)
 73        if ( myid .lt. np-1 )
 74       x call MPI_SENDRECV(
 75       x f0(1,ymax-1), xmax, MPI_REAL8, myid+1, 1,
 76       x f0(1,ymax), xmax, MPI_REAL8, myid+1, 1,
 77       x MPI_COMM_WORLD, status, ierr)
 78  c     Вычисляем функцию в ячейках сетки
 79        do x=2,xmax-1
 80          do y=2,ymax-1
 81          f1(x,y)=f0(x,y)+dt*(
 82       x    (f0(x+1,y)-2*f0(x,y)+f0(x-1,y))/(dx*dx)
 83       x    +
 84       x    (f0(x,y+1)-2*f0(x,y)+f0(x,y-1))/(dy*dy)
 85       x    )
 86          end do
 87        end do
 88  c     Копируем границу для следующей итерации
 89        do x=1,xmax
 90          f1(x,1)=f0(x,1)
 91          f1(x,ymax)=f0(x,ymax)
 92        end do
 93        do y=1,ymax
 94          f1(1,y)=f0(1,y)
 95          f1(xmax,y)=f0(xmax,y)
 96        end do
 97  c     Находим максимальную дельту
 98        df=0.0
 99        do x=1,xmax
100          do y=1,ymax
101            if ( df .lt. abs(f0(x,y)-f1(x,y)) ) then
```

```
102                 df = abs(f0(x,y)-f1(x,y))
103              endif
104            end do
105          end do
106          return
107          end
```

Поскольку мы решили, что сохранение данных будет происходить на каждой 10-й итерации, то есть запись будет происходить в зависимости от номера итерации, то мы должны будем передавать в тело подпрограммы номер текущего цикла. Поэтому вначале изменим описание подпрограммы, добавив дополнительный параметр ее вызова (строка 56).

Как мы знаем, рассматриваемая нами подпрограмма по неизменяемым (и это важно!) в течение цикла данным массива f0, вычисляет значения массива f1. Поэтому в ее теле мы должны решить, наступил ли десятый цикл, и если да, то инициировать неблокирующую запись массива f0 в файл. Что мы и делаем в строке 77b. Далее мы открываем файл и инициируем процесс записи в него массива данных (строки 77k и 77q).

Кроме того, надо позаботиться о том, чтобы каждый процесс параллельной программы сохранял свою часть данных в отдельный, принадлежащий только ему файл. Вычислением название файла данных мы в строке 77d.

После инициации записи управление передается циклу, в котором и происходят необходимые нам вычисления, причем процессы вычисления и записи идут параллельно. Перед возвратом управления из подпрограммы мы должны удостовериться, что процедура записи в файл закончена и закрыть файл (строки 105b и 105c).

Теперь посмотрим, как это можно реализовать в программном коде.

```
 55  c     Подпрограмма вычисления искомой функции
 56        subroutine iter(f0,f1,xmax,ymax,df,n)
 57        include 'mpif.h'
 58        integer xmax,ymax
 59        real*8 f0(xmax,ymax), f1(xmax,ymax)
 60        real*8 dt,dx,dy
 61        real*8 df
 62        integer x,y,n,myid,np,ierr,status(MPI_STATUS_SIZE)
 63        common myid,np
-----------------------------------------------------------------
 63a       character(LEN=20):: st,fname
 63b       logical ex
-----------------------------------------------------------------
 64        dt=0.01
 65        dx=0.5
 66        dy=0.5
 67  c     Обмениваемся границами с соседом
 68        if ( myid .gt. 0 )
 69       x call MPI_SENDRECV(
 70       x f0(1,2), xmax, MPI_REAL8, myid-1, 1,
 71       x f0(1,1), xmax, MPI_REAL8, myid-1, 1,
 72       x MPI_COMM_WORLD, status, ierr)
 73        if ( myid .lt. np-1 )
 74       x call MPI_SENDRECV(
 75       x f0(1,ymax-1), xmax, MPI_REAL8, myid+1, 1,
 76       x f0(1,ymax), xmax, MPI_REAL8, myid+1, 1,
 77       x MPI_COMM_WORLD, status, ierr)
-----------------------------------------------------------------
 77a c      Если десятый цикл, то сохраняем данные
 77b        if ( ((n/10)*10 .EQ. n ) then
 77c c         Вычисляем название файла данных
 77d           write(fname,'(A,I2.2,A)') "node",myid,".dat"
```

```
 77e c         Проверяем существует ли файл для записи
 77f           inquire(file=fname, exist=ex)
 77g c         Если существует, то открываем его для перезаписи,
 77h c         если файл не существует - открываем его как новый
 77i           write(st,'(A)') 'new'
 77j           if ( ex ) write(st,'(A)') 'old'
 77k           open(unit=20,
 77l      $         file=fname,
 77m      $         asynchronous='yes',
 77n      $         status=st,
 77o      $         form='unformatted')
 77p c         Инициируем запись массива f0 в файл
 77q           write(20,ID=idvar,asynchronous='yes') f0
 77r       endif
-----------------------------------------------------------------
 78  c     Вычисляем функцию в ячейках сетки
 79        do x=2,xmax-1
 80          do y=2,ymax-1
 81          f1(x,y)=f0(x,y)+dt*(
 82       x    (f0(x+1,y)-2*f0(x,y)+f0(x-1,y))/(dx*dx)
 83       x    +
 84       x    (f0(x,y+1)-2*f0(x,y)+f0(x,y-1))/(dy*dy)
 85       x    )
 86          end do
 87        end do
 88  c     Копируем границу для следующей итерации
 89        do x=1,xmax
 90          f1(x,1)=f0(x,1)
 91          f1(x,ymax)=f0(x,ymax)
 92        end do
 93        do y=1,ymax
 94          f1(1,y)=f0(1,y)
 95          f1(xmax,y)=f0(xmax,y)
 96        end do
 97  c     Находим максимальную дельту
 98        df=0.0
 99        do x=1,xmax
100          do y=1,ymax
101            if ( df .lt. abs(f0(x,y)-f1(x,y)) ) then
102                 df = abs(f0(x,y)-f1(x,y))
103            endif
104          end do
105        end do
-----------------------------------------------------------------
105a c     Дожидаемся окончания записи и закрываем файл
105b      wait(unit=20,ID=idvar)
105c      close(unit=20)
-----------------------------------------------------------------
106       return
107       end
```

Процедура записи данных, так, как она представлена на этой странице, не вполне оптимальна. Дело в том, что после выхода из подпрограммы до начала следующего цикла выполняется код, который не модифицирует записываемый массив. И код этот выполняется достаточно долго. Но для простоты восприятия мы всю процедуру сохранения данных уложили внутрь рассматриваемой подпрограммы. Выше дана ссылка на исходный код программы, в котором учтена эта недоработка.

В заключение посмотрим, как изменились временные (скоростные) характеристики нашей программы от добавления процедуры записи. В представленной ниже таблице дана

длительность работы нашей программы в трех вариантах. Первый вариант - наша исходная программа без записи данных в файл. Второй вариант - программа с записью данных в синхронном режиме, то есть без распараллеливания процедур вычисления и записи. И третий вариант - программа с записью данных на диск в асинхронном режиме, то есть с распараллеливанием счета и записи в файл.

Точность вычислений в программе немного увеличена по сравнению с исходной программой, которую мы рассматривали в предыдущих разделах, чтобы потребовалось большее число итераций для нахождения решения уравнения. Все варианты программы запускались для работы на трех процессорах. Запись данных велась на расшаренный в NFS ресурс, кроме первого процесса, где запись велась на локальный диск. Кроме того даны результаты, полученные при записи данных на локальный диск (всеми процессами параллельной задачи).

Вариант програмы	**Файловая система**	**Время Счета**
Записи данных нет	не исп.	1 мин. 26 сек.
Синхронная запись	NFS	2 мин. 46 сек.
Синхронная запись	локальная	1 мин. 37 сек.
Асинхронная запись	NFS	2 мин. 37 сек.
Асинхронная запись	локальная	1 мин. 28 сек.

> ***Резюме.*** *По результатам тестов можно сделать заключение. Использование NFS сильно замедляет процесс счета. Поэтому стоит подумать об использовании на вычислительных узлах кластера локальных файловых систем для сохранения данных. Однако в этом случае вам придется перед запуском программы и после окончания счета синхронизировать данные на всех узлах, поскольку вообще говоря заранее неизвестно, на каком узле кластера запустится процесс программы с определенным номером. Другими словами файл, созданный процессом 2 на узле node005 в следующий раз может понадобиться процессу 2 на узле node007.*

Следующий вывод, который можно сделать - использование асинхронной записи позволяет увеличить общее быстродействие программы, хотя и не существенно. Однако преимущество асинхронной записи будет тем заметнее, чем больше времени у вас будет занимать вычисление одного цикла итерации.

Кластер: NFS или локальные файлы?

В предыдущих разделах мы рассматривали варианты периодического сохранения данных в файлы на диске. Если задача требует достаточно много процессорного времени, то может возникнуть ситуация, когда вычислительный процесс требуется прервать, и возобновить его через некоторое время с той точки, где он был прерван.

Преимущества и недостатки

Есть два способа хранения файлов данных: на "расшаренном" в NFS ресурсе и на локальном диске. Оба способа имеют свои преимущества и недостатки. В случае, когда используется виртуальный кластер, альтернативы нет - может использоваться только NFS. Когда же мы используем стационарный кластер, имаются варианты.

Чем хороша NFS? Преимущества NFS заключаются в простоте ее использования и экономии места на диске. Ресурс, экспортированный в NFS, доступен каждому узлу кластера. Нет необходимости копировать исполняемые файлы и файлы данных на все вычислительные узлы. Если данные занимают много места, а обычно бывает именно так - сохраняется

состояние всей разностной сетки на какой-то момент времени,- то хранятся они в одном экземпляре на главном узле кластера. Нет необходимости заботиться о доступности нужных для конкретного процесса параллельной программы данных - все доступно всем.

Чем плоха NFS? Плоха она тем, что запись данных на сетевой ресурс происходит существенно медленнее по сравнению с записью на локальный диск. Это драматически сказывается на быстродействие всего кластера. Конечно можно минимизировать такое отрицательное влияние, как можно реже используя процедуры записи. В идеальном случае - сохранять данные только в конце работы программы. Однако это не всегда удобно, поскольку не всегда можно определить сколько будет работать программа и не придеться ли остановить ее раньше срока. Например, придя утром, вы обнаруживаете, что программа еще не закончила работу, а кластер нужно уже освободить. Наша тестовая программа решения уравнения теплопроводности, которую мы рассматривали ранее, в модификации, когда запись происходит на каждом втором цикле, в случае записи на локальный диск выполнялась 1 мин. 12 сек., а в случае записи на сетевой ресурс время счета уже составило 4 мин. 53 сек., при общем объеме записываемых данных 1.125 Гб и 30 итерационных циклах.

Чем хороша локальная ФС? Локальная файловая система хороша тем, что у нее отсутствуют недостатки NFS. Запись данных на современные винчестеры происходит достаточно быстро, позволяя нам по мере необходимости более часто выполнять сохранение данных.

Чем плоха локальная ФС? Недостаток локальной файловой системы заключен в ее локальности. Необходимым условием запуска параллельной программы является доступность исполняемого файла и файлов данных всем вычислительным узлам кластера, на которых программа будет запущена. Если в отношение исполняемого модуля все не так страшно - MPI имеет средства для автоматического распространения программы на те узлы, где она будет выполняться, то с файлами данных все не так хорошо. У параллельной программы есть особенность - в общем случае мы заранее не знаем, на каких узлах будет выполняться тот или иной процесс. Поэтому на потребуется скопировать все файлы на все узлы кластера, что приводит к многократному дублированию информации и неоптимальному расходованию дискового пространства. Конечно, мы можем явно указать на каких узлах кластера будет запущен данный конкретный параллельный процесс, однако это требует от пользователя знания архитектуры и топологии кластера, что не всегда удобно или возможно.

Тем не менее, если проблема свободного дискового пространства не является для вас проблемой, то предпочтительнее использовать локальную файловую систему. Давайте посмотрим, как можно сгладить недостатки локальной ФС, о которых мы упомянули выше.

Использование локальной файловой системы

Нет необходимости копировать исполняемый модуль на узлы кластера. Команда запуска параллельной программы **mpirun** имеет для этого встроенные средства. Достаточно указать дополнительный ключ **-s** и программа будет автоматически распространена между узлами. Например:

mpirun -hostfile ./../mpi/mpi.hosts -np 3 -s ./diffpw

Благодаря этому ключу mpirun непосредственно перед запуском параллельной программы скопирует исполняемый файл этой программы на все вычислительные узлы кластера, на которых она будет запущена. Таким образом пользователю нет необходимости заботиться о наличии копии программы на узлах кластера и нет надобности в наличии сетевой файловой системы NFS.

Однако для этого придется установить двунаправленные доверительные отношения между узлами. Ранее мы рассматривали, как организовать беспарольный доступ главного компьютера кластера на все узлы. Точно такую же процедуру надо провести в обратном

направлении, то есть обеспечить беспарольный доступ узлов к главному компьютеру.

Если используется локальная файловая система, то локальные для процесса данные будет лучше сохранять в отдельные файлы - так, как это сделано в нашей тестовой программе, то есть запись данных происходит в файлы с названиями (например) nodeNN.dat, где NN - номер процесса, который монопольно использует данный файл. В противном случае, когда данные пишутся в различные сегменты одного и того же файла, синхронизация такого единого файла между узлами кластера превращается в нетривиальную задачу.

Для синхронизации данных между многочисленными узлами кластера удобно использовать программу pdsh. Эта программа не входит в список программ, устанавливаемых в систему по умолчанию, поэтому придется ее доустановить. В Ubuntu это можно сделать следующей командой: **sudo apt-get install pdsh**.

При использовании для хранения данных локальной ФС, работа должна проходить в три этапа: передача файлов данных с главного компьютера кластера на все вычислительные узлы, запуск и выполнение параллельной задачи, получение на главный компьютер измененных файлов данных с вычислительных узлов. Пример последовательности команд для выполнения этих трех этапов такой:

```
cat $HOME/nodes| pdsh -R ssh -w - "rsync -uti supergate:$HOME/mpi1/node*.dat $HOME/mpi1/"
mpirun -hostfile $HOME/mpi.hosts -np 3 -s ./diffpw
cat $HOME/nodes| pdsh -R ssh -w - "rsync -uti $HOME/mpi1/node*.dat supergate:$HOME/mpi1/"
```

Здесь мы предполагаем, что файлы с данными называются node00.dat, node01.dat, node02.dat и т.д., рабочая директория программы - $HOME/mpi1, текстовый файл $HOME/mpi.hosts содержит список всех узлов кластера (по одному узлу в строке), первый среди которых - главный компьютер, и, наконец, файл $HOME/nodes содержит список всех узлов кластера (аналогично $HOME/mpi.hosts), но в котором отсутствует главный компьютер.

В случае использования для запуска параллельной задачи менеджера ресурсов Torque, задание на размещение задачи в очередь может быть оформлено примерно так:

```
#!/bin/sh
#
#PBS -l nodes=supergate:ppn=1+2:ppn=1
#PBS -N Flops_TEST
#PBS -m abe
#PBS -M yuri@linux-ekb.info
#PBS -l pmem=100mb
#PBS -l pcput=10:00:00
cd /home/mpiuser/mpi1
cat $HOME/nodes| pdsh -R ssh -w - "rsync -uti supergate:$HOME/mpi1/node*.dat $HOME/mpi1/"
mpirun -np 3 -s ./diffpw
cat $HOME/nodes| pdsh -R ssh -w - "rsync -uti $HOME/mpi1/node*.dat supergate:$HOME/mpi1/"
```

Для синхронизации данных используется команда pdsh, которая на всех компьютерах, перечисленных в $HOME/nodes, в параллельном режиме запускает процедуру копирования файлов с главного компьютера кластера, или в обратном направлении. Копирование выполняется с помощью утилиты rsync, которая анализирует время модификации файлов и копирует только файлы с наиболее свежими данными. Перед запуском нашей параллельной программы мы распространяем среди всех узлов наиболее свежие данные, которые, если программа это предусматривает, могут быть использованы для загрузки начального состояния разностной сетки. После окончания работы программы процедура копирования загружает на главный компьютер все результаты счета.

Резюме

Использование локальной файловой системы совсем ненамного усложняет процедуру выполенения параллельной программы, существенно увеличивая скорость операций ввода-вывода, повышая тем самым эффективность кластера.

Прерывание счета

Рекомендации, данные в этом разделе, относятся скорее к общей практике программирования, нежели параллельному программированию, как таковому. Тем не менее, думаю, что стоит заострить на них внимание.

Поскольку методы и способы параллельного программирования имеют отношение к "*тяжелым*" задачам, требующим большого времени счета, то вполне представима ситуация, когда потребуется по каким-то причинам досрочно, до получения окончательного результата, прервать счет. Например по причине истечения выделенного для работы на кластере времени. В случаях, когда параллельная программа выполняет периодическое сохранение данных на диск, весьма желательно осуществить такое прерывание аккуратно, способом, не приводящим к риску повреждения сохраненных данных. Риск повреждения данных связан с выбором момента времени прерывания. Если задача будет принудительно остановлена в тот самый момент, когда происходят файловые операции ввода-вывода, то есть большая вероятность, что данные сохранятся не полностью, что приведет к невозможности их дальнейшего использования.

Наибольшую актуальность корректное завершение программы приобретает в случае, когда для ее запуска используется менеджер ресурсов Torque, а для сохранения данных - локальная файловая система. Если остановить выполнение параллельной программы системными средствами Torque, то возникает не только риск нарушения целостности сохраненных данных, но и процедуры синхронизации сохраненных данных, которые должны быть запущены по завершению задачи, и которые прописаны в файле-задании Torque, выполнены не будут.

Наиболее простой в реализации способ принудительной остановки программы - это выставление семафоров. Семафор - это простой файл произвольного содержания (или без такового, то есть нулевой длины), наличие которого является сигналом остановки для программы. Наиболее безопасным с точки зрения сохранности данных моментом остановки является промежуток между окончанием очередного цикла итерации, за которым может следовать вызов процедуры сохранения данных, и началом следующего цикла. Рассмотрим фрагмент нашей тестовой программы решения уравнения теплопроводности.

```
33  c     Получаем разницу значений функции на слоях
34  c     с каждого узла кластера и находим максимальное значение
35  c     этой разницы для всей разностной сетки
36        call MPI_REDUCE(df, gdf, 1, MPI_REAL8,
37       x    MPI_MAX, 0, MPI_COMM_WORLD, ierr)
38  c     Сообщаем найденную максимальную разницу
39  c     всем узлам кластера
40        df=gdf
41        CALL MPI_BCAST(df, 1, MPI_REAL8, 0, MPI_COMM_WORLD, IERR)
42  c     Выводим на экран разницу между итерациями
43        if ( myid .eq. 0 ) write(*,*) 'Diff:',df
44  c     Если разница больше заданной, то решение не найдено
45  c     и мы идем на следующий цикл
46        if (df .GT. 0.01) goto 1
```

Именно в этом месте предыдущий цикл итерации уже закончился, следующий еще не начался, а файловые операции, если они имели место в предыдущем цикле, так же уже завершены.

Обратим внимание на строку 40. В этом месте программы переменной **df** присваивается значение, на основе которого все процессы параллельной задачи принимают решение о необходимости продолжения счета. Логично будет именно в этом месте произвести проверку существования файла-семафора, и в случае его наличия занести в переменную **df** значение,

останавливающее дальнейшие вычисления.

С учетом вышесказанного, фрагмент нашей программы приобретает следующий вид:

```
 33  c     Получаем разницу значений функции на слоях
 34  c     с каждого узла кластера и находим максимальное значение
 35  c     этой разницы для всей разностной сетки
 36        call MPI_REDUCE(df, gdf, 1, MPI_REAL8,
 37       x    MPI_MAX, 0, MPI_COMM_WORLD, ierr)
 38  c     Сообщаем найденную максимальную разницу
 39  c     всем узлам кластера
 40        df=gdf
40a c     Проверяем существует ли глобальный файл-семафор
40b       inquire(file="/etc/stop.stop", exist=ex)
40c       if ( ex ) df=0.0
40d c     Проверяем существует ли локальный файл-семафор
40e       inquire(file="stop.stop", exist=ex)
40f       if ( ex ) df=0.0
 41       CALL MPI_BCAST(df, 1, MPI_REAL8, 0, MPI_COMM_WORLD, IERR)
 42  c     Выводим на экран разницу между итерациями
 43        if ( myid .eq. 0 ) write(*,*) 'Diff:',df
 44  c     Если разница больше заданной, то решение не найдено
 45  c     и мы идем на следующий цикл
 46        if (df .GT. 0.01) goto 1
```

Как вы заметили, мы проверяем два семафора (файлы с именем "stop.stop"): один, находящийся в общесистемном, каталоге, другой - в рабочей директории программы. Наличие хотя бы одного из них приводит к завершению вычислительного процесса. Первый файл, находящийся в каталоге /etc, может быть создан только тем, кто имеет в системе права суперпользователя. Наличие этого файла влияет на выполнение всех, запущенных на кластере параллельных задач, вне зависимости от того, какой у этих задач рабочий каталог и кому они принадлежат. Второй файл влияет на выполнение только той задачи, в рабочем каталоге которой этот файл находится. Создать такой файл может только владелец задачи.

Проверка двойного семафора сделана для удобства системного администратора, которому для остановки всех задач нет необходимости выяснять, какие задачи на данный момент запущены и в каких рабочих директориях.

Таким образом, для корректного завершения программы пользователю необходимо любым удобным способом создать в рабочей директории программы файл с именем **stop.stop**. Администратор же (предполагаем, что кластер используется несколькими пользователями посредством менеджера Torque) для принудительного завершения всех параллельных задач и освобождения кластера может создать файл с таким же именем в системной директории /etc и минут через 10-20 выдать команду "**qdel all**" на случай, если кто-то не воспользовался рекомендацией проверять семафор /etc/stop.stop.

Методика визуализации данных

Одним из важных моментов при проведении численных экспериментов является визуализация полученных данных. Когда промежуточных результатов мало, например важно отобразить лишь факт окончания очередной итерации, можно ограничиться консольным (текстовым) выводом. В этом случае никаких особенных инструментов не требуется. Вывод даных осуществляется стандартными средствами языка. Так же в качестве отдельной задачи можно рассматривать вопрос обработки конечных результатов счета, сохраненных в файле. Для операционной системы Linux существует достаточно широкий спектр приложений, которые можно использовать для визуальной обработки данных, полученных в результате выполнения расчетной задачи. В качестве таких приложений можно упомянуть как

различные электронные таблицы типа OpenOffice.org Calc, Kspread и др., так и широкий список специализированных пакетов а la всем известный Grapher.

Однако вопрос использования подобных средств визуализации выходит за рамки темы настоящей книги. Здесь мы рассмотрим методики оперативного визуального отображения данных непосредственно в процессе счета задачи.

Зачастую возникает необходимость визуализировать промежуточные результаты, полученные по окончании каждой отдельной итерации. Например при проведении численных экспериментов, связанных с решением задач электо-, газо- или термодинамики, может появиться желание посмотреть эволюцию исследуемого процесса во времени. Иногда полученная картинка может повлиять на решение о целесообразности дальнейшего счета. К примеру, если увиденная динамика процесса далека от ожидаемой, то это может свидетельствовать о возможных ошибках в численном алгоритме или неверных начальных и граничых условиях задачи.

Поскольку визуализация должна осуществляться непосредственно во время счета, использование отдельных специализированных пакетов графической обработки данных отпадает. Средства визуализации должны быть встроены в программу как часть численного алгоритма. В Linux существует огромное количество различных библиотек работы с графикой как в режиме консоли (SVGA), так и в среде X-Windows.

В операционной системе UNIX (в частности в Linux) существует одна особенность, которую следует учитывать при разработке программ, имеющих средства визуализации. Дело в том, что консоль кластера (главная машина кластера) не является персональным компьютером в полном смысле этого слова. Скорее всего пользователь, проводящий численное моделирование на кластере, не будет подключен к системе в течение всего времени счета задачи. Закрытие пользователем сеанса работы (logout) без снятия задачи со счета или приостановки задачи возможно только в том случае, когда задача выполняется в режиме демона. То есть у задачи отсутствует как консольный, так и графический вывод. Другими словами, задача выполняется в системе в фоновом режиме. В случае с PVM это означает, во-первых, отсутствие постоянно действующего графического интерфейса задачи, и, во-вторых, программа запущена командой spawn без перенаправления вывода в STDOUT на консоль. То есть использовать "spawn -> myprog" нельзя.

Когда программа запускается командой spawn без перенаправления, весь консольный вывод програмы записывается в log-файл системы PVM (например в /tmp/pvml.0). Таким образом пользователь, запустив программу, может безболезненно закончить сеанс работы с консолью, программа при этом будет продолжать работу. Мониторинг получаемых данных можно в любой момент осуществить с помощью стандартной команды операционной системы tail.

```
tail -f /tmp/pvml.0
```

Запустив эту команду, вы можете в реальном времени отслеживать изменение log-файла. Текст, добавляемый программой в конец log-файла, будет тут же отображаться у вас на текстовой консоли.

Несколько сложнее обстоит дело с процессом визуального отображения данных. Закрытие пользовательского сеанса в худшем случае приведет к аварийному останову программы, в лучшем к закрытию графического окна и к невозможности восстановить это окно при следующих сеансах работы.

Решить эту проблему можно двумя разными способами. Первый способ заключается в разнесении процессов счета и визуализации на две отдельные задачи. Первая запускается на исполнение средствами параллельной виртуальной машины, вторая является обычной пользовательской задачей, которая будет работать только тогда, когда это необходимо. В этом варианте графическая программа должна как-то получать от основной программы данные,

подлежащие визуализации. Наиболее простым способом общения этих двух программ является их связь через файл, в который параллельная программа будет периодически (например после каждой итерации) записывать массив данных. Графическая программа, читая этот файл будет в цикле прорисовывать картинку, в соответствии с полученными данными.

Однако, использовать для подобного процесса обычные файлы нельзя, поскольку могут возникнуть непреодолимые сложности, связанные с синхронизацией операций чтения и записи в файл. Может произойти ситуация, когда графическая программа начала читать очередной блок данных, а программа счета в это время еще не закончила писать данные в файл. В этом случае весьма вероятен сбой. Вторая сложность заключается в объеме передаваемых через файл данных. Расчетная программа на каждой итерации может генерировать достаточно большое количество данных, которые требуется визуализировать. При ограниченном объеме жестких дисков в короткое время файловая система может переполнится, что не приведет ни к чему хорошему.

В операционных системах UNIX имеется такое понятие, как "именованные каналы" или FIFO. Термин "FIFO" - это аббревиатура от английской фразы "first in, fisrt out". Другими словами именованный канал является буфером. Данные, которые в него попадают при записи, могут быть прочитаны из него в той же последовательности, причем прочитанные данные автоматически удаляются из FIFO. Поскольку в UNIX все, начиная с текстовых файлов и кончая сокетами и процессором, для пользователя является файлами, к которым можно обратится обычными средствами файлового доступа, определенными в используемом языке программирования, то и именованные каналы FIFO тоже суть файлы. Используя FIFO для обмена данных, мы автоматически решаем проблему синхронизации чтения/записи данных для наших двух программ и, кроме того, прочитанные и обработанные данные будут автоматически удалены из файловой системы, предотвращая ее переполнение.

Как уже было сказано, с точки зрения программиста, работа с файлами FIFO ничем не отличается от работы с обыкновенными файлами. Отличие заключается в методе создания такого файла. Для создания FIFO следует воспользоваться командой операционной системы mkfifo. Для этого в каталоге, где будет располагаться именованный канал, необходимо выполнить следующую команду:

mkfifo my_fifo_data

Параметром команды является имя создаваемого файла FIFO.

После этого в расчетной части программы, точнее в том ее процессе, который будет исполняться на консоли кластера, в локальной файловой системе которой мы создали наш файл, программист должен организовать периодический сброс в этот файл необходимых для визуализации процесса блоков данных. Открытие файл FIFO в режиме write only и его закрытие может выполняться соответственно в самом начале и в самом конце программы. Оператор записи в файл будет находиться в теле цикла. В программе визуализации именованный канал также открывается и закрывается вне тела цикла. Открытие файла происходит в режиме read only.

Особенностью чтения данных из FIFO является отсутствие для программиста необходимости каким-либо образом организовывать процесс ожидания поступления новой порции данных. Если в программе управление передано оператору чтения до фактического появления в FIFO новой порции данных, выполнение программы автоматически приостанавливается до того момента, когда эти данные появятся в FIFO.

Следует однако отметить, что длительное отсутствие пользователя в системе и, соответственно, отсутствие в системе работающей программы визуализации, которая читает данные из FIFO, приводит к постепенному росту размера файла FIFO и, в конце концов, переполнению, файловой системы, как это происходило бы с обычным файлом.

Рассмотрим второй способ организации процесса визуализации данных. Поскольку с точки зрения операционной системы процесс, запущенный на узле параллельной виртуальной машины, является полноправной пользовательской задачей, то такая задача естественно может иметь графический интерфейс. Другими словами программист имеет возможность заставить материнский процесс параллельной задачи открыть графическое окно и что-либо в этом окне нарисовать. В случае, если выбран такой способ визуализации, на нет необходимости использовать ресурсы файловой системы и исчезают проблемы, связанные с файловым доступом, о которых мы говорили выше.

Однако задача, имеющая графический интерфейс, не работает в режиме демона (фоновом режиме) и, соответственно, нет возможности безболезненно для этой задачи закрыть пользовательскую сессия (logout). Проблема эта решаемая. Поскольку закрытие графического окна у задачи переводит ее в фоновый режим, можно заставить нашу программу визуализировать данные только тогда, когда мы этого хотим. Вопрос в том, как сообщить программе, что ей надо открыть графический интерфейс и показать нам, что она нам такое насчитала. Самый простой способ сделать это - использовать файл-триггер, наличие которого будет сигнализировать программе, что графический интерфейс требуется, отсутствие же которого - есть сигнал программе закрыть графическое окно, если оно открыто. В FORTRANe вероятно будет проще использовать файл-триггер несколько по-другому: файл присутствует всегда, а сигналом к открытию/закрытию графического окна служит его содержимое. Наиболее просто управлять таким файл-триггером можно с помощью следующего простого скрипта:

```
#!/bin/sh
#
if [ ! -e ~/trigger.dat ]; then
     touch ~/trigger.dat
fi

SHOW_FLAG=`cat ~/trigger.dat`

if [ "${SHOW_FLAG}" = "0" ];  then
    echo -n 1 > ~/trigger.dat
else
    echo -n 0 > ~/trigger.dat
fi
```

Этот скрипт проверяет содержимое файла trigger.dat, находящегося в домашнем каталоге пользователя (при необходимости создавая этот файл), и меняет его содержимое в циклическом порядке: "0" меняет на "1" и наоборот. Программа должна периодически читать этот файл и в зависимости от прочитанных данных открывать или закрывать графическое окно.

Перейдем непосредственно к способам создания в программе графического окна и прорисовки в нем изображения. Как уже говорилось ранее, инструментов для работы с графикой существует огромное количество. Но, поскольку цель этой книги - показать наиболее простые способы воспользоваться всеми прелестями кластерной архитектуры, то позволю себе дать одну рекомендацию.

Существует одна очень простая в установке и использовании библиотека: G2. Эта библиотека имеет еще то преимущество, что может использоваться как в программах, написанных на C, так и в программах на FORTRANe. Получить эту библиотеку можно на сайте разработчиков http://g2.sourceforge.net или в разделе Download этого сайта. На момент написания книги текущая версия библиотеки была 0.49a. Библиотека распространяется в исходных кодах. Забыл упомянуть еще одно замечательное свойство библиотеки G2. Библиотека может с одинаковой легкостью использоваться как для прорисовки графики в окне X-Windows, так и для создания аналогичной графики в виде .png или .jpg - файлов. Последнее представляет интерес для создания последовательности слайдов из которых при желании можно сделать презентацию или фильм, сконвертировав их в avi или DivX. Так же с помощью этой

библиотеки аналогично созданию графических файлов или прорисовки графики в окне создавать файлы в формате Enchanced PostScript, подготовленные для печати на любом принтере, понимающем PostScript или, если печать ведется в Linux, вообще на любом принтере.

Для установки библиотеки G2 в вашей системе необходимо для начала распаковать архив с исходными кодами в каком-либо каталоге.

После этого, для установки библиотеки, вы должны воспользоваться сакраментальной последовательностью команд:

```
./configure
make depend
make
make install
```

Теперь рассмотрим практические вопросы программирования с использованием библиотеки G2 на языках C и FORTRAN. Подробно рассматривать ВСЕ функции, которыми вы можете пользоваться при создании графических изображений мы не будем, описания функций вы всегда можете посмотреть в прилагаемой к библиотеке документации. Вместо этого приведем два примера программ, которые в окне X-Windows размером 640x640 пикселей рисуют распределение некоей функции и кратко опишем использованные в программе функции.

Для конкретики предположим, что мы считаем некоторую двумерную газодинамическую задачу. Разностная сетка у нас имеет размер 640x640. По окончании каждой итерации счета мы получаем распределение плотности по пространству. Для отображения на картинке величины плотности в каждой ячейки нашей разностной сетки мы будем использовать 255 градаций серого цвета. Минимальная плотность будет рисоваться черным цветом, максимальная - белым. Конечно саму программу расчета дифференциальных уравнений мы писать не будем, вместо этого в том месте программы, где мы должны получить массив плотностей, запишем простой цикл, в котором этот массив будет заполнен неким осмысленным, но произвольным образом.

Итак, программа на языке C:

```c
#include <g2.h>
#include <g2_X11.h>

main()
{
  int d,i,x,y;
  //массив плотностей
 double q[640][640];
  //массив соответствующих цветов
  int colors[640][640];
  //специальный массив, в котором хранятся оттенки серого цвета
  int grays[255];

  //откроем графическое окно размером 640x640
  d=g2_open_X11(640,640);

  //подготавливаем 255 возможных оттенков серого
  for(i=0;i<255;i++)
      grays[i]=g2_ink(d,(double) (i/254.0),(double) (i/254.0),(double)
(i/254.0));

  ////////////////////////////////////////////////////////
  //подготавливаем массив плотности (произвольным образом)
  //вместо чтения из fifo-файла или из pvm-сообщения заполняем массив вручную
  //в реальной задаче вместо этого куска кода будет
```

```
  //стоять один оператор чтения данных из файла или
  //операторы приема сообщения и распаковки его в массив
  for(x=0;x<640;x++) {
     for(y=0;y<640;y++) {
    q[x][y]=sqrt(abs((300*300)-(x*y)));
     }
  }
  ////////////////////////////////////////////////////////

  //вычисляем максимум и минимум плотности
  double min=q[0][0]; double max=q[0][0];
  for(x=0;x<640;x++) {
     for(y=0;y<640;y++) {
     if(min>q[x][y]) min=q[x][y];
    if(max<q[x][y]) max=q[x][y];
     }
  }

  //вычисляем цвета пикселей
  for(x=0;x<640;x++) {
     for(y=0;y<640;y++) {
         //конформно отражаем распределение плотностей
         //к диапазону 0-254 и в качестве цвета пикселя
         //берем целую часть от полученной величины
         int c=(int) ((254*(q[x][y]-min))/(max-min));
         colors[x][y]=grays[c];
     }
  }

  //рисуем распределение плотности в графическом окне X-Windows
  g2_image(d,0.0,0.0,640,640,&colors[0][0]);

  //рисуем белый прямоугольник в котором расположим текст
  int color = g2_ink(d,1.0,1.0,1.0);
  g2_pen(d,color);
  g2_filled_rectangle(d,15.0,15.0,80.0,35.0);

  //рисуем черным цветом подпись к картинке в
  //нарисованном белом прямоугольнике
  color = g2_ink(d,0.0,0.0,0.0);
  g2_pen(d,color);
  g2_string(d,20.0,20.0,"Test field");

  //немножко поспим чтобы можно было полюбоваться
  //полученной картинкой
  sleep(10);

  //закроем графическое окно
  g2_close(d);
}
```

Теперь сделаем тоже самое, но на языке FORTRAN:

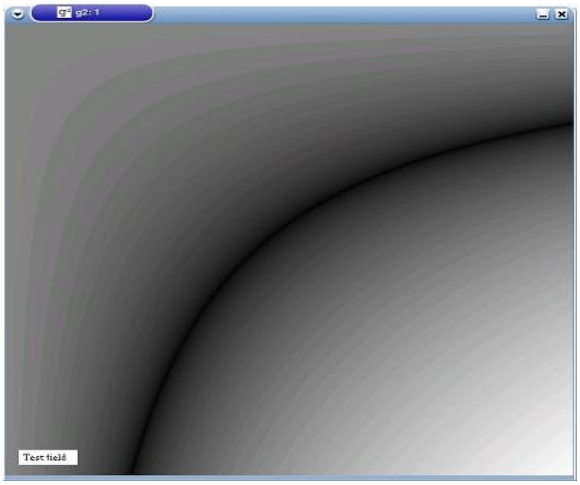
```
        program draw
    integer i,x,y
   real*8 q(0:639,0:639)
   real*8 qmin,qmax
        real colors(0:639,0:639)
        real grays(0:255)
       real c
  real color
```

```
     d=g2_open_vd()
c       откроем графическое окно размером 640x640
 d1=g2_open_X11(640.0, 640.0)
    call g2_attach(d, d1)

c       подготавливаем 255 возможных оттенков серого
      do i=0,254
          grays(i)=g2_ink(d1,i/254.0,i/254.0,i/254.0)
 enddo

c  подготавливаем массив плотности (произвольным образом)
c  вместо чтения из fifo-файла или pvm-сообщения заполняем массив вручную
c  в реальной задаче вместо этого куска кода будет
c  стоять один оператор чтения данных из файла или
c  операторы приема сообщения и распаковки его в массив
 do x=0,639
          do y=0,639
          q(x,y)=sqrt(abs((300.0*300.0)-(x*y*1.0)))
           enddo
       enddo
c  ////////////////////////////////////////////////////////


c         вычисляем максимум и минимум плотности
  qmin=q(0,0)
     qmax=q(0,0)
     do x=0,639
          do y=0,639
          if(qmin.gt.q(x,y)) then
             qmin=q(x,y)
         endif
           if(qmax.lt.q(x,y)) then
             qmax=q(x,y)
         endif
       enddo
       enddo


c        вычисляем цвета пикселей
        do x=0,639
          do y=0,639
c                 //конформно отражаем распределение плотностей
c          //к диапазону 0-254 и в качестве цвета пикселя
c         //берем целую часть от полученной величины
              c=((254*(q(x,y)-qmin))/(qmax-qmin))
             colors(x,y)=grays(c)
         enddo
        enddo

c рисуем распределение плотности в графическом окне X-Windows
     call g2_image(d1,0.0,0.0,640.0,640.0,colors)
c   call g2_line(d1,15.0,15.0,20.0,20.0)

c  рисуем белый прямоугольник в котором расположим текст
  color = g2_ink(d1,1.0,1.0,1.0)
  call g2_pen(d1,color)
   call g2_filled_rectangle(d1,15.0,15.0,80.0,35.0)

c      рисуем черным цветом подпись к картинке в
c     нарисованном белом прямоугольнике
       color = g2_ink(d1,0.0,0.0,0.0);
 call g2_pen(d1,color);
```

```
  call g2_string(d1,20.0,20.0,"Test field")


     call g2_flush(d1)

c      немножко поспим чтобы можно было полюбоваться
c           полученной картинкой
   read (*,*) a

   call g2_close(d1)

      stop
    end
```

В отличие от C-программы, в FORTRANe все массивы, использованные в параметрах G2-функций не целочисленные, а объявлены как REAL. Это главное отличие и его надо помнить. Второе отличие заключается в размерности массивов. В FORTRANe по-умолчанию индексы массивов начинаются с 1. Чтобы G2-функции правильно понимали данные, переданные из FORTRAN-программы, необходимо массивы объявлять с начальным индексом равным 0. Например "REAL A(0:99)" вместо "REAL A(100)".

Результатом работы обеих программ является картинка, показанная ниже.



В том случае, когда прорисовкой графики занимается материнский процесс параллельной задачи, алгоритм по которому построен этот процесс может быть таким:

```
<инициализируем параллельный процесс>
do <итерация>=1 to N
        <обмениваемся данными с дочерними процессами>
        if <флаг визуализации> = 1 then
                if <графическое окно не открыто> then
                        <открываем графическое окно>
                endif
                <подготавливаем данные для прорисовки>
                <рисуем картинку>
        else
                if <графическое окно запущено> then
                        <закрываем графическое окно>
                endif
        endif
enddo
<заканчиваем работу параллельного процесса>
```

Теперь опишем вкратце те функции, которые мы использовали в наших тестовых программах.

int dev = g2_open_X11(int width, int height) - C
dev=g2_open_X11(width,height) - FORTRAN

функция открывает X11 окно шириной width пикселей и высотой height пикселей, возвращает идентификатор устройства dev вывода, который используется в других функциях

void g2_close(int dev) - C
call g2_close() - FORTRAN

функция закрывает графическое устройство с идентификатором dev открытое предыдущей функцией

void g2_plot(int dev, double x, double y) - C
call g2_plot(dev, x, y) - FORTRAN

функция рисует точку (пиксель) цветм, установленным функцией g2_pen

void g2_filled_rectangle(int dev, double x1, double y1, double x2, double y2) - C
call g2_filled_rectangle(dev, x1, y1, x2, y2) - FORTRAN

функция рисует прямоугольник с координатами левого нижнего угла (x1,y1) и координатами правого верхнего угла (x2,y2), после чего заливает этот прямоугольник цветом, установленным функцией g2_pen

void g2_string(int dev, double x, double y, char *text) - C
call g2_string(dev, x, y, 'some text') - FORTRAN

функция рисует текст, заданный переменной text
позиция начала текста определяется координатами (x,y)
цвет текста устанавливается функцией g2_pen

void g2_image(int dev, double x, double y, int x_size, int y_size, int *pens) - C
call (dev, x, y, x_size, y_size, &pens[0][0]) - FORTRAN

функция заполняет прямоугольную область с координатами левого нижнего угла (x,y), шириной x_size и высотой y_size
цвета пикселей этой области задаются в двумерном массиве pens размерностью (x_size,y_size)

int color=g2_ink(int dev, double red, double green, double blue) - C
color=g2_ink(dev, red, green, blue) - FORTRAN

функция создает новый цвет определяемый RGB параметрами red,green,blue
значения этих параметров - действительные числа в диапазоне 0..1
функция возвращает индекс нового цвета color.

void g2_pen(int dev, int color) - C
call g2_pen(dev, color) - FORTRAN

функция устанавливает текущий цвет для рисования
цвет определяется индексом цвета color который получен с помощью функции g2_ink

Заметки об эффективности кластера

Предыдущая статья дает слишком пессимистические прогнозы по оценке эффективности кластера. Пессимизм заложен в формуле №2 статьи, которая в сущности описывает как сетевую инфраструктуру кластера, так и способ разделения исходной разностной сетки между узлами кластера. В данной работе мы попробуем изменить подход к этим вопросам на более реалистичный.

В статье оценка времени граничного объема записана так:

$$T_{ex}=\frac{E}{V_p} \quad (1)$$

где E – объем данных в области граничного обмена (Гбит), V – пропускная способность сети кластера (Гбит/с). В ней используется скорость порта в единицах Гбит/с. Далее будем обозначать его как V_p . В действительности все немного сложнее.

Есть очень интересный момент. Современные коммутаторы имеют такую характеристику, как «пропускная способность шины». Например 8-портовый коммутатор SuperStack II Switch 9000 имеет пропускную способность 17.2 Гбит/с. Это означает, что **каждый** из восьми портов может одновременно с другими в дуплексном режиме передавать и принимать данные с суммарной скоростью 2 Гбит/с. Точнее говоря, пропускная способность шины задана в пакетах в секунду и заявленные 17.2 Гбит/с достигается при пересылке больших пакетов данных. Но будем считать, что наши граничные области именно такие.

Таким образом, узел передает данные не быстрее, чем позволяет порт, а с другой стороны, суммарная скорость передаваемых и принимаемых всеми узлами данных не может превышать пропускной способности коммутатора (далее будем обозначать ее как V_b).

Поэтому, в предположении, что все узлы инициируют обмен данными одновременно (в том числе одновременно и передачу и прием), исходную формулу расчета времени обмена в условиях дуплексного режима работы сетевых интерфейсов следует записать таким образом:

$$T_{ex}=\frac{2\cdot E_i\cdot 64}{V_p}\,\frac{V_p\cdot(N-1)}{V_b}=\frac{2\cdot E_i\cdot 64\cdot(N-1)}{V_b} \quad \text{для} \quad V_b\leqslant V_p\cdot N \quad (2)$$

и

$$T_{ex}=\frac{2\cdot E_i\cdot 64}{V_p} \quad \text{для} \quad V_b> V_p\cdot N \quad (3)$$

Здесь E_i - количество ячеек разностной сетки типа real*8 отдельно взятого узла, участвующих в граничном обмене. Если мы распределяем исходную разностную сетку между узлами, деля ее равномерно по одной координате, то это величина постоянная.

Поскольку время одной итерации вычислительного процесса складывается из двух величин – времени непосредственно счета и времени обмена данными между узлами кластера

$$T_{it}=T_{calc}+T_{ex} \quad (4),$$

то следует ожидать, что эффективная скорость счета кластера будет расти примерно до момента, когда

$$T_{calc}\approx T_{ex} \quad (5)$$

Таким образом мы можем оценить количество узлов кластера, при котором задача будет

решаться наиболее эффективно. На тестовом компьютере (Intel Pentium 4 3GHz), имеющемся в моем распоряжении скорость расчетов задачи Эйлера при использовании компилятора Intel Fortran Compiler составляет примерно 0.87 секунды на одну итерацию для разностной сетки размером 6000x6000 элементов типа real*8. То есть скорость вычислений равна примерно 2600 Мбит/сек. $2.6 \cdot 10^9 \, бит/сек.$ для одной итерации. Если эту скорость вычислений обозначить как V_c, то время одной итерации можно записать как

$$T_{calc} = \frac{E_n \cdot 64}{V_c} \quad (6)$$

Здесь E_n - количество элементов разностной сетки размерности real*8 (64 бита), которое обрабатывается узлом кластера. Параметр V_c зависит не только от мощности центрального процессора, но может различаться в различных задачах, поскольку в нем заложена «мощность вычислений», то есть количество математических операций, требующихся для обработки одной ячейки разностной сетки.

Будем считать, что целью распараллеливания программы является уменьшение общего времени счета, а не увеличение доступной для расчетов памяти, то есть общий размер разностной сетки не зависит от числа улов кластера. Следовательно, учитывая соотношение (5), оптимальное количество узлов кластера можно оценить по формуле

$$\frac{64 \cdot E_n}{V_c} \approx \frac{2 \cdot 64 \cdot E_i \cdot (N-1)}{V_b} \quad \text{для} \quad V_b \leqslant V_p \cdot N \quad (7)$$

и

$$\frac{64 \cdot E_n}{V_c} \approx \frac{2 \cdot 64 \cdot E_i}{V_p} \quad \text{для} \quad V_b > V_p \cdot N \quad (8)$$

Если считать, что деление сетки между узлами происходит только по одной координате и распределение по узлам выполняется равномерно, то объем данных (количество обрабатываемых ячеек сетки) области вычислений для каждого узла будет равен

$$E_n = E_i \frac{E_x}{N} \quad (9)$$

Здесь E_x - длина разностной сетки по оси X, а E_i - как и раньше длина массива граничной области и одновременно длина разностной сетки по оси Y. Таким образом из уравнений (7), (8) и (9) мы получаем два условия:

$$\begin{array}{c} \dfrac{E_x}{N \cdot V_c} \approx \dfrac{2(N-1)}{V_b} \\ \dfrac{E_x}{N \cdot V_c} \approx \dfrac{2}{V_p} \end{array} \quad (10)$$

Отсюда мы получаем два значения оптимального количества узлов кластера при котором общее время вычислений, требуемое для решения задачи будет минимальным:

$$N \approx \frac{1+\sqrt{1+\frac{2 \cdot E_x \cdot V_b}{V_c}}}{2} \quad (11)$$
$$N \approx \frac{E_x \cdot V_p}{2 \cdot V_c}$$

Первое значение, полученное по теореме Виета, применимо при условии $V_b \leqslant V_p \cdot N$, а второе - при условии $V_b > V_p \cdot N$.

Для иллюстрации вышесказанного попробуем рассмотреть поведение некоего абстрактного кластера при решении задачи Эйлера. Для эксперимента зададим начальные условия:

$E_i = E_x = 6000 -$ *размер вычислительной сетки*
$V_b = V_p = 100\ Мбит/с - 100 -$ *мегабитная сеть, построенная на примитивном хабе*
$V_c = 2.6 \cdot 10^9 -$ *быстродействие компьютера*

Применив соотношение (11) мы получим значения N равные 11 и 115 соответственно. Но, поскольку значение 115 явно не удовлетворяет условию $V_b > V_p \cdot N$, то считаем, что оптимальный размер кластера для нашей задачи составляет 11 узлов. При таком размере кластера наша задача будет считаться примерно в 6 раз быстрее, чем если бы она была посчитана на одной машине.

В качестве проверки наших рассуждений попробуем построить график функции (4) при указанных условиях эксперимента.

Рис. 1

Теперь вернемся к формуле (11). Результат получился на первый взгляд достаточно парадоксальным, а именно: оптимальное количество узлов кластера не зависит от размера области обмена. Однако, как было упомянуто выше, распределение вычислительной области

между узлами кластера осуществлялось таким образом, что размер области обмена оставался постоянным, так что, соотношение времени, потраченного на обработку этой области и времени ее пересылки так же оставалось постоянным, не зависящим от ее размера.

Второй вывод, который можно сделать, заключается в следующем. Оптимальное количество узлов кластера (дающее наибольшее быстродействие) уменьшается с увеличением вычислительной мощности компьютеров, из которых состоит кластер. Но это вполне понятно, поскольку сетевой обмен между узлами кластера будет тем заметнее тормозить общий вычислительный процесс, чем меньше времени будет тратиться непосредственно на вычисления.

Наконец перейдем к более реальному случаю. Как и было сказано в самом начале, предыдущая статья дает слишком пессимистический прогноз эффективности кластера. Следуя формуле (11) Можно предположить, что при использовании современных коммутаторов и увеличении объема вычислений можно ожидать существенного роста эффективности работы параллельной программы. Для проверки этого предположения возьмем процессор Intel E8400 3 GHz, который дает скорость $V_c \approx 1.4 \cdot 10^{10}$ и задачу Эйлера с общим объемом разностной сетки порядка 4 Гб. Сеть кластера построим на упомянутых выше маршрутизаторах SuperStack II Switch 9000.

Таким образом имеем начальные условия:

$E_i = E_x = 23000 - размер\ вычислительной\ сетки$
$V_b = 17.2\ Гбит/с - пропускная\ способность\ шины\ маршрутизатора$
$V_p = 1\ Гбит/с - пропускная\ способность\ сетевых\ интерфейсов\ узлов\ кластера$
$V_c = 1.4 \cdot 10^{10} - быстродействие\ компьютера$

Оценочный размер максимально эффективного кластера согласно формуле (11) будет равен 119 узлов. А время счета по сравнению с одно-процессорным вариантом уменьшится примерно в 60 раз. Причем, если построить график, аналогичный рис. 1, то можно увидеть, что эффективность кластера до размера в несколько десятков узлов растет примерно линейно.

Теперь рассмотрим один нюанс, касающийся высокопроизводительных вычислений с использованием кластерной архитектуры, хотя к кластерам, как таковым, этот нюанс отношения не имеет. Речь пойдет о стиле программирования. Очень часто забывают, что на скорость вычислений влияет не только выбранный алгоритм, который затем реализуется в программный код, но и особенности компилятора в совокупности с реализацией механизмов доступа к оперативной памяти.

Подробно описывать эти особенности я не буду — все это можно найти в специальной литературе. В двух же словах проблема, с которой может столкнуться программист, заключается в том, что в современных компьютерах доступ к ячейкам оперативной памяти организован так, то последовательный доступ к соседним ячейкам выполняется гораздо быстрее, нежели чем последовательный доступ к ячейкам, разделенным большими промежутками.

В данной работе в качестве тестового примера была взята задача Эйлера. Разностное представление решения этой задачи выгладит следующим образом:

$$f^{t+1}_{x,y} = f^{t}_{x,y} + \partial t \cdot \left(\frac{f^{t}_{x+1,y} - 2f^{t}_{x,y} + f^{t}_{x-1,y}}{\partial x^2} + \frac{f^{t}_{x,y+1} - 2f^{t}_{x,y} + f^{t}_{x,y-1}}{\partial y^2} \right)$$

Соответственно этому имеем такой программный код на языке Fortran:

```
 do x=2,xmax-1
  do y=2,ymax-1
  f1(x,y)=f0(x,y)+dt*(
x   (f0(x+1,y)-2*f0(x,y)+f0(x-1,y))/(dx*dx)
x   +
x   (f0(x,y+1)-2*f0(x,y)+f0(x,y-1))/(dy*dy)
x   )
  end do
 end do
```

Казалось бы все правильно. С точки зрения математики так и есть. Однако попытка экспериментально определить скорость обработки данных этой программой при различных размерах разностной сетки дает вот такие странные результаты:

Размер сетки	V_c, ***Мбит/сек.***	**Размер сетки**	V_c, ***Мбит/сек.***
600x600	1963	3000x3000	591
1000x1000	1967	6000x3000	420
1000x3000	732	3000x6000	285
3000x1000	1841	6000x6000	242

Теперь немного изменим текст программы, изменив вложенность циклов:

```
 do y=2,ymax-1
  do x=2,xmax-1
  f1(x,y)=f0(x,y)+dt*(
x   (f0(x+1,y)-2*f0(x,y)+f0(x-1,y))/(dx*dx)
x   +
x   (f0(x,y+1)-2*f0(x,y)+f0(x,y-1))/(dy*dy)
x   )
  end do
 end do
```

Ни с точки зрения математики, ни с точки зрения полученных результатов решения задачи ничего не изменилось. Однако теперь эксперимент показывает радикальное изменение скорости обработки данных. Смотрим таблицу.

Размер сетки	V_c, ***Мбит/сек.***	**Размер сетки**	V_c, ***Мбит/сек.***

600x600	2618	3000x3000	2643
1000x1000	2637	6000x3000	2644
1000x3000	2622	3000x6000	2647
3000x1000	2627	6000x6000	2643

Таким образом, установка правильной последовательности обращения к ячейкам разностной сетки (и соответственно к ячейкам оперативной памяти компьютера) дает возможность не только стабилизировать скорость выполнения программы, убрав зависимость от размеров и конфигурации области счета, но и существенно ее (скорость) увеличить.

Компиляторы для параллельных задач

Обзор

Вопрос выбора аппаратной платформы для кластера, по всей видимости не стоит: мы будем использовать ту технику, которая у нас уже имеется, а это скорее всего компьютеры на основе процессоров Intel. Поэтому обсуждать преимущества той или иной платформы мы не будем. Тем не менее, если вы собираетесь самостоятельно разрабатывать приложения для своего кластера, вам необходимо озаботиться подбором подходящих средств разработки.

Так, для процессоров Intel лучшим выбором будет разработанные Intel компиляторы с языков C/C++/Fortran95 (доступны бесплатно для некоммерческого использования в версии Linux), либо компиляторы от Portland Group (C/C++/Fortran90/HPF). Если же вам по каким-то причинам нет возможности установить у себя эти компиляторы, то вы можете использовать свободно распространяемые компиляторы серии GCC (Gnu Compiller Collection), которые поддерживают большинство современных аппаратных и программных платформ и поставляются практически со всеми дистрибутивами Linux и Unix.

> *Следует заметить, однако, что компиляторы серии gcc обладают одним существенным недостатком - в них отсутствует поддержка типов данных Real*16 и Complex*32, требующиеся для многих научных расчетов. Если ваша задача требует именно такой точности решений, то рекомендую загрузить и установить компиляторы для некоммерческого пользования компании Intel. Более подробно этот процесс описан в разделе Ubuntu-кластер.*

Параллельные программы можно писать "вручную", непосредственно вставляя в нужные места вызовы коммуникационной библиотеки. Этот путь требует от программиста специальной подготовки. Альтернативой является использование систем автоматического и полуавтоматического распараллеливания последовательных программ.

- В случае полуавтоматической системы распараллеливания, в тексте последовательной программы выделяются блоки, которые могут выполнятся параллельно. Обычно, в текст вставляются специального вида комментарии, которые игнорируются обычным (последовательным) компилятором.
- Автоматические системы распараллеливания выполняют декомпозицию

последовательного алгоритма самостоятельно. На вход подается последовательная программа, на выход выдается её параллельный аналог. Системы из этого класса так же могут помочь пользователю выяснить является ли данная задача параллельной, оценить время выполнения данной задачи, определить оптимальное число процессоров.

Создание систем автоматического распараллеливания и распознавания параллелизма в алгоритмах является не простой математической задачей и большинство такого рода систем являются коммерческими.

Выбор компилятора

Наиболее простыми, с точки зрения установки в систему Linux, можно назвать компиляторы серии GCC (Gnu Compiller Collection). Если говорить о Фортране, то коллекция GCC содержит два варианта компилятора: **fort77**, который является препроцессором, перекодирующим фортран-программу в си-код, который затем компилируется стандартным для дистрибутива си-компилятором, и **gfortran**, который является полноценным компилятором подмножества языка Fortran 95. Эти компиляторы входят в состав любого дистрибутива и их установка не представляет никаких трудностей.

Компиляторы фирмы Intel не входят в состав дистрибутивов Linux, однако их получение с официального сайта и установка в систему так же максимально проста. Процедура установки компилятора Intel Fortran описана в разделе Ubuntu кластер.

Приведем сравнительную таблицу быстродействия фортран-программы, откомпилированной различными компиляторами. В качестве тестовой задачи была взята последовательная программа решения уравнения теплопроводности, описанная в одном из предыдущих разделов.

Компилятор	Параметры	Время счета Real*8	Время счета Real*16
fort77	нет	3 мин. 44.56 сек.	не поддерж.
fort77	-O3	2 мин. 36.22 сек.	не поддерж.
gfortran	нет	3 мин. 40.11 сек.	не поддерж.
gfortran	-O3	2 мин. 29.21 сек.	не поддерж.
ifort	нет	0 мин. 08.92 сек.	7 мин. 48.22 сек.
ifort	-O3	0 мин. 08.77 сек.	7 мин. 41.57 сек.

Тестирование показало, что фортран-компилятор фирмы Intel дает на выходе исполняемый код, который примерно в 17 раз быстрее полученного в результате использования компиляторов GCC. Кроме того, только Intel Fortran позволяет использовать типы данных Real*16 и Complex*32, хотя физический процессор и не умеет непосредственно работать с такими типами (отсюда и существенное увеличение времени счета на таких данных - используется софтверная эмуляция).

Резюме

Если параллельный кластер используется вами исключительно в целях обучения, то компиляторы GCC вас вполне удовлетворят. Однако, если вы заинтересованы в получении максимальной эффективности и наибольшей скорости счета ваших программ, то стоит потратить немного времени и усилий для установки компиляторов Intel.

Операционная система

При выборе операционной системы следует основываться прежде всего на рекомендациях разработчиков программного обеспечения. Однако, если есть выбор, то при прочих равных условиях следует отдать предпочтение Linux.

Под Linux доступно огромное количество серверного ПО, компиляторов, библиотек, средств отладки и пр. Большое количество программного обеспечения имеется в свободном доступе, для многих программ есть исходные коды и обширная документация.

Плюсом Linux является "прозрачность" для пользователя и системного администратора, что позволяет быстрее и проще разрешать все возникающие проблемы.

Однако зацикливаться на выборе операционной системы не надо. Поскольку вы являетесь не системным администратором, а научным работником, операционная система для вас значит не больше, чем офис для бизнесмена. Основными вашими инструментами являются карандаш, бумага и транслятор с вашего любимого языка программирования. Кластерный суперкомпьютер есть не цель, но всего лишь средство для усовершенствования и оптимизации вашей основной работы.

Основой кластера является не операционная система, а коммуникационная среда (MPI), обеспечивающая возможность частям параллельной программы, выполняющимся на разных компьютерах, эффективно взаимодействовать между собой.

Рассмотренные ранее средства для построения кластера (MPI) имеют реализации как для операционных систем семейства UNIX (Linux, FreeBSD и т.п.), так и для систем фирмы Майкрософт. Поэтому, если вы испытываете непреодолимые трудности в отказе от Windows, то расстраиваться по этому поводу не надо. Кластер можно поднять и под Windows, причем трудозатраты на установку коммуникационной среды будут такими же как и в варианте с UNIX, то есть небольшими. Основная ваша трудность будет заключаться в том, чтобы научиться писать параллельные программы.

Однако, следует заметить, что подавляющее большинство более-менее серьезных кластеров в мире работает все же в среде UNIX. Разбор преимуществ и недостатков того или иного семейства операционных систем выходит за рамки рассматриваемой нами темы. Поэтому я предлагаю просто поверить мне на слово, что лучшим выбором для вас будет Unix (Linux в частности).

> ***Важное замечание.*** *Поскольку библиотеки для параллельных вычислений MPICH/MPI являются кроссплатформенными, то выбор операционной системы (Windows vs Linux) не важен. Однако следует учесть тот факт, что Linux является заметно менее ресурсоемкой системой. Например при использовании PelicanHPC GNU Linux система занимает в оперативной памяти не более 40Мб! Вся остальная память доступна параллельной программе. Это очень важный фактор в том случае, когда кластер используется с целью моделирования процессов на как можно более подробной сетке.*

Хочется отметить одни немаловажный аспект, проявляющийся при попытке перенести свою работу из Windows в Linux. Имеются в виду психологический и административный факторы. Человек, приходящий в мир Linux, испытывает чувство растерянности и неуверенности в том, что он сможет найти в новой системе привычные для него инструменты. Это, как если бы, человек с детства говорящий только на русском языке, выехал за границу. Кроме того, если вы примите решение о строительстве кластера на базе имеющегося у вас компьютерного класса, который используется в обучении студентов, вам неизбежно придется в той или иной степени менять учебный процесс.

Что касается психологии, то давно прошли те времена, когда работа в UNIX была уделом компьютерных гуру, разговаривающих на непонятном языке и пишущих программы в машинных кодах. Современный уровень развития Linux позволяет чувствовать себя пользователю не менее комфортно, чем в Windows. Более подробно об этом можно прочитать **в этой статье**. Статья посвящена несколько другой теме, но представление о том, что вас ждет в мире Linux, вы получить сможете.

В настоящее время основной операционной системой, используемой при проведении учебных занятий в вузах, является операционная система Windows. При всех достоинствах системы ей присущи некоторые недостатки, существенно затрудняющие ее использование. К таким недостаткам можно отнести:

- малую защищенность системы от неквалифицированных действий пользователей (студентов);
- подверженность системы различного рода "взломам" при сетевом использовании и подверженность вирусам;
- неустойчивость работы системы, проявляющаяся в зависаниях и потере информации;
- большая стоимость лицензий на использование систем;
- закрытость операционной системы, затрудняющая написание учебных программ в ее среде и обучение;
- большие требования к возможностям компьютера (память, быстродействие);
- частая смена версий ОС (примерно каждые два года).

По поводу ломки учебных планов можно сказать следующее. Решение построить кластер на базе вычислительных мощностей имеющегося в наличии компьютерного класса может быть принято скорее всего на физических или математических факультетах ВУЗов. Целью обучения студентов на этих факультетах не ставится подготовка квалифицированных секретарей-референтов со знанием компьютера. Все же остальные цели компьютерной практики вполне могут быть достигнуты и при использовании операционной системы Linux. Жесткая ориентация на продукты фирмы Майкрософт в действительности не обоснована нуждами учебного процесса. Кроме того, ОС Linux вполне может сосуществовать с Windows на одном и том же компьютере и загружаться только по мере необходимости. Другое дело, что проблемой может встать наличие лаборантов, имеющих достаточный уровень квалификации в Linux.

Использование Linux в качестве базовой операционной системы в учебных классах кроме возможности построения кластера из имеющихся в классе компьютерах позволит:

- более эффективно использовать имеющиеся вычислительные средства
- снизить затраты на обслуживание всей системы (благодаря возможностям к гибкой настройки и четкого отслеживания прав доступа различных пользователей)
- решить проблемы с необходимостью приобретения лицензий на используемое ПО
- сделать работу компьютеров в сети и работу всего класса более надежной и устойчивой

Варианты построения кластера

После всего того, что вы здесь прочитали, вы уже, вероятно, имеете представление о том, что такое есть кластерный суперкомпьютер и как его можно собрать из подручных материалов. Осталось только дать несколько практических рекомендаций. Рассмотрим самые простые варианты быстрой и безболезненной установки кластера. Для этого нам потребуется

несколько машин (минимум две), связанные в сеть, пропускной способностью не менее 100Мбит.

Вариант первый

Допустим у нас имеются несколько машин, работающих под управлением операционной системы Windows (не ниже NT4). В этом случае для ввода в строй кластера необходимо и достаточно загрузить из интернета windows-вариант системы MPI и установить ее на всех будущих узлах кластера. Аналогичные по смыслу телодвижения потребуются и в том случае, когда у вас есть несколько Linux-машин. Случай кластера, состоящего из Linux-Машин, описан в разделе Ubuntu-кластер.

Вариант второй

На машины, которые будут у вас узлами кластера, устанавливается Linux. Для установки используется дистрибутив общего назначения, например ALT Linux или Ubuntu. После установки системы и настройки сети, на узлы будущего кластера устанавливается UNIX-вариант системы MPI. Этот вариант хорош еще тем, что избавляет нас от необходимости решать проблему, связанную с переходом на незнакомую операционную систему. Например, если кластер организуется на базе компьютерного класса, используемого в учебном процессе ВУЗа, нет необходимости перестраивать учебные планы с учетом использования Linux в качестве базовой операционной системы. Установка ОС Linux позволяет применять мультизагрузку. То есть имеется возможность не уничтожать ранее установленную на компьютере операционную систему, а использовать обе системы совместно. Linux и кластер может загружаться только в определенное время, например на ночь.

Вариант третий

Одной из ведущих мировых Linux-компаний Mandrake выпускается специализированный дистрибутив CLIC. Этот дистрибутив предназначен для создания высокоэффективного кластерного суперкомпьютера, основанного на сети рабочих станций. После установки дистрибутива все необходимое матобеспечение для работы и мониторинга кластера уже будет установлено в вашей системе. Подробные инструкции по установке и настройке кластера CLIC находятся на диске дистрибутива. Замечу, что кластер CLIC является профессиональным решением, поэтому его установка и настройка наиболее тяжела, по сравнению со всеми другими приводимыми мною вариантами.

Вариант четвертый

Этот вариант самый простой. Про него читаем здесь. Вкратце: мы не устанавливаем кластер, мы загружаем его с CD-ROM и работаем!

Вариант пятый

Этот способ установки кластера немного сложнее предыдущего, но позволяет получить полномасштабный выделенный кластер за очень короткое время. Как и в предыдущем варианте, в данном случае используется специализированный дистрибутив: Rocks Cluster Distribution. Для установки кластера потребуются два диска: загрузочный *Rocks Base* и дополнительный *HPC Roll*. Эти два диска позволяют установить кластер в минимально необходимой конфигурации. Процедуру установки кластера Rocks мы рассмотрим в следующих разделах. В дополнение к минимальной конфигурации, дистрибутив кластера содержит компакт-диск *Intel Roll*, позволяющий установить MPI, интегрированный с оптимизирующим компилятором языка FORTRAN 95 от компании Intel. Так же полезным может оказаться дополнение, находящееся на компакт-диске *PBS/Maui Roll*. Пакет PBS является программным средством для управления очередью заданий, запускаемых на кластере. Управление очередью заданий может оказаться необходимым компонентом кластера в случае, когда он интенсивно используется большим количеством пользователей. Пакет PBS решает вопросы распределения процессорного времени кластера между

пользовательскими программами и очередности их выполнения.

Вариант шестой

После того, как мы попробовали несколько вариантов установки, мы можем сформулировать требования, предъявляемые к полномасштабному кластеру, обеспечивающие простоту установки и использования.

- Установка кластера должна занимать минимальное время
- Кластер должен позволять установку на имеющуюся операционную систему
- Кластер должен использовать сетевую файловую систему для обеспечения синхронизации доступа к программам и данным пользователя
- Узлы кластера должны загружаться в автоматическом режиме, без участия пользователя
- Узлы кластера должны (могут) быть бездисковыми
- Для минимизации объема требуемой под систему оперативной памяти узлы кластера должны использовать корневую файловую систему, расшаренную посредством NFS
- Подключение к кластеру дополнительных узлов должно происходить без перенастройки системы

Этим условиям удовлетворяет проект ThinCluster. Для организации кластера с помощью этого проекта необходимо на имеющуюся Linux-систему определенным образом установить необходимые пакеты (NFS,DHCP,MPICH,TFTP,gcc). Подробное описание процедуры установки ThinCluster описано на странице проекта. При подготовке к семинару по кластерным технологиям, проведенному в октябре месяце 2004 года в Волгоградском государственном университете, мне пришлось подготовить свой ноутбук для работы в качестве сервера кластера ThinCluster. Для этого мне понадобилось около 40 минут времени.

Резюме

Таким образом, мы видим, что создание кластерного суперкомпьютера технически достаточно простая процедура. В случае применения четвертого варианта, вполне работоспособный кластер может быть создан в течение 10 минут. Четвертый вариант хорош еще тем, что он идеально подходит для первоначального знакомства с кластерными технологиями. Возможно BCCD будет не всегда удобен, вполне вероятно возникнет мысль установить кластер на нормальной системе, но, тем не менее, с его помощью можно молниеносно быстро получить из разрозненных машин полнофункциональный кластер и решать на нем нормальные, не тестовые задачи.

Шестой вариант кластера идеально подходит для случая, когда необходимое оборудование уже имеется. В этом случае, полномасштабный кластер так же может быть развернут в минимальные сроки. Фактически для этого понадобится всего несколько часов.

Ubuntu кластер

***** ЧТО ТАКОЕ UBUNTU?**

Ubuntu - это разрабатываемая сообществом, основанная на ядре Linux операционная система, которая идеально подходит для использования на персональных компьютерах, ноутбуках и серверах. Она содержит все необходимые программы, которые вам нужны: программу просмотра Интернет, офисный пакет для работы с текстами, электронными таблицами и презентациями, программы для общения в Интернет и много других.

Дистрибутивы семейства Ubuntu являются комплектом специально подобранного ПО. Это операционная система плюс множество нужных и полезных программ, которые свободно и без ограничений могут быть использованы вами и вашими друзьями и коллегами. Базовая

полнофункциональная версия системы для персональных компьютеров и ноутбуков или версия для серверов занимают всего 1 CD, поэтому вы легко можете получить её для себя и делиться с окружающими. Специальные DVD версии дистрибутива включают практически всё ПО, которое вам может понадобиться для решения любых повседневных задач.

***** ПОДГОТОВКА КЛАСТЕРА**

Для установки линукс-кластера на основе дистрибутива Ubuntu вам потребуется установочный диск Ubuntu Desktop Edition, с помощью которого вы установить операционную систему на консоль (главный компьютер) кластера. С того же самого диска нужно будет установить операционную систему на вычислительные узлы кластера. Однако, если вычислительные узлы не предполагается использовать в качестве офисных машин, но только как узлы кластера, то на эти компьютеры правильнее будет установить серверную редакцию дистрибутива - Ubuntu Server Edition, не содержащую графического интерфейса, и соответственно более легкую и быструю.

Сеть кластера следует спроектировать так, чтобы все узлы имели доступ в интернет. Это нужно для большего комфорта при настройке кластера и установки необходимого программного обеспечения на его узлах. Дело в том, что установка ПО в Ubuntu выполняется посредством закачки новейших версий необходимых пакетов из внешних репозиториев. В действительности достаточно будет обеспечить выходом в Интернет только главный компьютер (консоль кластера), а для вычислительных узлов необходима только возможность загружать программные пакеты через прокси. Осуществить это можно, установив на консоли кластера пакет **apt-cacher-ng**, а на вычислительных узлах сконфигурировать менеджер пакетов **apt-get** для работы через прокси. Для этого на консоли кластера выполним команду:

sudo apt-get install apt-cacher-ng

После этого на всех остальных узлах выполним команды:

```
sudo sh -c "echo \"Acquire::http { Proxy \\\"http://192.168.1.1:3142\\\"; }; \">
/etc/apt/apt.conf.d/01proxy"
sudo apt-get update
```

Здесь мы предполагаем, что адрес 192.168.1.1 - это адрес консоли кластера во внутренней сети кластера.

После установки базовой операционной системы необходимо доустановить необходимое для функциониования кластера по: компиляторы, библиотеки и программы OpenMPI, SSH, NFS. Выполнить это можно с помощью следующих команд:

```
sudo apt-get install mc build-essential fort77 gfortran libstdc++5 libltdl7-dev
sudo apt-get install openssh-server nfs-kernel-server
sudo apt-get install openmpi-bin openmpi-doc
```

Данная последовательность команд устанавливает в систему OpenMPI и компиляторы из набора Gnu Compiller Collection (gcc). Компиляторы gcc обладают одним существенным (в некоторых случаях) недостатком. В них нет поддержки работы с типами данных REAL*16 и COMPLEX*32 (в терминологии Фортрана). Если для ваших задач необходима такая точность вычислений, то вы вместо стандартного набора компиляторов и пакета OpenMPI из состава дистрибутива должны будете установить компиляторы фирмы Intel и скомпилировать OpenMPI с поддержкой этих компиляторов. Как это сделать - рассказано в следующих двух параграфах этой статьи. ***Если же точность REAL*8 и COMPLEX*16 вас устраивает, то следующие два параграфа вы можете пропустить.***

ВАЖНО! Если вы решили использовать компиляторы фирмы Intel, то не следует устанавливать OpenMPI из состава дистрибутива. То есть команду ***sudo apt-get install openmpi-bin openmpi-doc*** (см. выше) выполнять не следует.

***** УСТАНОВКА КОМПИЛЯТОРА INTEL FORTRAN**

Скачать свежую версию компилятора можно с официального сайта компании Intel, с этой

страницы. Получив и разархивировав в своем домашнем каталоге архив с компилятором, выполним процедуру установки, запустив скрипт **install.sh**, который находится в каталоге, получившимся после разархивации. При запуске скрипт спросит вас о том, в каком варианте произвести установку компилятора. Нужно выбрать вариант N3 - установку в пользовательский каталог. Для ответа на все дальнейшие вопросы программы следует выбирать вариант, предложенный программой по умолчанию.

После окончания установки, чтобы все заработало необходимо поправить некоторые переменные окружения. Для этого исправим файл **.bashrc**, находящийся в домашнем каталоге пользователя, используя следующую команду (не забыв после переподключиться в систему):

```
echo ". $HOME/intel/Compiler/11.1/046/bin/ifortvars.sh ia32" >> $HOME/.bashrc
```

Эта команда построена в предположении, что используется версия 11.1.046 компилятора. Если же к тому моменту, когда вы воспользуетесь данными рекомендациями, фирма Intel выпустит новую версию, то команду надо будет поправить соответствующим образом.

*** УСТАНОВКА OpenMPI ДЛЯ КОМПИЛЯТОРА INTEL

К сожалению пакет OpenMPI, имеющийся в дистрибутиве Ubuntu, не знает о существовании компилятора Intel. Поэтому его придется установить из исходников, которые можно загрузить с официального сайта проекта OpenMPI. Разархивировав загруженный файл, перейдем в полученный в результате этого каталог исходников и выполним процедуру установки, последовательно запустив на исполнение следующие команды:

```
./configure --prefix=$HOME/openmpi F77=ifort FC=ifort
make
make install
```

В случае, если вы намереваетесь использовать и установили кроме компилятора Intel Fortran, так же и Intel C++, то команда конфигурации должна будет выглядеть так:

```
./configure --prefix=$HOME/openmpi CC=icc CXX=icpc F77=ifort FC=ifort
```

По окончании процесса установки, как и в случае с компилятором, необходимо поправить некоторые переменные окружения. Делаем это с помощью следующих команд модифицируем файл **.bashrc**:

```
echo "PATH=$HOME/openmpi/bin:\$PATH" >> $HOME/.bashrc
echo "export PATH" >> $HOME/.bashrc
echo "LD_LIBRARY_PATH=$HOME/openmpi/lib:\$LD_LIBRARY_PATH" >> $HOME/.bashrc
echo "export LD_LIBRARY_PATH" >> $HOME/.bashrc
```

По окончании процедуры необходимо переподключиться в систему (перелогиниться).

*** НАСТРОЙКА СЕТИ КЛАСТЕРА

Для того, чтобы MPI имела возможность без дополнительных запросов запускать части параллельной прогаммы на вычислительных узлах кластера, вы должны будете организовать беспарольный доступ по протоколу SSH с консоли кластера на все вычислительные узлы. Как это сделать - читайте соответствующий раздел этого сайта.

Следующим нашим шагом будет организация общего для всех узлов кластера каталога, где будут храниться исходники параллельных программ, их исполняемые модули и файлы с данными. Сделать это можно, настроив сетевую файловую систему NFS. Для простоты общий каталог сделаем подкаталогом домашней директории пользователя. Назовем его к прмеру "mpi".

Предположим, что узлы кластера имеют адреса внутри сети 192.168.1.0/255.255.255.0, и консоль кластера имеет адрес 192.168.1.1.

На консоли кластера добавляем строчку в файл **/etc/exports** для создания сетевого ресурса:

```
sudo sh -c "echo \"$HOME/mpi 192.168.1.0/24(rw,async,no_subtree_check,no_root_squash)\" >>
/etc/exports"
```

После этого необходимо перезапустить сервис NFS:

```
sudo /etc/init.d/nfs-kernel-server restart
```

На всех остальных узлах кластера мы должны будем поправить файл **/etc/fstab**, добавив туда запись для автоматического монтирования сетевого ресурса:

```
sudo sh -c "echo \"192.168.1.1:$HOME/mpi $HOME/mpi nfs rsize=8192,wsize=8192,rw,user,owner,auto 0
0\" >> /etc/fstab"
```

Теперь для проверки перезагрузим вычислительные узлы кластера. После перезагрузки каталог mpi в домашней директории пользователя должен быть смонтирован и его содержимое должно быть эквивалентно содержимому такого же каталога на консоли кластера.

Если вы решили использовать компилятор Intel Fortran/Intel C++ и, соответственно, на консоли кластера установили этот компилятор и OpenMPI, скомпилированный из исходников, то то же самое вы должны сделать на всех вычислительных узлах. Однако последнее - не обязательно. Вместо этого вы можете подключить по сети (через NFS) каталоги с установленными компилятором и OpenMPI (openmpi и intel) аналогично тому, как вы только что это сделали подключая рабочий каталог mpi. В дополнение к этому вам необходимо подкорректировать на всех вычислительных узлах кластера файл **/etc/environment** следующими командами:

```
sudo sed -i -e
's;PATH=";PATH="'"$HOME"'/openmpi/bin:'"$HOME"'/intel/Compiler/11.1/046/bin/ia32:;' /etc/environment
sudo sh -c "echo \"LD_LIBRARY_PATH=\\\"$HOME/openmpi/lib:
$HOME/intel/Compiler/11.1/046/lib/ia32:$HOME/intel/Compiler/11.1/046/lib/32\\\"\" >>
/etc/environment"
```

*** НАСТРОЙКА ПАРАЛЛЕЛЬНОЙ СРЕДЫ

Настройка OpenMPI максимально проста. Вы должны в любом удобно месте создать текстовый файл (назовем его например **mpi.hosts**), в который запишете список всех узлов вашего кластера, как вычислительных узлов, так и консоли. Для простоты будем считать, что вы перечислите там ip-адреса узлов. Один узел - одна строчка. В самом простом варианте конфигурации кластера этого будет достаточно для начала работы.

*** КОМПИЛЯЦИЯ ПАРАЛЛЕЛЬНОЙ ПРОГРАММЫ

Для примера и для проверки работоспособности кластера скачаем в каталог mpi тестовую программу вычисления числа π flops.f. Для компиляции необходимо выполнить следующую простую команду:

```
mpif77 flops.f -o flops
```

В результате должен получится исполняемый файл с именем "flops". Для запуска его в локальном однопроцессорном режиме достаточно запустить этот файл на исполнение обычным образом, то есть выполнить команду **./flops**. На выходе мы получим информацию о скорости работы программы. Если все прошло нормально, то можем переходить к тестированию кластера.

*** ЗАПУСК ПАРАЛЛЕЛЬНОЙ ПРОГРАММЫ

Для запуска нашей программы параллельно на нескольких процессорах (узлах кластера) необходимо использовать диспетчер задач MPI или менеджер очередей, например Torque. Про менеджер очередей и ресурсов Torque мы поговорим позже. Пока же обойдемся без него.

Нам понадобится созданный нами ранее файл-список узлов **mpi.hosts**. С помощью этого списка запуск нашей параллельной программы осуществляется командой:

```
mpiexec -hostfile mpi.hosts -np 5 ./flops
```

Параметр **-np 5** означает, что программа будет запущена на пяти узлах кластера. Отработав, программа выдаст на экран информацию о скоростных характеристиках кластера в целом и отдельных его узлов.

В случае, если ваша параллельная программа состоит из отдельных программных модулей, то имеется возможность запускать из вместе, как единый комплекс. Предположим, что ваша программа состоит из двух модулей - visio и calc. Модуль calc - это собственно параллельная программа, которая в многопроцессороном режиме обрабатывает большой массив данных, и которая занимает 5 процессоров кластера. Модуль visio - программа визуализации полученных данных. Программа visio должна работать на одном процессоре. Запуск такой комплексной программы будет выглядеть так:

```
mpiexec -hostfile mpi.hosts -np 1 ./visio : -np 5 ./calc
```

ROCKS Cluster

Еще одним вариантом специализированного Linux-дистрибутива, ориентированного на создание высокопроизводительного кластера, является Rocks Cluster Distribution. Одним из преимуществ использования Rocks является простота построения и управления кластером. Одно из важных свойств данного дистрибутива является возможность построения кластера как с использованием полнокомплектных компьютераов, так и с использованием бездисковых станций.

Дистрибутив Rocks прострен на основе Red Hat Linux, и следующее оборудование:

Процессоры

- x86 (IA32, AMD, и пр.)
- IA64 (Itanium, McKinley, и др.)
- x86-64 (AMD Opteron)

Сетевое оборудование

- Ethernet (все устройства, поддерживаемые Red Hat, включая Intel Gigabit Ethernet)
- Myrinet (Lanai 9.x)

Физически Rocks кластер состоит из двух или более узлов одного из следующих типов:

Frontend

Узлы этого типа предназначены для доступа к кластеру пользователей либо извне локальной сети (через интернет), либо непосредственно с консоли. То есть они выступают в роли консоли кластера, к которым пользователи имеют возможность подключиться (login), откомпилировать на них свои программы и запустить задания на исполнение. На узлах данного типа запущено большое количество сервисов (NFS, NIS, DHCP, NTP, MySQL, HTTP, ...). Администрирование таких узлов требует от сисадмина определенного уровня компетенции. Как минимум один frontend-узел должен присуствовать для нормального функционирования кластера.

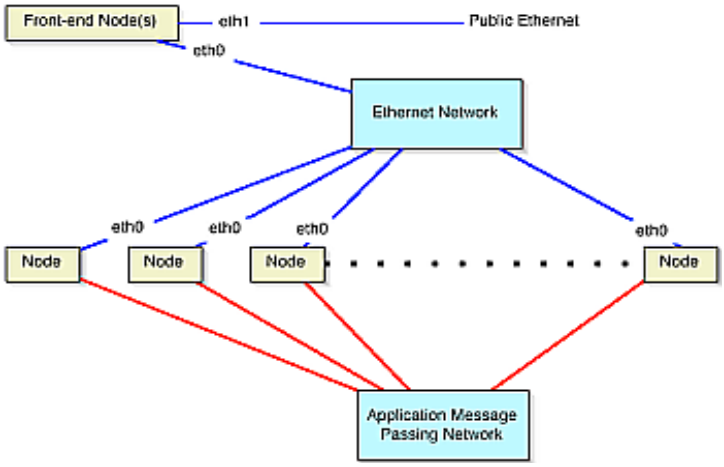
Обычно узлы данного типа имеют следующие характеристики:

- Две сетевых карты - одна для приватной сети кластера, другая для подключения к внешним сетям.
- Большой объем дискового пространства для для хранения пользовательских программ и данных.

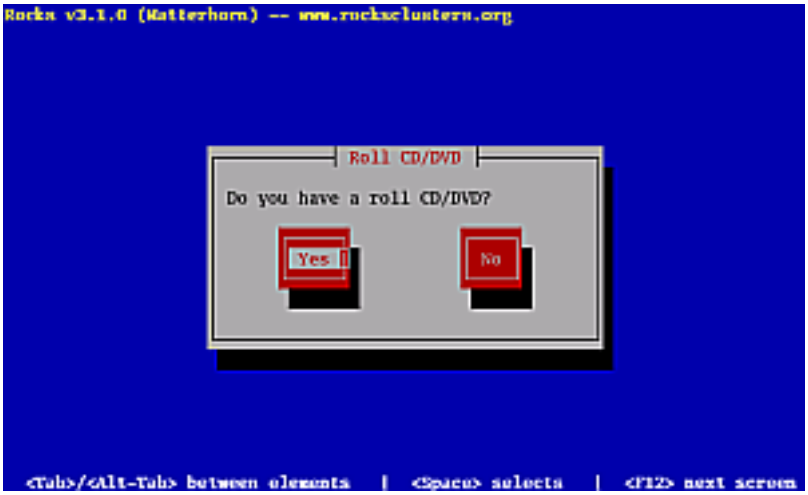
Compute

Узлы этого типа - рабочие лошадки кластера. Дистрибутив позволяет установить на такие узлы полноценную ОС, причем в максимально автоматическом режиме за очень короткое время (порядка 10 минут). Другой вариант compute-узлов - бездисковые станции. В этом случае загрузка системы выполняется по сети (PXE) с frontend-узла.

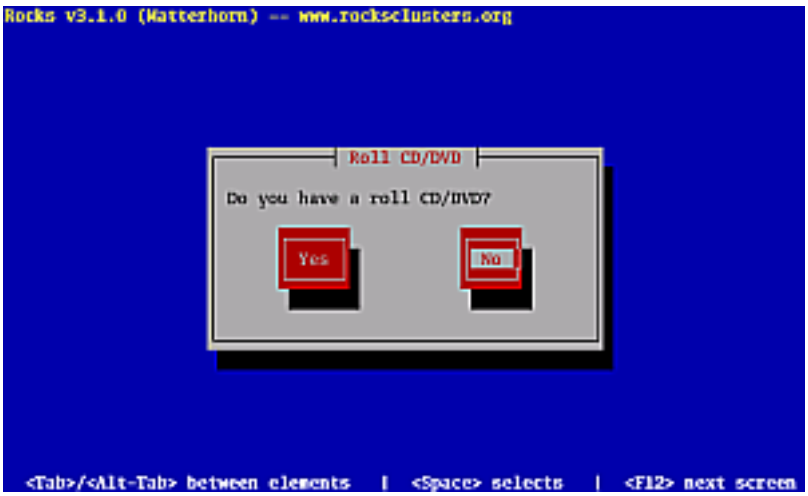
Архитектура Rocks-кластера диктует следующую схему подключения узлов к сети:



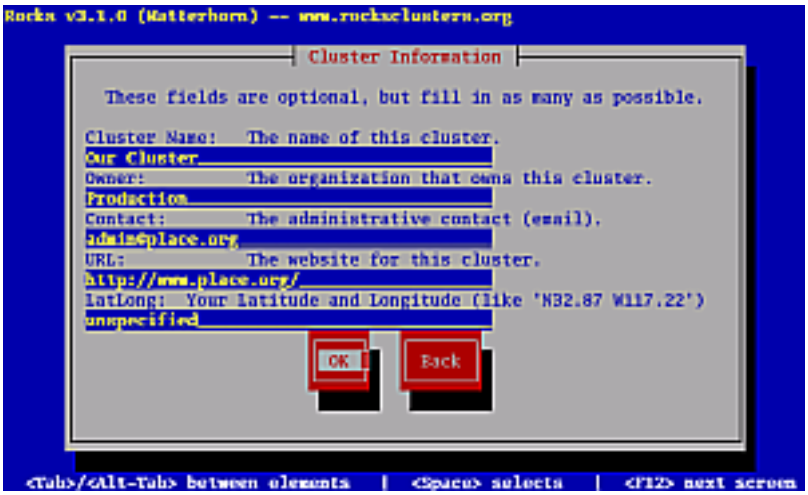
Далее рассмотрим процедуру установки кластера, которая начинается с установки операционной системы на frontend-узел. Для установки необходимы два CD: **Rocks Base CD** и **HPC Roll CD**. Загрузка начинается с диска Rocks Base CD. При появлении начального логотипа вы должны указать способ загрузки: **frontend**. Обратите внимание, режимом загрузки по умолчанию является режим compute, причем время паузы перед загрузкой достаточно мало, поэтому не пропустите предложение выбрать режим. После того, как программа установки frontend-узла будет запущена, через некоторое время на экране появится следующий вопрос:



Чтобы добавить необходимый для frontend-режима пакет HPC Roll, ответьте на этот вопрос "Yes", и после того, как ваш CD-привод откроется, поместите в него HPC Roll CD. Для установки HPC Roll нажмите "Yes", и когда диск опознается системой вы увидите окошко с надписью **Found Roll 'hpc'**. Далее, когда система предложит вам установить дополнительный пакет, ответьте отказом:

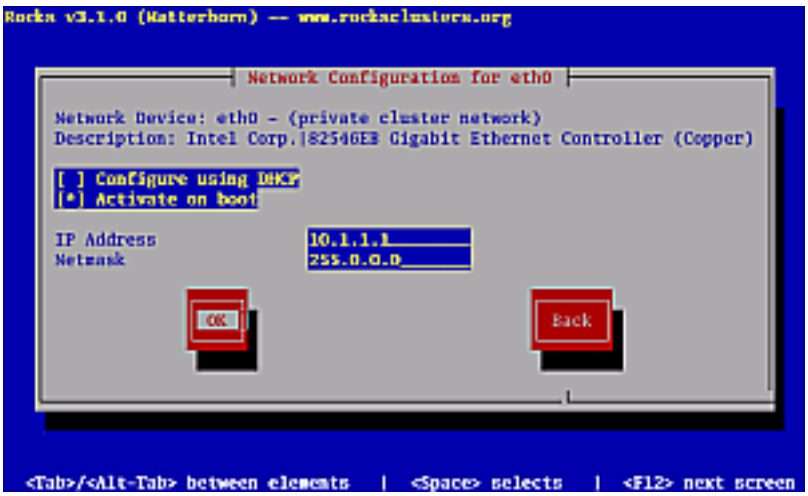


После этого система попросит вас вставить в CD-привод исходный пакет **Rocks Base CD**. Сделайте это и нажмите "Ok". Далее система попросит вас заполнить форму информации об устанавливаемом кластере:

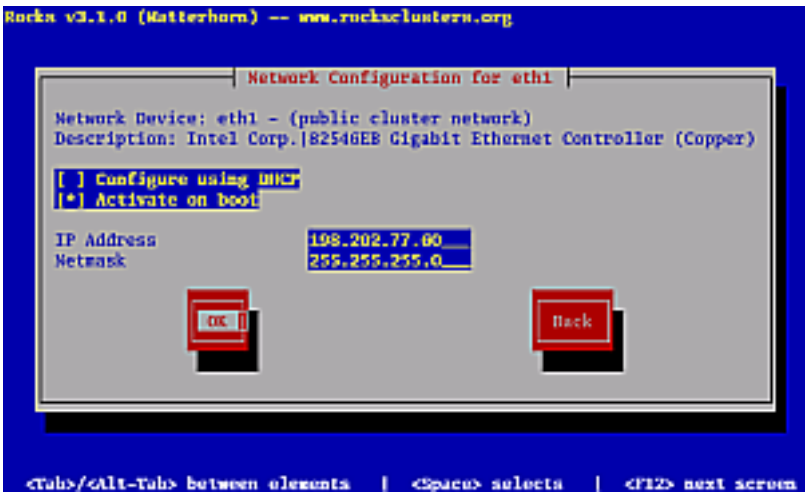


Заполните ее и нажмите "Ok". Следующий этап установки кластера - разбиение вашего жесткого диска на разделы. Вы можете эту работу выполнить самостоятельно или доверить ее процедуре установки. В последнем случае все необходимые разделы будут созданы и отформатированы автоматически.

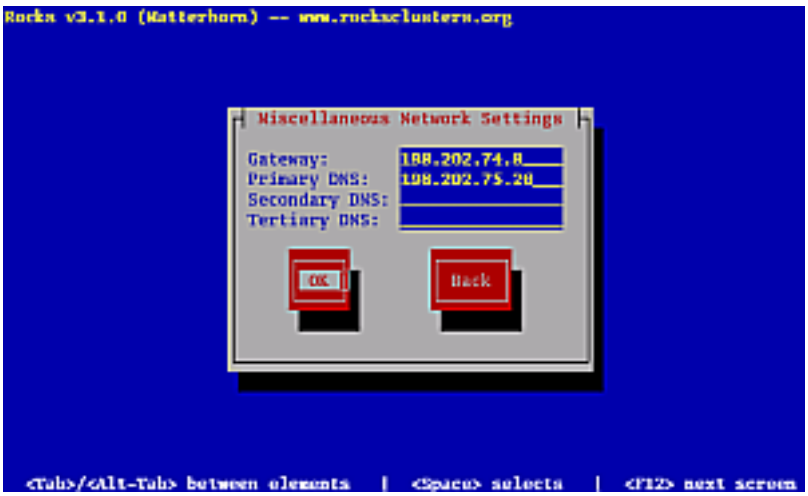
Обычно имеет смысл для простоты работы выбрать автоматический режим. В этом случае мы выбираем пункт меню "**Autopartition**". Далее мы перейдем к определению конфигурации сетевых интерфейсов.



В случае, если у нас имеется две сетевых карты, как это было описано выше, предыдущая картинка у вас появится дважды. Превый раз мы определяем сетевый реквизиты приватной сети. Реквизиты могут быть заданы как вручную, так и автоматически с помощью имеющегося DHCP-сервера. Для приватной сети рекомендуется ручной режим. После того, как мы определили ip-адрес и маску подсети для приватной сети, аналогично задаем сетевые реквизиты для второй сетевой карты (для публичной сети), то есть для карты, через котроую будет осуществлятся связь кластера с внешним миром.



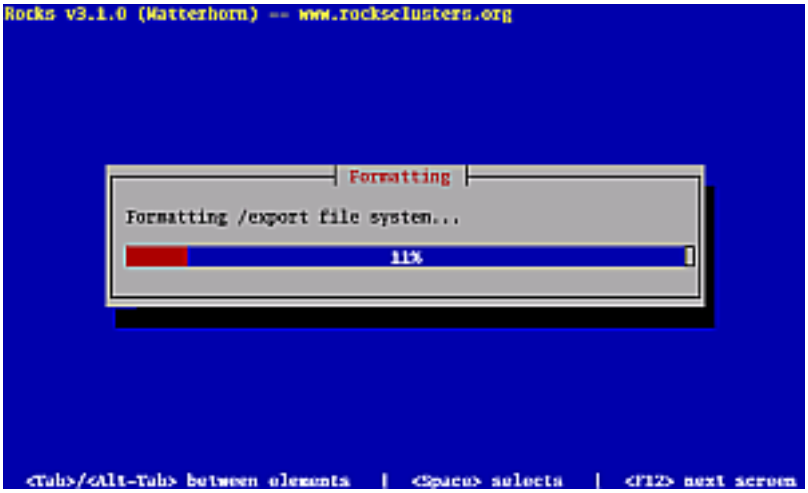
После того, как мы задали ip-адреса сетевых карт, указываем системе адрес шлюза и DNS-сервера:



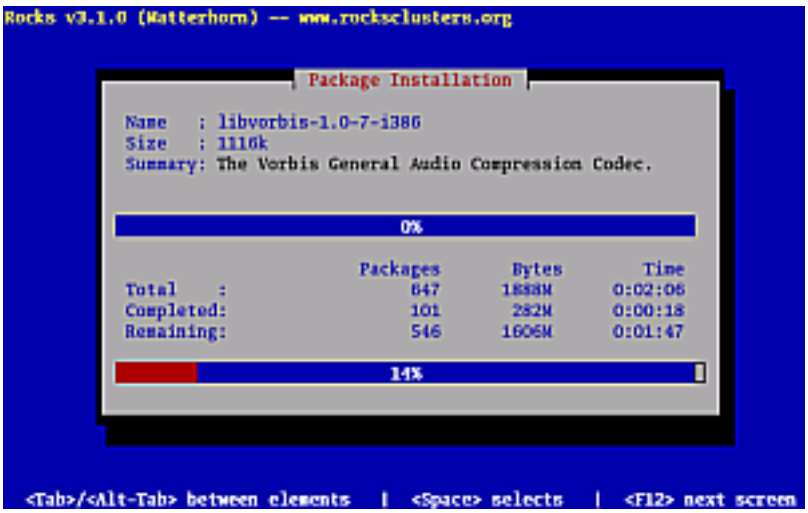
И, напоследок задаем доменное имя нашей frontend-машины:

Далее определяем пароль суперпользователя:

Теперь все основные параметры заданы и система начинает форматировать винчестер:



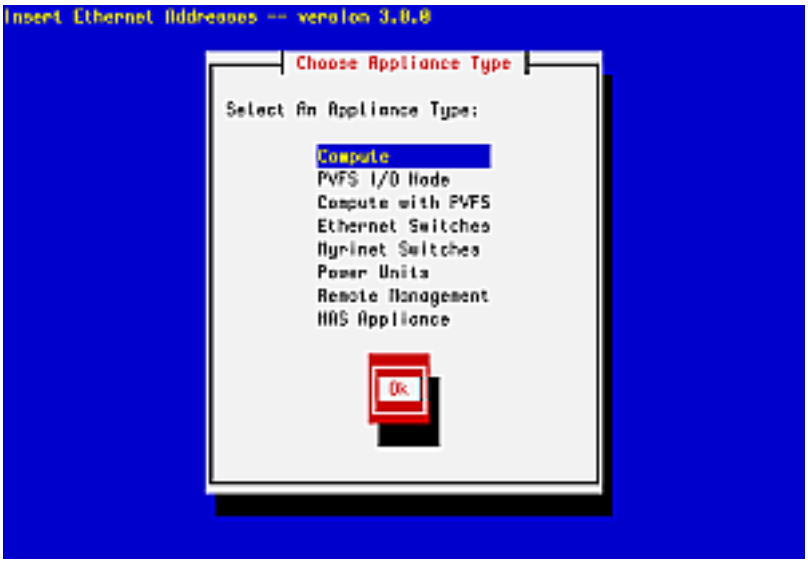
После успешного разбиения диска на разделы и форматирования разделов начинается установка пакетов:



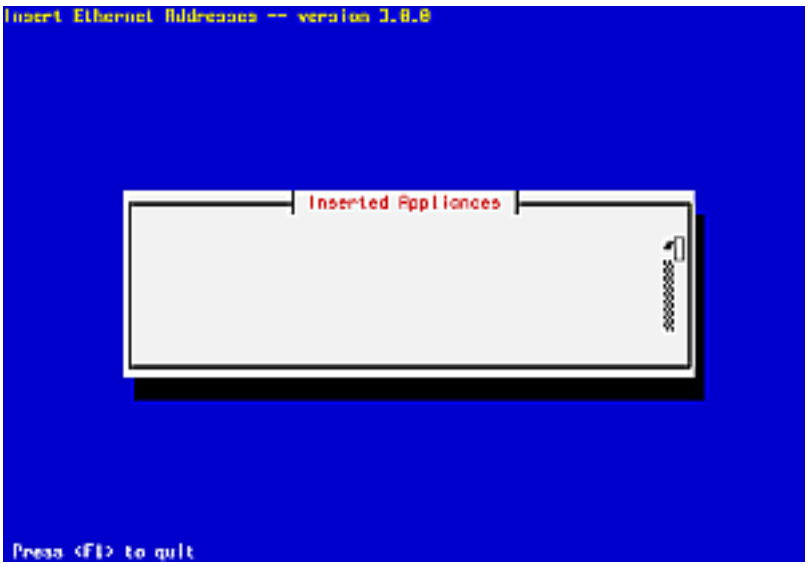
После установки пакетов с первого диска инсталлятор попросит вас вставить те дополнительные диски пакетов, которые вы выбрали в самом начале. После того, как все диски будут установлены, система перезагружается. На этом установка frontend-узла завершается.

Переходим к установке compute-узлов. В самом простом варианте дополнительные узлы кластера у нас будут бездисковыми. В качестве альтернативы может рассматриваться вариант, когда в качестве узлов кластера используются имеющиеся рабочие станции с установленной операционной системой (например Windows). В этом случае загрузка таких компьютеров осуществляется не обычным образом, а по сети (PXE), если это позволяют сетевые карты или с дискеты, образ которой, соответствующий имеющейся в компьютере карте, можно взять с сайта www.rom-o-matic.net. В случае, если вы не хотите использовать удаленную загрузку системы, а, вместо этого, желаете установить на compute-ноды нормальную ОС, процедура установки будет аналогична описанной выше.

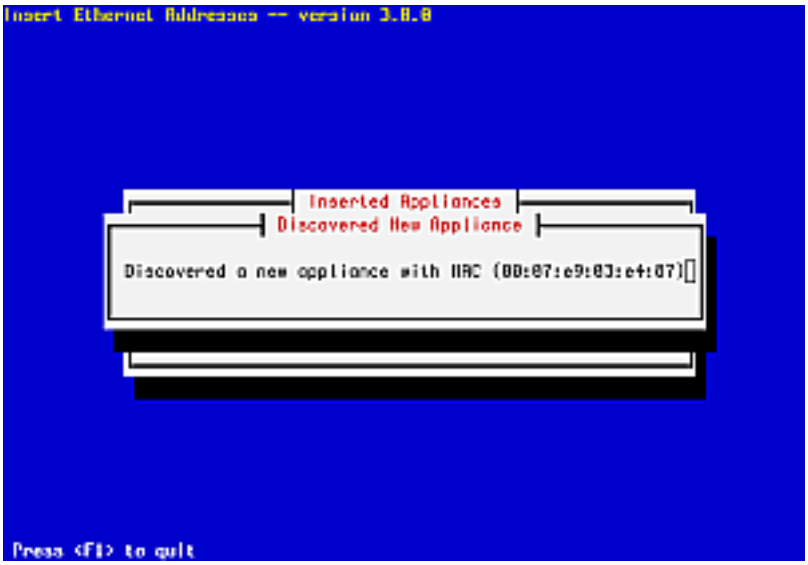
Итак, устанавливаем бездисковые compute-ноды. Для этого мы должны подключиться к только что установленному нами frontend-узлу под именем **root**. Из командной строки запускаем программу **insert-ethers**. Эта программа перехватывает DHCP-запросы от загружаемых compute-нод и сохраняет информацию о них в базе данных Rocks. Следующий скриншот показывает картинку, которая появится у вас в результате запуска insert-ethers.



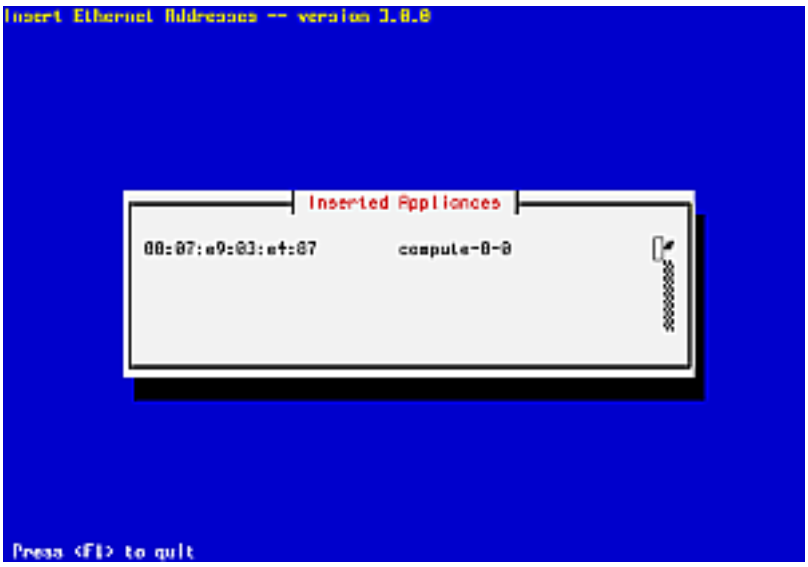
Выберите пункт меню по умолчанию (Compute) и нажмите "OK". В результате вы увидите такую картинку:



Это означает, что программа ждет появления в сети новых compute-узлов. Включите первую бездисковую станцию, сконфигурировав ее для удаленной загрузки по сети. В тот момент, когда frontend получит DHCP-запрос от compute-узла вы увидите на экране картинку примерно следующего содержания:



Эта картинка означает, что insert-ethers приняла DHCP-запрос от compute-ноды, внесла полученную информацию в базу данных и обновила конфигурационные файлы. Картинка продержится на экране несколько секунд, после чего сменится на примерно следующее:



В этот момент начинается установка операционной системы на compute-узел. Аналогичную процедуру проведите для всех остальных compute-узлов. После того, как все узлы кластера будут установлены, выйдите из программы insert-ethers, нажав "F1".

Специализированные дистрибутивы

В этом разделе дан обзор специализированных Linux-дистрибутивов и пакетов программ, предназначенных для быстрого развертывания кластера.

PelicanHPC GNU Linux

PelicanHPC - это модификация хорошо известного Linux-дистрибутива Debian Live, которая позволяет установить кластер компьютеров для выполнения параллельных вычислений с использованием LAM-MPI и/или MPICH реализаций интерфейса MPI или параллельных виртуальных машин PVM. Вы можете превратить комнату полную машин, работающих под управлением Windows в Linux-кластер, а после после выполнения работ и перезагрузки ваши Windows-машины оказываются в исходном состоянии, какое было до загрузки кластера. Компьютеры в кластере могут быть как гомогенные, так и гетерогенные. Если сетевые карты имеют возможность осуществлять PXE-загрузку, то запуск и конфигурация кластера занимает около 5 минут. Поддерживаются кластеры, содержащие от 2 до 200 машин. Руководство содержит детальные и пошаговые инструкции по установке и настройке кластера.

Debian Cluster Components

Debian Cluster Components (DCC) - это набор инструментов для максимально простого построения, управления и развертывания высокопроизводительного Linux-кластера. Набор состоит из следующих системных компонентов: инсталляционный набор, C3, система очередей TORQUE, OpenLDAP и Ganglia. Набор Debian-пакетов, установленный в используемой вами Debian-системе, обеспечивает полные функциональные возможности главной консоли кластера.

DCC/Live

DCC/Live - основанный на Knoppix загрузочный CD, который обеспечивает виртуальную инфраструктуру Linux-кластера на единственном компьютере. Консоль кластера (front-node) и три виртуальных вычислительных узла (work-nodes) запускаемых с помощью механизма User Mode Linux доступны пользователю после загрузки системы. Планировщик очередей заданий и другие специфические сервисы кластера предустановлены в систему и не требуют дополнительной настройки. Этот проект главным образом разработан для образовательных

целей. Примеры параллельных программ и инструкции включены в состав CD.

Scalable Cluster Environment

Этот набор программ, названный SCE (Scalable Computing Environment), состоит из инструментального набора для построения кластера, комплексной системы управления (SCMS), масштабируемой системы контроля в реальном времени, Web-ориентированной системы мониторинга (KCAP), параллельных версий Unix-команд и планировщика задач.

ComputeMode

ComputeMode является средством для организации высокопроизводительной сети (HPC grid), которая может быть расширена путем подключения неиспользуемых компьютерных ресурсов. Например, ComputeMode позволяет построить виртуальный кластер, используя в качестве вычислительных узлов офисные компьютеры во время их простоя. На практике, большое количество машин в крупных компаниях простаивают в ночное время, в выходные, во время отпусков и командировок сотрудников. ComputeMode позволяет использовать эти ресурсы в своих интересах.

Warewulf

Warewulf is a Linux cluster implementation tool that builds RAMDISK based filesystems and distributes them to the slave nodes. This facilitates a Central administration model for all nodes and includes tools needed to build configuration files, monitor, and control the nodes. It is totally customizable and can be adapted to just about any type of cluster. The node distributions are built from a virtual node filesystem residing on the master and transfered to the nodes either by Etherboot or CD-ROM images. Administration becomes scalable by having only one point of admin for all nodes. Node filesystems are very easy to customize, which makes Warewulf fit many different solutions. There are also a number of add-on software components (MPI, PVM, and Sun Grid Engine) for Warewulf.

Grid-технология

Давайте отвлечемся от детских игр в конструкторы Лего и посмотрим, что может предложить XXI век для тех, кто серьезно занимается научными исследованиями, требующими совсем немаленьких вычислительных ресурсов.

За последние годы прогресс в вычислительных технологиях достиг впечатляющего уровня. Тем не менее, можно сказать, общее мнение таково, что экстенсивное развитие - простое увеличение мощности процессоров, емкости накопителей, пропускной способности каналов - уже практически исчерпало свои возможности. Необходим качественный, концептуальный скачок в вычислениях.

В настоящее время технологическая основа такого скачка существует. Развитие мощных компьютеров сегодня идет по пути параллельных вычислений. Помимо "классических" монолитных высокопроизводительных систем, бурное развитие получили кластеры, состоящие из однотипных вычислительных единиц, строить которые мы уже научились. Следующим шагом стало появление Grid-технологии - грубо говоря, создание кластера кластеров, причем в глобальном масштабе

Термин **Computing grid** (Вычислительная сеть, Grid-вычисления), по смыслу аналогичен выражению "единая энергосистема". Суть его заключается в стремлении объединить все компьютеры мира в единую систему - в виртуальный суперкомпьютер невиданной мощности, что позволит распределять и перераспределять ресурсы между пользователями в соответствии с их запросами. Именно так человечество пользуется электричеством единых энергетических сетей. Имея такой суперкомпьютер, пользователь может в любое время и в любом месте запросить столько вычислительных ресурсов, сколько ему требуется для

решения своей задачи.

Первые опыты в области Grid-технологий связаны с расчетами экспериментов в ядерной физике. Считается, что этот опыт вообще стал базой формирования World Wide Web, WWW - Всемирной Паутины. С ним связывают имя Тима Бернеса-Ли. Перед этим ученым была поставлена задача найти способ, который позволил бы ученым, участвующим в экспериментах, обмениваться данными и представлять результаты их обработки на всеобщее обсуждение. Многие физики, большую часть времени находящиеся в своих научных институтах, тоже хотели полноправно участвовать в анализе данных. В 1990 году прототип того, что впоследствии получило название Всемирной Паутины, был создан в CERN, а начиная с 1991 года, первые браузеры и WWW-серверы появились в распоряжении ядерных физиков всего мира. Широкое распространение сразу же получили язык HTML и протокол HTTP.

Однако теоретическое обобщение опыта CERN и развитие идеи WWW в область современного представления о будущей сети Grid, было сделано американскими учеными Яном Фостером и Карлом Кессельманоме. По их представлению, Grid действительно является "надстройкой" над Интернетом, предназначенной для распределенных вычислений при решении задач высокой сложности в области науки и технологий.

Основная идея технологии Grid заключается в использовании каждым входящим в систему компьютером логических и физических мощностей всей сети. Это похоже на локальную сеть с тем отличием, что помимо возможности доступа к данным на других компьютерах новая технология делает возможным объединение ресурсов, находящихся на расстоянии входящих в Grid вычислительных машин, для решения одной задачи. Выгоды такого подхода очевидны: пользователю станут доступны операции, которые в силу программных или аппаратных ограничений не способен производить его компьютер.

Возникает вопрос, а как конкретному научному работнику воспользоваться grid-ресурсами для проведения численных экспериментов? Отвечу: не просто. Однако сложность эта чисто организационная, призванная отсеить праздношатающихся "экспериментаторв" и хакеров всех мастей. Система Grid - это открытая система, которой может воспользоваться любой, буде у него достаточная мотивация.

Литература

1. Илья Евсеев. Использование PVM. Введение в программирование. http://www.csa.ru/~il/pvm_tutor/
2. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam. PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. http://www.netlib.org/pvm3/book/pvm-book.html
3. Евгений Борисов. Вычислительные системы сверхвысокой производительности. http://mechanoid.narod.ru/high_perf/index.html
4. Дмитрий Арапов. Можно ли превратить сеть в суперкомпьютер? http://www.osp.ru/os/1997/04/6.htm
5. Павлов В.М. Программное обеспечение для построения систем распределённых вычислений. Реферат по междисциплинарному государственному экзамену
6. Константин Михайленко. Параллельный стиль. http://www.altlinux.ru
7. Михаил Кузьминский. Кластеры на базе ОС Linux. ComputerWorld 5/1998
8. А.М. Горелик. Средства поддержки параллельности в языках программирования. Открытые системы 02/1995.
9. Андрей Кузнецов. Параллельные миры. http://ccc.ru/magazine/depot/00_08/web3.htm

10. Вл. В. Воеводин. Суперкомпьютерная грань компьютерного мира. http://parallel.ru/vvv/intro2hpc.html
11. Е.С.Борисов. Декомпозиция последовательных программ при параллельных вычислениях. http://mechanoid.narod.ru/parallel/semiautomata/
12. В.Н. Дацюк, А.А. Букатов, А.И. Жегуло. Методическое пособие по курсу "Многопроцессорные системы и параллельное программирование" http://rsusu1.rnd.runnet.ru/tutor/method/index.html
13. А. Кильпио, П. Кайгородов. Параллельные системы, их создание и применение. http://www.xform.ru/library/article/default.asp?I=20
14. Е.С.Борисов. Вычислительные системы сверхвысокой производительности. http://mechanoid.narod.ru/parallel/high_perf/
15. Бутко А.И., Пярнпуу А.А., Донских В.В. ОС Linux в учебном поцессе. http://ict.edu.ru/vconf/index.php?a=vconf&c=getForm&r=thesisDesc&d=light&id_sec=59&id_thesis=1949
16. А.В.Комолкин, С.А.Немнюгин. Программирование для высокопроизводительных ЭВМ. http://www.hpc.nw.ru/COURSES/HPC/
17. Портал Parallel.ru. http://parallel.ru/computers/interconnects.html
18. А.Б. Барский. Параллельное программирование. http://www.intuit.ru/department/se/parallprog/
19. Андрей Карпов. Введение в проблематику разработки параллельных программ. http://www.viva64.com