Кластеры: Легко

Динамический и виртуальный кластеры

Наш эксперт

Андрей Сухарев
Физик, участник экспериментов СО РАН и CERN. Любит свою работу за переплетение с увлечениями — программирование и администрирование Linux-систем.

Наш эксперт

Андрей Грозин
Доктор физ.-мат. наук. Стал разрабочиком Gentoo, сдав 2 письменных экзамена и устный, причем устный — только со второй попытки (последний раз его выгоняли с экзамена лет за 30 до того).

Наш эксперт

Евгений Балдин
Физик, который действительно знает, что такое нехватка вычислительных ресурсов.

Были бы данные, а **Андрей Сухарев** и **Евгений Балдин** кластер для их параллельной обработки соорудить сумеют.

У вас есть данные, которые можно обрабатывать параллельно и независимо? Есть поблизости дружелюбный кластер? Тогда можно еще сильнее упростить себе жизнь и размножить привычное программное окружение сразу на нем.

Введение

Одним из важных классов параллельных вычислений является так называемый портфель задач. Каждая задача в этом случае является независимой единицей, а распараллеливание производится на уровне данных. Это возможно, если данные являются независимыми. Задачи помещаются в портфель и достаются оттуда, как только освобождается необходимый для ее исполнения ресурс. Такие вычисления замечательно масштабируются — надо только как-то суметь захватить доступные компьютерные ресурсы.

В современном мире фактически каждый физик/химик/биолог и даже экономист (если он ученый) должен быть немного информатиком: бурно развивающиеся вычислительные ресурсы расширяют возможности как экспериментатора, так и теоретика. Безусловно не заменяют, но являются весьма ценным дополнением к уже имеющимся инструментам.

Одной из особенностей долго идущих экспериментов является разнородность компьютерной инфраструктуры и постепенно разрастающиеся связи. Пример схемы такого окружения можно увидеть на рис. 1. Там интересны три кружочка в левом нижнем углу — это три независимых суперкомпьютерных кластера, которые можно использовать для своих нужд. И здесь вылезает основная проблема: ПО, как и версии GNU/Linux, на этих вычислительных ресурсах различны. Нет ни времени, ни желания выяснять, а нет ли каких-либо багов, специфичных только для этой коллекции библиотек, поэтому проще принести свое привычное окружение, с уже известными и полюбившимися багами, прямо туда. Также следует учитывать, что кластеры на стороне не являются собственностью исследователей, поэтому как только необходимость в вычислениях отпадает, то ресурсы нужно автоматически освободить. Особенно это важно в случаях, если требуется высокая пиковая производительность, но лишь изредка.

Разберем одну из реализаций того, как можно задействовать эти вычислительные мощности в условиях ограниченных денежных и человеческих ресурсов. Она не обязательно будет оптимальной для вашей задачи, и наверняка есть решения получше, но была успешно собрана, исправно работает и развивается.

Динамический кластер

Естественным решением является виртуализация. Надо только развернуть виртуализованную среду, научиться запускать ее на внешнем вычислительном кластере как задачу тамошней батч-системы и, наконец, автоматизировать весь цикл «пользовательское задание → запуск ВМ → работа ВМ → освобождение ресурсов». Чтобы получить оптимальную конфигурацию виртуализованной среды, способной обеспечить и высокую эффективность использования ЦПУ физических узлов, и стабильную работу виртуальных машин (ВМ), мы изучили три системы виртуализации: *VMware Server* (http://www.vmware.com/products/server/), *Xen* (http://xenproject.org/) и *KVM* (http://www.linux-kvm.org). Для постоянного использования мы выбрали *KVM*. Главным и определяющим ее преимуществом стала простота применения: *KVM*, как правило, уже входит в состав современных дистрибутивов GNU/Linux, а значит, проста в установке. В отличие от *Xen*, *KVM* не требует специального модифицированного ядра GNU/Linux, установка которого на кластер бывает и невозможна — например, из-за конфликта с драйверами InfiniBand. *KVM*-решение прошло широкомасштабное тестирование, в котором до 512 двуядерных ВМ одновременно выполняли до 1024 заданий обработки данных и моделирования нашего домашнего эксперимента КЕДР.

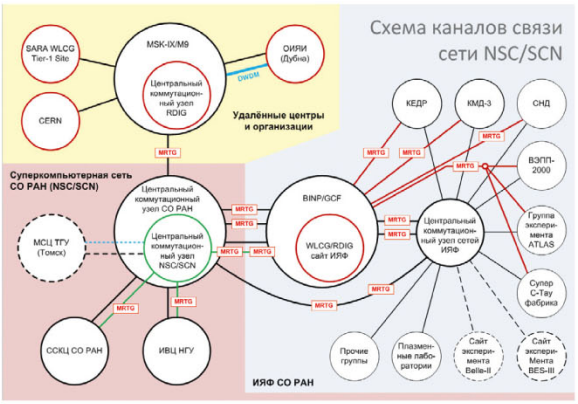

Рис. 1. Структура суперкомпьютерной сети ННЦ. Зелеными линиями показаны основные 10 GbE соединения, а пунктиром — планируемые группы и связи с ними. Отмечены точки сбора статистики.

Общий вид схемы интеграции систем управления пакетной обработкой заданий показан на рис. 2. Специальный сервис периодически проверяет наличие ожидающих выполнения заданий в очереди батч-системы эксперимента. При появлении таких заданий он ставит в очередь системы пакетной обработки кластера соответствующее количество так называемых «заданий запуска ВМ». Каждое «задание запуска ВМ» полностью занимает один физический узел кластера и запускает несколько ВМ, по количеству физических ядер ЦПУ. ВМ загружаются, сообщают о своей готовности системе пакетной обработки эксперимента, и начинают выполняться задания пользователей. Когда заданий в очереди не остается, инициируется процедура самовыключения ВМ, и ресурсы кластера освобождаются для других пользователей.

Пара слов о KVM

Kernel-based Virtual Machine или *KVM* — это программное решение, обеспечивающее виртуализацию в среде GNU/Linux на платформе x86, которая поддерживает аппаратную виртуализацию

сделать самим

на базе Intel VT (Virtualization Technology) либо AMD SVM (Secure Virtual Machine). Очень полезной опцией *KVM* является использование дисковых образов ВМ в режиме snapshot. При этом базовый образ во время работы ВМ остается неизменным, а все изменения записываются во временный файл; причем если этот файл не указать явно, то он создастся в $TEMPDIR и автоматически исчезнет после остановки ВМ. Это дает сразу две выгоды: во-первых, разместив один образ диска на системе хранения вашего дружественного кластера, можно запускать столько ВМ, сколько вам разрешат; а во-вторых, если внутри ВМ что-то сломалось, то все это ликвидируется после простого перезапуска.

Готовим виртуальную машину

Создание рабочего дискового образа ВМ не представляет сложности и хорошо документировано. В рамках нашего эксперимента было отдано предпочтение образам формата raw, к содержимому которых довольно легко получить доступ, не запуская ВМ (обратитесь к **man lomount**). Пустой файл для такого образа можно создать командой **dd if=/dev/zero**... GNU/Linux устанавливается стандартно, путем загрузки ВМ с установочного образа. Для этой процедуры можно воспользоваться каким-либо вспомогательным ПО — например, *virt-manager* (http://virt-manager.org/).

Первое, о чем приходится позаботиться в условиях динамического кластера — настройка сети. Виртуальные машины под управлением *KVM* могут иметь сетевые устройства, обслуживающиеся различными драйверами — например, e1000 или паравиртуальный virtio. В зависимости от версии *KVM* на физическом вычислительном узле и от версии ядра GNU/Linux в виртуальной машине предпочтительнее могут быть разные варианты. Чтобы оставить возможность легко менять драйвер, поступим так:

```
device=»eth0»
mac=`/sbin/ifconfig $device | /bin/awk '/HWaddr/ { print $5 }'`
if [ -z "$mac" ]; then
  echo "no $device, try with driver e1000..."
  /sbin/modprobe e1000
  mac=`/sbin/ifconfig $device | /bin/awk '/HWaddr/ { print $5 }'`
fi
```

Поскольку дисковый образ у всех ВМ один, единственное, чем они могут отличаться друг от друга, это индивидуальный MAC-адрес, задающийся в командной строке запуска каждой ВМ. Сетевые настройки ВМ полностью выводятся из него — какой будет у ВМ IP-адрес, какой сетевой шлюз, какие маршруты.

Для каждого физического узла кластера выделяется поддиапазон IP-адресов ВМ. Физический узел работает маршрутизатором между ВМ и внешней сетью. Внешняя же сеть может быть какой угодно. Когда внутри ВМ появился доступ к сети, все необходимое стандартное ПО можно установить из репозиториев дистрибутива — ровно так же, как и в случае реальной машины.

В случае каких-то сбоев при старте может оказаться так, что *KVM*-процесс запустился и выполняется, а сама ВМ недоступна. Чтобы не занимать ресурсы зря, такие ВМ надо сразу останавливать. Для этого применяются:

- параметр ядра panic=15 (перезагрузка ВМ в случае паники ядра);
- параметр командной строки KVM **-no-reboot** (выключение ВМ при перезагрузке);
- *cron*-задача периодической проверки, работает ли на ВМ демон системы управления заданиями. Если это не так, ВМ выключается.

В остальном настройка виртуальной машины ничем не отличается от обычной. Если специальное ПО находится на NFS-серверах, надо настроить доступ к ним. Требуется внести адреса и имена ВМ в DNS, зарегистрировать их в батч-системе в качестве вычислительных узлов, убедиться, что ходит почта, и т. п.

Запуск ВМ

Когда задание запуска ВМ распределено батч-системой на конкретный физический узел кластера, начинает работу скрипт запуска ВМ. Он выполняет следующие действия:

1. Определяет количество процессоров и памяти на узле, куда он попал.
2. Решает, сколько ВМ он может запустить в этих условиях. Виртуальные машины тоже могут быть многоядерными, например, вы запускаете две 4-ядерных ВМ на 8-ядерном узле. Память делится между ВМ поровну, и примерно 20 % оставляется для работы собственной системы физического узла.
3. Запускает виртуальные машины.

Поскольку задание работает от имени обычного пользователя, а запуск ВМ требует более высоких привилегий, используется механизм **sudo**. Затем скрипт ждет завершения работы ВМ или истечения времени, отведенного на задание.

Команда запуска отдельной ВМ внутри батч-задания выглядит примерно так:

```
/usr/libexec/qemu-kvm -cpu qemu64 -name $name-$vncnbr -vnc
:$vncnbr -daemonize -drive file=$file,if=ide,index=0,boot=on,
snapshot=on -m $mem -smp $smp -no-reboot -monitor
telnet:127.0.0.1: $monitor,server,nowait -net nic,macaddr=$mac,
model=$model -net tap,script=$script,downscript=$downscript,
vlan=0,ifname=$itf drive file=$file2,if=ide,index=1,boot=off,
snapshot=on
```

Как видно, для доступа к консоли ВМ может использоваться VNC. Мониторный порт, работающий в режиме telnet, используется заданием запуска виртуальных машин для отправки им различных сигналов. В данном примере для дисковых образов используется интерфейс ide, но это может быть и virtio — в зависимости от того, что позволяют вам ваша версия KVM и ваша версия Linux в ВМ. »

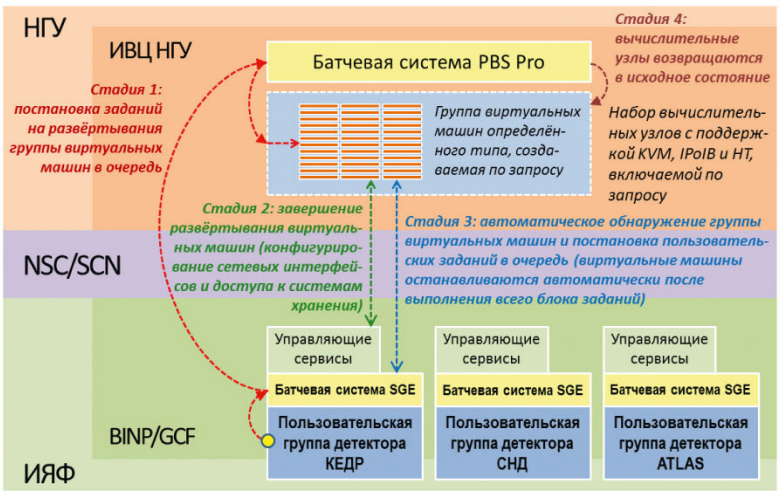

Рис. 2. Схема интеграции систем управления заданиями на группы ИЯФ и кластера ИВЦ. Показаны четыре стадии цикла развертывания ВМ на стороне ИВЦ.

Скрипт создания сетевого устройства:

```
#!/bin/sh
if [ -n “$1” ]; then
  /usr/sbin/tunctl -u root -t $1 2>&1 | grep -v ‘TUNSETIFF: Device or resource busy’
  /sbin/ifconfig $1 0.0.0.0 up
  sleep 0.5s
  /sbin/ifconfig $1
  exit 0
else
  echo “Error: no interface specified”
  exit 1
fi
```

После запуска ВМ завершается настройка сети:

```
# маршрут к ВМ
/sbin/route add -net 172.16.${block_id}.${vm_id}netmask 255.255.255.255 dev ${itf}
# другие нужные маршруты
...
# не забываем про ip-forwarding
/sbin/sysctl -w net.ipv4.conf.${external_interface}.forwarding=1
/sbin/sysctl -w net.ipv4.conf.${itf}.forwarding=1
```

Сервис-интегратор

С локальной батч-системой сервис-интегратор взаимодействует через интерфейс командной строки — запускает команды и анализирует их вывод. Он в цикле периодически проверяет наличие новых пользовательских заданий в очереди; определяет, сколько ВМ сейчас запущено и на каких физических узлах; решает, нужны ли новые ВМ. Взаимодействие с внешними батч-системами тоже осуществляется через интерфейс командной строки, пропущенный через ssh-канал. Сервис не запоминает своего состояния, начинает каждый новый цикл с чистого листа и может быть спокойно остановлен и вновь запущен в любое время.

Использование командной строки для взаимодействия с локальной батч-системой — один из недостатков данного решения. Оно приводит к повышенной нагрузке на ее сервер управления и чревато сбоями. Идеальным вариантом было бы встроить интегратор непосредственно в батч-систему, но это требует детального разбирательства того, как она устроена.

Остановка ВМ

Остановка ВМ должна происходить автоматически, если:

- отсутствуют новые пользовательские задачи, подходящие для выполнения на данной ВМ;
- истекло время, выданное батч-системой кластера;
- произошел сбой ВМ.

Одно задание запуска ВМ занимает целиком один многоядерный физический вычислительный узел кластера. Как правило, в рамках задания запускается несколько ВМ. После этого запускающий скрипт продолжает свою работу, периодически проверяя, не завершились ли ВМ и не пора ли завершать их принудительно, если время, отпущенное на задание, истекает.

Выключение ВМ при отсутствии подходящих заданий в локальной батч-системе производится через саму эту систему. Сервис-интегратор запускает задание, выключающее блок ВМ, работающих на одном физическом узле. Если окажется, что за время принятия решения об остановке этих ВМ батч-система уже успела распределить на какую-то из них настоящее задание, задание выключения будет просто отброшено.

Если скрипт запуска ВМ прождал их завершения до конца отведенного на задание времени, он посылает всем ВМ через мониторный интерфейс сигнал **system_powerdown**, а затем, если ВМ не смогли по какой-то причине выключиться нормально — сигнал **quit**, вызывающий немедленную остановку *KVM*. (Заметим, что, поскольку дисковые образы используются в режиме snapshot, это не приведет к повреждениям файловой системы.) В конце концов, когда все ВМ на узле остановлены, скрипт запуска завершает свою работу, а с ним завершается и задание. Физический узел свободен для новых заданий.

Итог

В целом система вышла довольно надежной, хотя и требующей присмотра. Она легко расширяема — ограничивающим фактором здесь будут скорее возможности хранилища данных. Усовершенствования вносятся в процессе работы по мере накопления опыта.

Кластер на коленке

Если по соседству кластеров нет, **Андрей Грозин** обеспечивает распределенные вычисления своими руками.

У вас нет под рукой готового и хорошо настроенного кластера? А считать хочется, и вы имеете представление о Python? Тогда можно собрать кластер «на коленке».

Почему именно Python? К достоинствам этого скриптового высокоуровневого активно развиваемого языка программирования можно отнести то, что первые полезные результаты появляются минут через 5–10 после начала работы. Очень многие вещи, в том числе и касающиеся высокоэффективных вычислений, не надо сочинять самому, потому что они есть в стандартных и не очень стандартных библиотеках «питона».

Есть и несколько библиотек для организации «многомашинных» вычислений. Этот текст — краткий обзор возможностей пакета RPyC (Remote Python Call, http://rpyc.sourceforge.net/). Это очень простой способ организации «кластера», а так как он может работать везде, где есть Python, то в свою вычислительную сеть можно объединить почти все, что обладает процессором, даже хоть сколько-нибудь продвинутые телефоны и домашние маршрутизаторы.

В классическом способе применения RPyC на всех облюбованных компьютерах запускаются ведомые [slave] серверы, а клиент обращается к ним и поручает выполнить некую работу. Ведомым серверам дозволено то же, что и интерпретатору Python на этой машине, без ограничений.

Важное предупреждение! Все эксперименты с «наколенным кластером» следует запускать только в изолированной сети, где нет посторонних! Ведомый сервер (по поручению клиента) может делать все, что разрешено запустившему его пользователю — скажем, удалить все файлы этого пользователя. Никакой аутентификации клиента не производится. Это оборотная сторона простоты. Для более стационарных расчетов присмотритесь к готовым системам управления распределенных вычислений.

Теперь начнем. Пусть в изолированной за межсетевым экраном домашней сети есть машины bilbo (основная) и gandalf (ведомая). Входим на gandalf через ssh и запускаем ведомый сервер:

```
gandalf> rpyc_classic
INFO:SLAVE/18812:server started on [0.0.0.0]:18812
```

Сервер ждет соединения от клиентов на порте 18812. Запускаем интерпретатор Python на bilbo и выполняем команды:

```
bilbo> python
Python 3.2.3 (default, Jan 20 2013, 20:05:19)
[GCC 4.6.3] on linux2
Type “help”, “copyright”, “credits” or “license” for more information.
>>> import rpyc
>>> gandalf=rpyc.classic.connect(‘gandalf’)
```

После этого можно выполнять различные действия с bilbo на gandalf:

```
>>> gandalf.execute(‘n=2’)
>>> gandalf.eval(‘n+1’)
3
```

Можно использовать встроенные функции Python на gandalf, хотя интерпретатор работает на bilbo:

```
>>> gandalf_file=gandalf.builtins.open(‘/home/grozin/rpyc/‘
... ‘mymodule.py’)
```

Здесь на удаленной машине (gandalf) создался файловый объект, а на локальной машине (bilbo) создалась «сетевая ссылка» (прокси-объект) gandalf_file на него. Любые исполняемые над этой ссылкой действия переадресуются объекту на удаленной машине.

```
>>> print(gandalf_file.read())
#!/usr/bin/env python
from time import sleep
class MyClass:
  def __init__(self,t):
    self.t=t
  def f(self,n):
    sleep(self.t)
    return n+1
>>> gandalf_file.close()
```

Этот «прокси-объект» можно подставить в любую программу, ожидающую иметь файловый объект. По фундаментальному для Python принципу утиной типизации «если объект ходит, как утка, плавает, как утка, и крякает, как утка, значит, он утка», этот объект — файл.

На gandalf можно использовать функции и прочие объекты из библиотечных модулей Python:

```
>>> gandalf_path=gandalf.modules.sys.path
>>> print(gandalf_path)
[‘/usr/bin’, ‘/usr/lib/portage/pym’, ‘/usr/lib64/python32.zip’,
‘/usr/lib64/python3.2’, ‘/usr/lib64/python3.2/plat-linux2’,
‘/usr/lib64/portage/pym’]
>>> gandalf_path.append(‘/home/grozin/rpyc’)
```

Теперь gandalf_path — это прокси-объект для sys.path на gandalf, и любые изменения этого объекта сразу передаются на удаленную машину. Расширив path, можно использовать файл из этой директории на gandalf:

```
>>> gandalf_object=gandalf.modules.mymodule.MyClass(0)
>>> gandalf_object.f(3)
4
```

gandalf_object — это прокси-объект (сетевая ссылка) для объекта класса MyClass на машине gandalf. Его метод f прибавляет 1 к аргументу; чтобы мы могли моделировать длительные вычисления, он это делает за t секунд, где t — атрибут данного объекта.

```
>>> gandalf_object.t=2
>>> gandalf_object.f(4)
5
```

На сей раз нам пришлось ждать 2 секунды.

Через параметры можно передавать удаленным функциям любые объекты, в частности, локальные функции. Определим такую:

```
>>> def loc(n):
...   print('loc',n)
...   return n+1
```

Тогда функция map на gandalf на каждом шаге вызывает функцию loc на bilbo (callback):

```
>>> list(gandalf.builtins.map(loc,[1,2,3]))
loc 1
loc 2
loc 3
[2, 3, 4]
```

Все, что мы до сих пор обсуждали, несомненно, красиво — разные объекты могут жить на разных машинах, и единая программа работает с ними, не замечая этого. Но все эти операции синхронные: одна машина просит другую что-то сделать и ждет, когда та вернет ей результат. Для организации распределенных вычислений нужны асинхронные операции:

```
>>> gandalf_object.t=10
>>> async_f=rpyc.async(gandalf_object.f)
>>> res=async_f(1)
>>> res.ready
False
>>> # 10 секунд спустя
>>> res.ready
True
>>> res.value
2
```

Это уже лучше. Клиент может время от времени спрашивать, готов ли результат, и когда он будет готов, забрать его. Если запросить res.value, когда результат еще не готов, то клиент блокируется до момента, когда он будет готов:

```
>>> res=async_f(2)
>>> res.value
3
```

(после res.value 10 секунд ожидания, потом появляется ответ).

Но еще лучше определить callback-функцию, которая будет вызвана на локальной машине, когда результат будет готов:

```
>>> def callback(res):
...   print(res.value)
```

Эту функцию можно вызвать только в отдельном потоке [thread]:

```
>>> bgsrv = rpyc.BgServingThread(gandalf)
>>> res=async_f(3)
>>> res.add_callback(callback)
>>> # продолжаем что-то делать
4
>>> # продолжаем что-то делать
```

Это печать из функции callback из другого потока. Теперь этот поток можно и остановить:

```
>>> bgsrv.stop()
```

Например, на клиентской машине может работать графический пользовательский интерфейс (на Python такой написать легко, причем он будет работать на любой платформе — Linux, Windows, Mac — без малейших изменений в программе). Эта клиентская программа обращается к нескольким мощным серверам для проведения длительных вычислений, и регистрирует callback-функции, которые, например, добавляют очередную точку на график.

Наконец, закроем связь с машиной gandalf:

```
>>> gandalf.close()
>>>
bilbo>
```

Собственно говоря, вот и все. LXF

Обратная связь

Приглашаем высказаться потенциальных авторов статей по параллельным вычислениям — ценные предложения, критику и советы присылайте по электронной почте: E.M.Baldin@inp.nsk.su, A.G.Grozin@inp.nsk.su, A.M.Suharev@inp.nsk.su.