Virtualization for computational scientists

G.K. Thiruvathukal, Konrad Hinsen, Konstantin Laufer, Joe Kaylor

7-1-2010

Virtualization for Computational Scientists

George K. Thiruvathukal
Loyola University Chicago, gkt@cs.luc.edu

Konrad Hinsen

Joseph P. Kaylor

Konstantin Läufer
Loyola University Chicago, klaufer@luc.edu

Editors: Konstantin Läufer, laufer@cs.luc.edu
Konrad Hinsen, hinsen@cnrs-orleans.fr

VIRTUALIZATION FOR COMPUTATIONAL SCIENTISTS

By George K. Thiruvathukal, Konrad Hinsen, Konstantin Läufer, and Joe Kaylor

Virtualization lets you carve your computer into slices, allowing for great experimentation with different operating systems, tools, and techniques.

The fun all began in May of 1999, when VMware launched VMware Workstation, a product that lets you run multiple operating systems simultaneously on your desktop computer. In truth, the story begins much earlier with the VM OS concept, which was pioneered (like many things) by IBM in 1960 but eventually perfected by others. The idea behind virtualization is simple. You can run multiple OSs simultaneously and share the CPU, memory, and peripherals among them.

In this article, we're not going to cover what virtualization is per se. This would easily require two articles, and the actual ideas behind virtualization are well explained elsewhere. And besides, we've already covered the use of virtualization in this column for use in maintaining experimental computing laboratories.[1] Instead, we'll focus here on a fairly simple use case that's likely to be of interest to our readers: setting up your own mini compute cluster to use for developing and testing high-performance computing applications.

Why Should Computer and Computational Science Researchers Virtualize?

For many years, we've been a bit frustrated with applied computer science, which overlaps with computational science. Seemingly, a new framework, tool, or solution is born every minute that was certainly predestined to be published in the literature. We're even guilty of publishing papers with purportedly useful techniques that were (at best) hard to reproduce, simply because readers couldn't download, build, install, and run the code without significant work. Even worse, sometimes even the best instructions or configuration scripts simply won't do. Systems work is complex and peppered with dependencies that trip up even the most seasoned systems hackers (as all of us can attest). Given the entropy of modern OSs (especially Linux and friends), it's often the case that something changes that causes something not to work for some subset of users. A carefully placed virtual machine (VM) image, however, makes it possible to demonstrate your work in most cases.

As an example, in a couple of recent papers, Thiruvathukal and Kaylor made their development sandboxes available as VM images. Their research looked at FUSE (file systems in user space) and at building a layer atop it to make it possible for developers to create their own FUSE filesystems without having to be complete systems geeks. While the code was actually easy to build and install, it goes without saying that giving people the ability to test drive the framework out of the box, on their own hardware, is a huge plus for prospective users and collaborators—and also spares the authors from having to make their own servers freely accessible. The idea of doing this itself was not all that new. After all, VMware has offered the Virtual Appliance Marketplace (www.vmware.com/appliances/) for years, aimed at making it easy for its customers to evaluate enterprise server software *sans* the obligatory install.

While many of our projects these days are focused on computer science (with an eye to computational science) in the systems area, systems ideas are particularly valuable to computational scientists, who often work on a variety of platforms—such as Linux, OS X, and Windows—but might not have their platforms configured for scientific application development. To this end, we decided to bite the bullet and show you how to establish a fully functioning message passing interface (MPI) cluster on your computer. We'll show you how to do it on your own, but you can also download VM images to get your own cluster going in a matter of minutes (instead of the hour or so it took us to do it from scratch).

Your Own Virtual Cluster for MPI Development

After struggling mightily to come up with a straightforward (but not too straightforward) example that would be interesting to computational scientists, we decided to start from scratch by revisiting an old friend from the

 1521-9615/10/$26.00 © 2010 IEEE

past: cluster computing. In this example, we're going to set up a simple two-node network that could be used for developing parallel software (say, using MPI). It's important to note that this isn't intended to be a complete cluster computing solution, but can be a great environment to test, develop, and learn in before working with a cluster.

There are already similar projects out there such as Bootable Cluster CD (BCCD), which we featured in a previous column,[2] and Rocks (www.rocksclusters.org/wordpress). However, we ultimately decided against both of these because they required way too much computing power or were lacking in customization potential without significant upfront investment (something that's difficult to come by when you're simultaneously writing an article). Nevertheless, this "how to" overlaps strongly with BCCD's goals to make cluster computing accessible and teachable without the hassle of setting up everything you need in a production clustering environment.

For this demonstration, we'll use the freely available VirtualBox, which runs on all platforms of current interest: Windows, Linux, OS X, and several others. If you're going to try this at home, it's important that you have a computer with sufficient computational power. A dual- or quadcore system will do, and we recommend having more than 100 Gbytes of free disk space and at least 2 Gbytes of RAM (4 Gbytes or more are preferable). Our Mac Mini test system has 2 Gbytes RAM, dual-core Intel 2.26 gigahertz CPU, and plenty of free disk space (more than 200 Gbytes). Nevertheless, in this case, we were computationally "challenged" with only two cores, so we don't plan to set up too many nodes in our virtual cluster.

Getting Started: A Process Overview

First things first: You need to visit virtualbox.org and download and install the appropriate installer for your platform. You'll also want to grab the ISO image for Ubuntu Server Edition (version 9.10 will do, 32-bit or 64-bit). Any version of Linux will do, but we assume here that you can find the equivalent packages on your favorite version of Linux (or other Unix, such as OpenSolaris or FreeBSD). We're using the 32-bit distribution, given a desire to optimize compatibility (32-bit still works best for most things) and realizing that we don't need to address more than 4 Gbytes RAM on this system anyway. Everything we're doing should work fine in 64-bit, but we haven't tested it.

Once the installer completes its work, start VirtualBox. (As we'll show shortly, much can be done at the command line as well.) We're going to set up a cluster of two nodes, which will be appropriate for our dual-core test system, but if you have more capacity feel free to repeat the instructions for additional VMs when indicated.

The process basically has four steps:

1. Set up a head node.
2. Set up one or more compute nodes. (We're actually going to use the head node as a compute node in this example with an identical software setup. In practice, however, you might have some special requirements on the head node, such as I/O.)
3. Set up the compiler toolchain and MPI runtime/scripts. This environment will be sufficient, but is by no means complete (for brevity's sake).
4. Run the world-famous "compute Pi" example.

In the remaining sections, we'll take you through the various steps, ultimately leading to a virtual MPI development cluster. The end-to-end or wall-clock time to complete all of these steps can be as short as an hour.

We're going to set up a cluster of two nodes, which will be appropriate for our dual-core test system, but if you have more capacity feel free to repeat the instructions for additional VMs when indicated.

Creating your VM

Let's start by creating the head node. We'll be cloning this node to add compute nodes, and we're likely to dive into the VirtualBox command line interface to accomplish parts of this. As indicated, you need to have the ISO image for Ubuntu Linux at this time and to know where it's located on your computer.

Creating a VM is generally simple. As you're doing this, we'll explain some of the core ideas along the way and why you should care. More important, we'll explain some alternatives that might be worth exploring. Feel free to experiment! Figure 1 shows VirtualBox's main screen, which is the first thing you will see when you launch VirtualBox to create

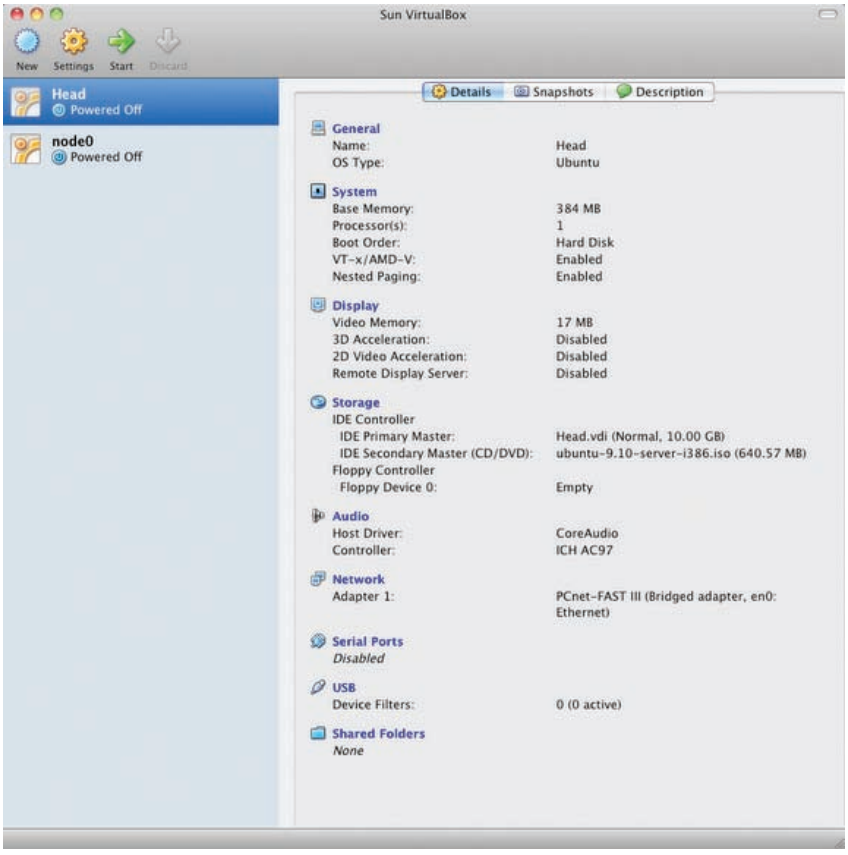

Figure 1. Main VirtualBox screen. This screen shows the authors' inventory of VMs, or cluster nodes.

or start an existing virtual machine—or virtual machines, given the topic of this article.

Initial Decisions

You can name the VM and OS type anything you wish, choosing Linux and Ubuntu as the OS and version, respectively. If you're using a current version of Linux not mentioned in the dropdown, just select "Linux 2.6." (We don't recommend using older kernels for performance reasons.) We've chosen to name this VM "head." (Later, we'll repeat this process to create "node0," "node1," and so on.)

As for base memory size, you can start lean with 384 Mbytes. Given that we're not running a desktop—such as Gnome (www.gnome.org) or KDE (www.kde.org)—in this VM, it can probably be a bit lower if you're working with limited memory (under 2 Gbytes total).

Next, you create the virtual hard disk by selecting "Create new hard disk." You'll want at least 5 Gbytes. We've found that even when we do a desktop install of Ubuntu with almost all packages installed, it takes 3.3 Gbytes. That said, you want to leave room for growth, so 5 to 6 Gbytes should be more than sufficient.

Disk Creation Decisions

At this point, we could either use the command line utility `VBoxManage createhd` to create all of our hard disks for us, or use `VBoxManage clonehd` to clone an existing hard disk. We'll likely take advantage of these shortly when adding compute nodes. For now, it's time to enter a disk creation wizard.

Hard disk storage type. Go ahead and choose "dynamically expanding storage." In general, you'll want this option because hard disks are stored as files on the host, and you want to allocate only space that the installed OS will actually use. That is, as a virtual hard disk fills up, a disk file will begin to "grow" in size.

If you're a hardcore systems geek, you're probably thinking that this can't be good for performance—and you'd be right to a point, because this strategy is replete with opportunities for fragmentation. That said, most OSs are getting better at handling large files and virtualization in general (which is truly here to stay), so if you're seriously worried about performance, select "Fixed-size storage." Be warned, however, that you'll need to wait while VirtualBox allocates the space for the entire hard disk!

Virtual disk location and size. Accept the default location name, which should match the VM name you selected earlier. We recommend setting the size to at least 20 Gbytes, given that you'll lose only what you use, so to speak. You're not going to need more than that for these initial experiments.

Speaking of location, keep in mind that VirtualBox lets you put your VMs and disks anywhere you like. However, by default, a directory/folder (`.VirtualBox`) is kept in your home directory. This is true even on Windows and OS X (`Library/VirtualBox` instead of `.VirtualBox`) and can be a bit confusing if you're not familiar with hidden files and the like.

Customizing Your Virtual Machine

After finishing both the disk and VM wizards, you'll see a "Details" tab that shows you the details of your VM configuration. However, we're not quite ready to boot the VM yet. Although you've completed the wizard, it's still possible to make some

last minute changes. And there are several things you're likely to want to change. We'll focus here on some of the highlights.

Storage. To actually install Ubuntu Linux, we must ensure that the CD device is mapped to the actual ISO image. After clicking on "Storage," you'll notice that there should be two devices there: a virtual hard disk (the file you created for the hard disk, likely named `head.vdi`) and an empty CD. On the right side of this panel is a folder icon, which you can use to select the ISO image and add it to the inventory. I'm going to assume that most people can select the file (`ubuntu-9.10-server-i386.iso`). If all goes well, what you see will resemble Figure 2.

System. If you've got major power at your fingertips, you might want to adjust the number of processors and base memory. We're assuming most readers don't have these kinds of resources on a desktop computer, but we do want to call your attention to the boot order: make sure that CD/DVD-ROM precedes the hard disk. Some virtualization solutions (such as VMware) don't make it easy to change these, leaving the task to the emulated BIOS setup. In such cases, you need very fast fingers to select ESC, F2, or DEL (which is how you usually get into the BIOS setup on your own computer). Luckily, in VirtualBox it's a piece of cake.

Network. We're also going to need to tweak this later. To set up our virtual cluster's head node, we can go with the single adapter (Bridged). In some situations, you might want to pursue other options. Bridged is great when you're already set up with a private, secure home network, and it's appropriate for our experiments. However, if you're behind a proxy or find that your VM cannot get Dynamic Host Configuration Protocol (DHCP) addresses from your upstream router, you'll want to consider using Network Address Translation (NAT) or setting up a host-only network, which will dynamically assign IPs to your configured VMs. For brevity's sake, I'm going to forego some of the details of networking here so we can focus on the good stuff!

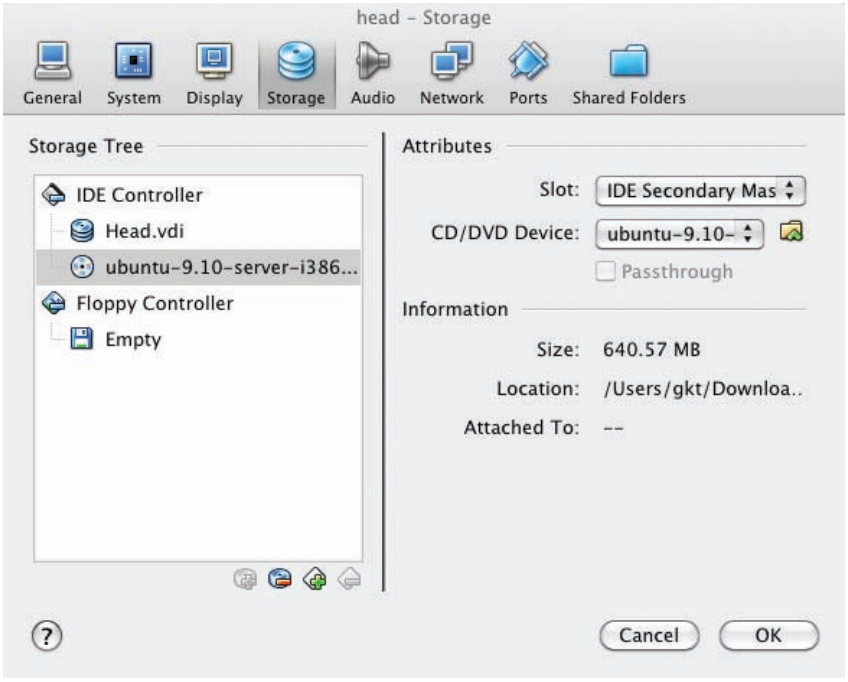

Figure 2. The storage screen showing the CD mapped to the Ubuntu 9.10 server ISO image. You must see what is shown here, especially if you want to install the actual OS in the VM.

Universal Serial Bus. We actually recommend disabling the USB for now. We're setting up a cluster, so we don't need access to peripheral devices that are available for USB. Deselecting this option doesn't affect your keyboard and mouse, which are handled using legacy serial logic. So these devices are always seen in the VM.

Installing the OS

Now that we have a configured VM for the head node that's mostly (if not entirely) to our liking, it's time to do the actual OS install. Go ahead and boot your VM machine by clicking on the "Start" arrow.

At this point, you'll see the Ubuntu server boot menu (see Figure 3). If you don't, then it's likely that you either haven't set the boot order to have the CD-ROM first or didn't map the ISO image to the CD-ROM device correctly.

First, select "Install Ubuntu Server." The entire setup process is straightforward; for those new to this type of thing, here's a quick cheat sheet:

1. Select your options, including keyboard layout, time zone, and so on. In most cases, choosing these options is trivial.
2. Partition your virtual disk. To make life easy, simply select the auto-partitioning option.
3. Set up a user/password for the administrator account. For new users, we recommend something like "manager." For this demo, make sure that every node is configured with the same users, because we're not going to be configuring an advanced naming

Figure 3. Ubuntu server installation menu. The setup process is straightforward; once installation is complete you can boot the server to see if everything worked as planned.

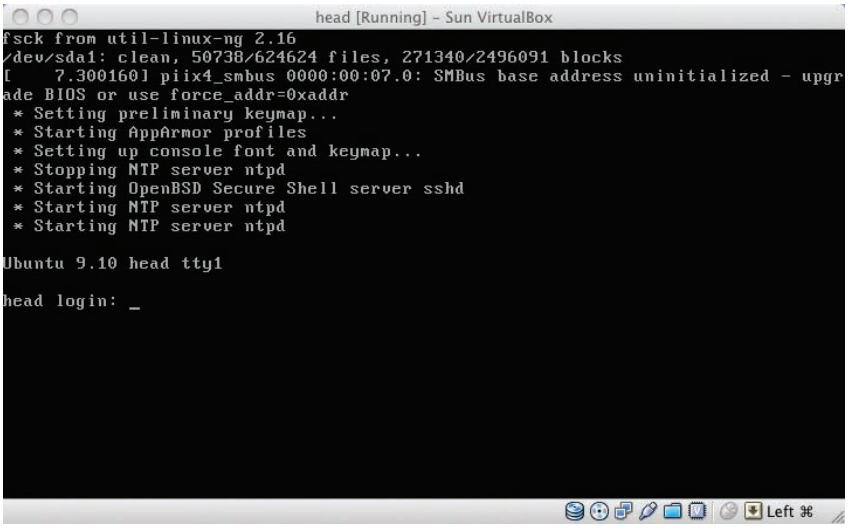

Figure 4. The head node console after boot. The output in this figure shows several boot diagnostics and services being started. If all has gone well, a `login:` prompt will also appear.

service. There will be a "manager" user on every node, and we'll show you how to set up "ssh" so you can log into all of the nodes, password-free.

4. The installation will complete after awhile. We need to set up the OS only once. We'll show you how to clone nodes, although some manual steps will be required as we add nodes to the cluster.

Once the installation has completed, you can boot the server to ensure everything worked out (see Figure 4). You might want to first perform a proper shutdown so you can deselect the CD-ROM from the system "Boot Order" section. However, if you choose not to, you'll just see the Ubuntu installation menu and can boot from the first hard disk to enter your newly installed node.

If you're using VMware instead of VirtualBox, it will automatically do the base OS installation, doing steps 1 through 4 completely unattended. As you'll see in the rest of the article, you'll still have your work cut out for you (to install networking services and MPI) but it's slightly less work.

A Look at the VirtualBox Command Line Interface

Once you've booted and logged in as manager, you should see the following:

```
$ VBoxManage list vms
Sun VirtualBox Command
  Line Management Interface
  Version 3.1.6
(C) 2005-2010 Sun
  Microsystems, Inc.
All rights reserved.

"Head" {90bbcce2-6102-4057-
  b701-0e781e77131a}
"node0" {98552b67-8ea8-4533-
  a2ba-a8926ddcce87}
```

This shows you all of the VMs that VirtualBox is presently managing. On our system, we have a couple of VMs. To see the VM information for the head node, you type

```
VBoxManage showvminfo
  90bbcce2-6102-4057-b701-
  0e781e77131a
```

This basically gives you a textual version of the information that you

see in the VirtualBox main screen's "Details" tab:

```
$ VBoxManage showvminfo
  90bbcce2-6102-4057-b701-
  0e781e77131a
Sun VirtualBox Command
  Line Management Interface
  Version 3.1.6
(C) 2005-2010 Sun
  Microsystems, Inc.
All rights reserved.

Name:            Head
Guest OS:        Ubuntu
....
```

As you can see, it's fairly lengthy, but it gives you an idea of how you can do just about everything outside of the GUI framework—if you wish. Because the long-term interest of this exercise is to set up a complete cluster on-demand and automatically, it's important to be able to do it all from the command line. (Imagine, for example, that you wanted to set up a 16-node cluster—of course, most people don't have enough memory to try that at home.)

Suppose, for example, that we want to create a new compute node, say, `node1`. The steps we'd take are

- clone an existing hard drive (likely from `node0`);
- create a new VM that's identical to `node0`'s configuration, except for its hard disk;
- attach the cloned hard drive to the VM; and
- boot the VM and change its hostname from `node0` to `node1`.

The final step isn't strictly automated at this point (and is a bit beyond what we'd planned for this article). In general, because we can dynamically assign hostnames with DHCP, this step doesn't (strictly speaking) violate our design ideal of complete automation.

So let's take a look at the hard disks in our inventory (belonging to existing VMs):

```
$ VBoxManage list hdds
Sun VirtualBox Command
  Line Management Interface
  Version 3.1.6
(C) 2005-2010 Sun
  Microsystems, Inc.
All rights reserved.

UUID:        27048708-1f52-
             4c84-ac9e-
             641f3e706086
Format:      VDI
Location:    /Users/gkt/Library/
              VirtualBox/
             HardDisks/Head.vdi
Accessible:  yes
Type:        normal
Usage:       Head (UUID:
             90bbcce2-6102-
             4057-b701-
             0e781e77131a)

UUID:        f963ccf6-0263-
             4037-8c6f-
             f525bfde4394
Format:      VDI
Location:    /Users/gkt/Library/
             VirtualBox/
             HardDisks/node0.
             vdi
Accessible:  yes
Type:        normal
Usage:       node0 (UUID:
             98552b67-8ea8-
             4533-a2ba-
             a8926ddcce87)
```

As you can see, there are two hard drives, each of which belongs to one of our VMs. These disks are kept in the VirtualBox directory (usually `~/.VirtualBox` on Linux or Windows and `~/Library/VirtualBox` on Mac). We recommend keeping your disks in the default locations so you don't have to think about where to put them (at least while learning how stuff works!)

We recommend keeping your disks in the default locations so you don't have to think about where to put them (at least while learning how stuff works!)

Here's the `VBoxManage` command to clone a hard drive:

```
VBoxManage clonehd <uuid>|
  <filename> <outputfile>
    [--format VDI|VMDK|
      VHD|RAW|<other>]
    [--variant Standard,Fixed,
      Split2G,Stream,ESX]
    [--type normal|writethrough|
      immutable]
    [--remember] [--existing]
```

The most important options here is `--remember`. If you don't use it, then you won't be able to select this hard disk for inclusion in a new VM. That is, it won't show up on the `VBoxManage` list `hdds` output above!

You'll obviously need to look up the Universally Unique Identifier (UUID). For `node0`, we want

Variations on a Virtual Theme

The main article's topic, desktop-based virtualization, is in many ways a limited subset of the larger virtualization space. Virtualization is a topic of growing interest and one we're likely to explore again in this department. Following is a sampling of other things you can do with virtualization.

Virtual Appliances

The idea of using a VM to package and distribute ready-to-run software is still rather new, but established enough already that the term "virtual appliance" has been coined for such a software distribution.

To get an idea of what a virtual appliance can do, look at TurnKey Linux (www.turnkeylinux.org) or Bitnami (http://bitnami.org/). These sites provide a collection of ready-to-run VMs for various server tasks, including web servers with content management systems, issue-tracking systems for software development, and wikis. Just download the virtual appliance you want and import it into your VM manager with a few mouseclicks, then boot the VM and follow the instructions. You get an administration interface, command-line access via ssh, and of course the web applications you wanted in the first place. You can also use most of the VMs immediately on Amazon's Elastic Compute Cloud (EC2) cloud service. Setting up your own Web server has never been easier.

An example of a scientific computing virtual appliance is HUBzero (http://hubzero.org), a collaborative platform[1] that offers users a collection of virtual appliances based on OpenVZ (http://openvz.org), a lightweight form of VMs specifically designed for virtual appliances. OpenVZ supports only one OS (Linux) but has less overhead than complete virtualization solutions. Using OpenVZ or similar technologies, you can completely isolate software installations running on the same hardware and even have the installations managed by different people.

Virtual Machines in the Cloud

Among the reasons for virtualization's growing popularity is the rise of grid and cloud computing. Both techniques aim to make computing resources commodities just like water and electricity. You can prepare and submit jobs and have them executed by some available CPU somewhere—in fact, you don't care where. But running your computational job requires that the CPU have all of your software, in exactly the versions you need. How can you prepare such an installation for an unknown computer? Build a VM!

The oldest and best-known cloud computing service, EC2, is based on virtualization technology. To run a job in EC2, you must first prepare an Amazon machine image or choose from a range of existing ones. You prepare and test the machine on your own computer, and then upload it to Amazon's computing centers. Other cloud services work in a similar way.

Rent Your Own Virtual Machine

There are times when something is so useful that it deserves a mention. We've already done that in our article by mentioning VMware, which remains one of our favorite virtualization solutions, especially on the desktop. So there's already at least a precedent of sorts.

So, among the growing number of solutions for renting your own virtual server is Linode.com, which lets you pay as you go for a hosted server. For as little as US$20 a month, you can rent a server that lets you set up any version of Linux you like. Linode.com uses Xen and has its own control panel to set up the OS from the Web. With Xen, you can attach to the console, which is particularly

UUID `f963ccf6-0263-4037-8c6f-f525bfde4394`. We recommend that you carefully name your disks after the VMs they'll ultimately live in.

```
VBoxManage clonehd f963ccf6-
  0263-4037-8c6f-f525bfde4394
  node1.vdi -remember
```

And voila, you now have a new hard disk that contains a complete clone of `node0`, which we can use in a new VM.

At this stage, we could use `VBoxManage createvm` to create a new VM with this hard disk from the command line. However, there's some customization required to get a new node up and running, even though every node's configuration is largely the same. In particular, the new node must have its own hostname (`node1` instead of `node0`, and so on)

So, the easiest thing to do at this point is to fire up VirtualBox and create a new VM (just like we did for the head and `node0`). When you come to the point where you need to create the disk, you'll select the option to use an existing disk, which you can then attach to the newly created VM. Simply boot the VM, login as manager, and make the following changes:

- edit `/etc/hosts` and replace the entry for `127.0.0.1` with the proper hostname (`node0`),
- edit `/etc/hostname` with the proper hostname, and
- reboot.

After rebooting, it's official: You have a two-node cluster!

Setting up Network Services on the Cluster

Now that we have our nodes up and running, it's important to set up some additional packages. We'll begin by setting up some commonly needed network services. Make sure you're logged in as manager on each of the nodes. We use OpenSSH to provide remote login service:

```
$ apt-get install openssh-
  server avahi-utils ntp
```

Avahi provides zero configuration (zeroconf) networking, letting us look up hosts by name without running a standalone network directory

useful if anything goes wrong with your virtual server and it needs a reboot or other maintenance.

Virtual Machines on Virtual Machines

Long time readers of this column know that we've covered Java and other languages that run on virtual machines. These VMs differ from the notion of virtualization. Language VMs (a more precise term) are aimed at providing a layer of abstraction above the hardware (often at a significant penalty) by using a fictitious machine design with its own instruction set.

The Java VM (JVM), for example, is significantly different than the processor it typically runs on. For example, it uses a stack-based design (as opposed to registers). It also has a relatively simple, abbreviated instruction set. It even imposes some restrictions that many computational scientists don't like, such as mandatory bounds checking.

Virtualization differs from the language VM notion because it actually uses the host hardware to run the OSs and applications as is. And, with few exceptions, you need to run an OS that has the *right* architecture (or a compatible one) to virtualize it. For example, on his 64-bit dual core laptop, Thiruvathukal runs Linux in 32-bit mode (primarily because he finds that 32-bit Linux has fewer headaches than 64-bit, but the situation is improving).

From VMs to virtual VMs

For a few years, confusion caused by the different meanings of "virtual machine" was limited: there were basically virtualized computers (the topic of our main article) and language VMs (as we just described). Recently, another "virtual" technology has appeared and rapidly gained interest: the low-level virtual machine (LLVM, http://llvm.org/).

LLVM started as a research project in 2000, and got considerable press exposure last year when Apple decided to use it as the basis for its next-generation MacOS compilers and for its OpenCL implementation in MacOS 10.6. LLVM bears some resemblance to language VMs, such as the JVM, in that it defines a fictitious machine design to be targeted by compilers. However, LLVM lacks other parts of a typical VM, such as memory management and OS interfaces, and has a different role: it's a building block for compiler writers that provides state-of-the-art optimization techniques.

If you've resisted confusion until now, let's move on to the next level. Suppose you want to write your own (language) VM, such as a JVM optimized for a specific application domain or target platform. Wouldn't it be nice to have a toolkit with most of the required ingredients, such as a just-in-time (JIT) compiler and a memory management library with garbage collection? If so, perhaps VMKit (http://vmkit.llvm.org/) is just what you need. It uses LLVM as its JIT compiler, so your newly written VM will actually be based on another VM (of a quite different kind). And you can run all that on a virtualized PC if you want. That's three levels of virtualization!

But there's no need to stop there. How about a virtual VM? That's a VM that takes as its input the specification of another VM, which then runs Java bytecodes or something similar. It also lets you change the VM while it's running by loading "VMlets." This is different from VMKit's approach, where the VM is defined by a C++ program, although of course nothing stops you from using VMKit to implement a virtual VM that loads extensions dynamically. Virtual VMs are still a research project for now (http://vvm.lip6.fr/), but they illustrate what can be done using various virtualization approaches.

Reference

1. M. McLennan and R. Kennell, "HUBzero: A Platform for Dissemination and Collaboration in Computational Science and Engineering," *Computing in Science & Eng.*, vol. 12, no. 2, 2010, pp. 48–52.

service such as network information service (NIS) or Lightweight Directory Access Protocol (LDAP) (or having to manually edit the `/etc/hosts` file). When a node runs Avahi, it "announces" itself using multicast so other peers can find it. The `ntp` is the Network Time Protocol. It's generally good practice in networked systems to have the correct time on all connected stations/devices.

Again, you'll run the `apt-get` command on each node in the cluster. Once we have OpenSSH on all nodes, we can actually do most system administration in our cluster remotely.

```
$ ssh-keygen -t dsa -b 2048
```

Do this just on the head node, logged in as the manager user (that you created earlier), then copy the key to all of the other $N-1$ nodes to create a Digital Signature Algorithm (DSA) key. You'll then be asked to provide a pass phrase. Leave it blank. Otherwise, you'll need to enter your password when you use the key to access this or another node in the cluster (which you don't want).

Copy the key to the compute node(s):

```
$ scp -r ~/.ssh manager@
  node0.local:.
$ scp -r ~/.ssh manager@
  node1.local:.
```

If all has gone well, you should now be able to log into any node (from any node) as follows and execute a command without having to enter your password:

```
$ ssh node0.local hostname
$ ssh node1.local hostname
```

Getting MPI Running on Our Cluster

So now that we have our nodes up and running and can log in with SSH, we're just about ready to run our first MPI program, compute Pi (see Figure 5). We just need to make sure that an essential build environment and MPI are installed on all of the nodes. This is fairly easy on Ubuntu!

```
$ sudo apt-get install build-
  essential gcc g77 g++
  openmpi-bin openmpi-common
  libopenmpi-dev
```

Figure 5. The compute Pi (`cpi`) example running on the two-node virtual cluster. This example is often used as a basic test of the "sanity" of one's MPI environment. If all goes well, you'll see that `cpi` has run on the various compute nodes selected from the machinefile.

This command installs MPI applications using C, C++, and Fortran 77 (you'll need to dig deeper if you aren't using one of those oldies but goodies). This environment should be installed on every node. So, now that you have `ssh` installed on every node, you can actually do it this way:

```
$ ssh node0.local sudo
  apt-get install build-
  essential gcc g77 g++
  openmpi-bin openmpi-common
  libopenmpi-dev
```

Note: You'll be asked to enter your manager password to complete the `sudo` operation. You can configure this to work password-free by changing the `%admin` line in `/etc/sudoers` to

```
%admin ALL=(ALL) NOPASSWD:
  ALL
```

Please note: letting administrators become root without a password is a potential security risk in production environments. However, because this server is running in a more confined setting, the convenience of being able to run a command on multiple cluster nodes outweighs the security risk. Nevertheless, if you're using a machine connected to the public Internet (say, your office computer), please don't add the `NOPASSWD` option above.

To make it easy for you to get the examples going without having to download the entire MPI Chameleon (MPICH), we've set up a repository at Google Code. This repository contains a clone of the `mpich/examples` directory, where useful MPI demonstration programs live (the ones we typically use to test our MPI installation's sanity). You'll need to have Mercurial installed on at least your head node. You can do this as follows:

```
$ sudo apt-get nstall
  python-dev python-setuptools
$ easy_install -U mercurial
```

Then you can use Mercurial to get a hold of the examples.

```
$ hg clone https://
  virtualization.cise.
  googlecode.com/hg/
  cise-virtualization
```

Build the Compute PI (cpi) example:

```
$ cd cise-virtualization/
  mpich-examples
$ mpicc -o cpi cpi
```

Next, you can create a machines file that tells MPI the nodes you want to use to run programs. (These are selected round-robin based on how many MPI processes you intend to create.)

```
$ cat > machines
head.local
node0.local (and any other
  nodes you created)
^D
```

Use `rsync` to "backup" your MPI code (most important, the executable code) to the other nodes. Here is how you get the code over to `node0.local`:

```
$ rsync -avz $HOME/cise-
  virtualization/ node0.local:
  $HOME/cise-virtualization/
$ mpirun -machinefile
  machines -n 2 cpi
```

When you invoke `mpirun` as above, you'll see the nodes where processes were started.

We intended this as a *development environment*—not a high-performance cluster. Nevertheless, it's awfully close to what you find in a real cluster. With a bit more work (beyond the scope of this article), we could have set up Network File System (NFS) to include shared directories (so that we don't need to `rsync` the code).

Great, you might say, *but do I really need to know how to do this from scratch?* Of course not, but then again, what fun would there be in that?

It's likely that many readers will want to simply grab the images and get started right away. We can understand this, given how most of us are seemingly busier than ever. However, because we think there's a growing interest in topics like this—and because at least one of us has students who surely could have used this in an HPC course last year—we plan to continue working on evolving this and would certainly be interested to hear what you think.

For updated instructions and links to the latest images, see http://gkt.etl.luc.edu/writings/cise-virtualization. Our hope is to one day have a fully automated process that will let you build your own *N*-node cluster (and perhaps we'll have even worked out most of the details by the time you see this link …).

References

1. J. Kaylor and G.K. Thiruvathukal, "A Virtual Computing Laboratory," *Computing in Science & Eng.*, vol. 10, no. 2, 2008, pp. 65–69.
2. P. Gray and T. Murphy, "Something Wonderful this Way Comes," *Computing in Science & Eng.*, vol. 8, no. 3, 2006, pp. 82–87.

George K. Thiruvathukal is an associate professor of computer science at Loyola University Chicago and associate editor in chief of CiSE. His technical interests include parallel/distributed systems, programming language design/implementation, and computer science across the disciplines. Thiruvathukal has a PhD in computer science from the Illinois Institute of Technology. Contact him via http://gkt.etl.luc.edu.

Konrad Hinsen is a researcher at the Centre de Biophysique Moléculaire in Orléans (France) and at the Synchrotron Soleil in Saint Aubin (France). His research interests include protein structure and dynamics and scientific computing. Hinsen has a PhD in theoretical physics from RWTH Aachen University (Germany). Contact him at konrad.hinsen@cnrs-orleans.fr.

Konstantin Läufer is a professor of computer science at Loyola University Chicago. His research interests include programming languages, software architecture and frameworks, distributed systems, mobile and embedded computing, human-computer interaction, and educational technology. Läufer has a PhD in computer science from the Courant Institute at New York University. Contact him via www.cs.luc.edu/laufer.

Joe Kaylor is a software engineer at a Chicago-area software firm and a member of the Emerging Technologies Laboratory. His research interests include storage, virtual memory, and compilers. Kaylor has an MS in computer science at Loyola University Chicago in 2010.