;login:
THE MAGAZINE OF USENIX & SAGE
August 2001 • Volume 26 • Number 5

Special Focus Issue:Clustering

Guest Editor: Joseph L. Kaiser

inside:

CLUSTERS

LARGE CLUSTERS FOR THEORETICAL PHYSICS AT FERMI LAB
by Don Holmgren and Ron Rechenmacher

USENIX & SAGE
The Advanced Computing Systems Association &
The System Administrators Guild

large clusters for theoretical physics at Fermi Lab

by Don Holmgren,

Don Holmgren is the leader of the Distributed Systems Projects Group (DSP) of the Integrated Systems Development Department of the Computing Division of Fermilab.

djholm@fnal.gov

and Ron Rechenmacher

Rechenmacher is an engineer in the Distributed Systems ProjectsGroup of the Integrated Systems Development Department of the Computing Division at Fermilab in Batavia, IL.

ron@fnal.gov

Introduction

The Fermi National Accelerator Laboratory (*http://www.fnal.gov*) is a US Department of Energy facility located in Batavia, Illinois, about 35 miles west of Chicago. Scientists at Fermilab work on understanding elementary particles, the area of science known as high energy physics. One area of study, performed by theoretical physicists, concerns quarks and gluons. Sets of quarks, bound together by gluons, compose particles such as neutrons and protons, which in turn make up most of the known mass of the universe.

Lattice gauge quantum chromodynamics, or lattice QCD for short, is a numerical technique for modeling the interactions, or QCD, between quarks and gluons. Physicists employ lattice QCD to help interpret the results of experiments, with the ultimate goal of determining the validity of the "Standard Model" of physics. The calculations of lattice QCD are very floating-point intensive, requiring anywhere from a few days on a machine capable of gigaflops for simple calculations, to several years on machines capable of teraflops for the most complicated problems. Once the domain of commercial and purpose-built supercomputers, lattice QCD now often finds a home on clusters of commodity computers.

At Fermilab, we currently operate a lattice QCD cluster consisting of 80 systems, each a dual 700MHz Pentium III computer connected to a high-bandwidth, low-latency Myrinet network. We regard this as a prototype production cluster, and on it we seek to develop the techniques and codes which we will use in the next few years on clusters of 1,000 or more computers.

Lattice QCD codes attempt to solve the Dirac equation using non-perturbative methods. The continuous fields and wave functions of this four-dimensional partial differential equation are discretized onto lattices represented by multidimensional arrays, essentially the same finite-difference technique employed in electromagnetic and structural engineering analyses. As in finite-difference methods, differences in the values of the modeled functions between a given lattice site and its neighbors approximate various derivatives.

Unlike conventional finite-difference techniques, the quantum mechanical properties of the modeled particles require additional corrective terms. Such properties manifest themselves on the distance scales characteristic of the particles, such as neutrons and protons, which are composed of quarks. Lattice spacings on the order of 10^{-16} meters must be used, and lattice sizes of L^4, where L ranges from as low as 6 to as high as 64 or even 128, are employed. Many quantities, in the form of complex 3x3 matrices and complex vectors, must be stored to represent the model at each lattice site.

A large lattice QCD problem may thus require tens or hundreds of gigabytes of memory. Starting from a random configuration, the lattice QCD code will iterate a number of times over the lattice, performing at each site a set of floating-point operations on the values stored at neighboring sites. Such an algorithm may be parallelized and the work spread across a number of processors in order to reduce the time required to obtain a solution. When running on a cluster of computers, the code distributes the lattice over the memory of the machines of the cluster and uses a message-passing API, such as MPI

or PVM, to communicate the values stored at lattice sites at the boundaries between computers. The most common lattice QCD code used on the Fermilab lattice QCD cluster is known as MILC and was developed by a consortium of physicists at a number of universities (see *http://physics.indiana.edu/~sg/milc.html*).

In order to solve the problems of interest over the next decade, teraflops-scale facilities are required. The 80-node production prototype at Fermilab sustains approximately 10Gflops on MILC codes. Therefore, we envision building clusters on the order of 1,000 computers ("kiloclusters") over the next years. To reach the teraflops scale, we will depend both on large numbers of processors and on the heretofore steady increase in processor performance described by Moore's law.

In order to build and operate kiloclusters, we need to develop tools and techniques which scale. In the rest of this article, we discuss a pair of examples. First, we describe the network-based tools we use to install the Linux operating system on our nodes and to upgrade or modify the BIOS and other firmware of our computers. Second, we describe command-invocation tools which we use to perform common administration tasks simultaneously on all nodes of our clusters.

We envision building clusters on the order of 1,000 computers ("kiloclusters") over the next years

Network Installs and Firmware Upgrades

On large clusters which execute programs similar to our lattice QCD codes, it is essential that all of the nodes be as nearly identical as possible. The codes implicitly assume this homogeneity, spreading the lattice evenly across the systems. Each iteration of the lattice requires the same calculations be performed at each site. The slowest machine in the cluster dictates the rate at which iterations occur. Thus, no difference in hardware or software configuration which might affect performance can be tolerated on any of the computers. Further, minimizing the amount of state held by any system in the cluster simplifies the replacement of a failed unit.

We achieve homogeneous software configuration by installing identical system images on the machines in the cluster, modifying only those files related to identity (hostname, network address, etc.). We achieve homogeneous hardware configuration by installing the same versions of the BIOS and other firmware on each of the nodes of the cluster, and by setting identical BIOS configurations on each system. We minimize the man-power required for these operations by employing automated, network-based installation and firmware upgrade tools.

We have grown to fear what we call "the 15-minute catastrophe." Commodity hardware occasionally requires "out of band" maintenance – the sort of operation in which the system administrator hooks up a keyboard and a video monitor, or reboots the system from a special floppy disk. For a cluster of 10 systems, an operation which requires 15 minutes of attention per computer is inconvenient. On a cluster of 1000 systems, this same operation can be a disaster – 1,000 systems times 15 minutes per system equals six person weeks.

For some operations, we cannot avoid directly interacting with a node. For our 80-node cluster, we selected a motherboard, the L440GX+ from Intel, which allows redirection of BIOS input and output to a serial port. We have connected each of the nodes to a serial multiplexer, and from any X Window session we can open a window and interact with the BIOS in a manner identical to using a directly attached keyboard and video monitor. This motherboard also has an "emergency management port" (EMP). Via a second serial line connected to the EMP, we can reset or power cycle the computer.

Operating system installation and BIOS upgrades are two examples of "out of band" maintenance, which usually requires booting from a special floppy disk or CD-ROM. Instead, we rely on the network booting capability available on certain brands of Ethernet interfaces. Installation of Linux on a node in our cluster proceeds as follows. First, on our boot server we modify the /etc/bootptab entry corresponding to the system requiring installation. Our Ethernet interfaces (Intel Pro/100) have PXE (preboot execution environment) support. When reset, the systems execute the PXE code, which checks for a DHCP or BOOTP server willing to provide a system image. Peter Anvin's "pxelinux," a variant of the "syslinux" used on many Linux distributions' installation boot floppies, is loaded via TFTP, as specified in the server's bootptab file. "pxelinux" in turn uses TFTP to load and start a minimal memory-based Linux image. Via the standard init.d mechanisms, this image fetches and starts an installation script. The script partitions the computer's disk drive and makes file systems, fetches master tar files, explodes these tar files, modifies network configuration scripts, and runs LILO to set up future booting from the hard drive. Via rsh the script then unmarks the /etc/bootptab entry on the file server. Finally, the script resets the node. The node then boots from the newly installed image on its system disk.

The installation of Linux over the network using this technique completes in less than five minutes. Other than the initial modification of the server's bootptab file, the operation is unattended. We can reinstall Linux onto all 80 nodes of our cluster in about 40 minutes. The limiting factor in such mass installs is the network bandwidth available from our server. Scaling to a kilocluster will require either multiple servers or multicast techniques. The PXE code from Intel already includes a multicast TFTPD and a multicast TFTP client in the BIOS. However, multicast TFTP clients for Linux are not currently available.

Network-based BIOS installation proceeds along similar lines. Vendor-supplied BIOS installation programs all require booting the system into an MS-DOS or similar environment. Normally this requires running the operating system from either a floppy disk or a system disk. Fortunately, the Netboot and Etherboot projects (see *http://etherboot.sourceforge.net/*) have contributed codes which patch standard DOS to run out of a RAM disk. A BIOS upgrade using these tools proceeds as follows. As before, the server's bootptab file is modified and the system to be upgraded is reset. The PXE code on the system's NIC solicits a DHCP or BOOTP server. "pxegrub," from the GNU Grub project, loads via TFTP and starts. "pxegrub" in turn fetches a DOS image tagged by the Etherboot tools. DOS starts from this image, which includes a large RAM disk containing the various binaries and datafiles required for installation of the BIOS. The standard autoexec.bat mechanism is used to launch the firmware upgrade binaries. Unfortunately, we have not been able to use the same rsh technique to undo the modification to the server's bootptab file. So, prior to rebooting the system, we must manually reconfigure this file, then reboot the node. BIOS installation via this technique completes in less than five minutes.

The installation of the BIOS also results in modification of various BIOS parameters to their default values. In some cases, these values are not appropriate for our cluster. We have had some limited success under Linux with restoring parameters via the special /dev/nvram device. However, we often must manually alter parameters via direct interaction with the various BIOS menus.

Parallel Command Tools

Nearly every system administrator tasked with operating a cluster of UNIX machines will eventually find or write a tool which will execute the same command on all of the nodes. At Fermilab we call this tool rgang. The history of rgang begins in the distant mists of farm computing at Fermilab. It was originally scripted in an afternoon by Marc Mengel to deal with the common administration and installation problems associated with the IBM farms that Fermilab was using. While its code has changed, the basic use of the tool remains the same. rgang relies on files in /usr/local/etc/farmlets/ which define sets of nodes in the cluster. For example, all lists all of the nodes, row1 lists all of the nodes in row 1, and so forth. The administrator issues a command to a group of nodes using this syntax:

```
rgang farmlet_name 'command arg1 arg2 ... argn'
```

On each node in the file farmlet_name, rgang executes the given command via rsh or ssh, displaying the result delimited by a node-specific header. rgang is implemented in Bourne shell.

Because rgang executes the commands on the specified nodes serially, execution time is proportional to the number of nodes. In Python, we have implemented a parallel version of rgang. This version forks separate rsh/SSH children, which execute in parallel. After successfully waiting on returns from each child or after timing out, parallel rgang displays the node responses in identical fashion to rgang. In addition, parallel rgang stores the OR of all of the exit status values in a shell variable. Simple commands execute via parallel rgang on all 80 nodes of our cluster in about three seconds.

To enable scaling to kiloclusters, parallel rgang can utilize a tree, via an nway switch. When so invoked, parallel rgang uses rsh/SSH to spawn copies of itself on multiple nodes. These copies in turn spawn additional copies.

There are currently two major modes for rgang.py: command mode and copy mode.

The command syntax for the modes are 1):

```
rgang.py  [options] <nodes-spec> <command>
```

and 2a):

```
rgang.py -c [options] <nodes-spec> <file> <dir|file>
```

or 2b):

```
rgang.py -c [options] <nodes-spec> <file> <file>... <dir>
```

Of all the various options, the most significant is the –nway=number option. This option determines the "fan-out" behavior of rgang. The default, when the option is not specified, is 0. This makes the fan-out equal to the number of nodes specified, which causes the rgang action (command or copy) to occur in parallel. Other values for nway, cause rgang.py to attempt to build a tree-structure in a "worm-like" fashion to accomplish its task.

For instance, a value of 3 for nway with a list of 10 nodes will attempt to fan out the process in a manner illustrated by diagram 1.

Contrast the above structure with diagram 2, which results when the nway option is not specified (the default nway):

In order for the first tree structure to work, the rgang.py script must be executable from each node that is not at the end of a branch. However, the structure of the copy mode is

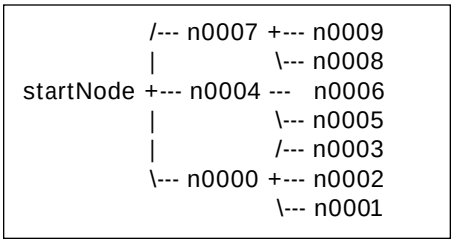

Diagram 1

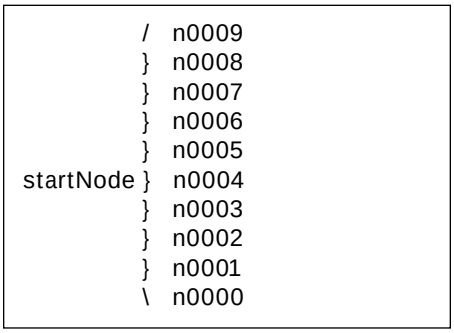

Diagram 2

such that rgang.py can be used to copy itself to the list of nodes. It copies the file to the downstream node first, then, if required, rshells to the node and executes the rgang.py script to continue the propagation.

In other words, rgang.py can be used to install itself.

The system used in the following example is our QCD cluster, which consists of a "home" node and 80 "worker" nodes. The rgang.py script is symlinked to rgang and is installed (executable) from all 81 nodes. The output in the following examples is edited to save space.

Example 1:

```
$ time rgang -nn "{,}qcd0{1-8}{01-10}" echo hi
qcd0101  =  hi
qcd0101  =  hi
qcd0102  =  hi
qcd0103  =  hi
qcd0104  =  hi
qcd0105  =  hi
qcd0106  =  hi
qcd0107  =  hi
qcd0108  =  hi
qcd0109  =  hi
qcd0110  =  hi
qcd0201  =  hi
[edit]
qcd0704  =  hi
qcd0705  =  hi
qcd0706  =  qcd0706: No route to host
qcd0707  =  hi
qcd0708  =  hi
[edit]
qcd0703  =  hi
qcd0704  =  hi
qcd0705  =  hi
qcd0706  =  qcd0706: No route to host
qcd0707  =  hi
qcd0708  =  hi
qcd0709  =  hi
[edit]
Command exited with non-zero status 1
1.45user 2.18system 0:03.84elapsed 94%CPU (0avgtext+0avgdata
0maxresident)k 0inputs+0outputs (20209major+37650minor)pagefaults 0swaps
```

In Example 1, the -nn option specifies a more compact output format. The nodes specification {,}qcd0{1-8}{01-10} results in specifying 160 nodes. (The node names result from the physical configuration of 8 shelves with 10 nodes per shelf.) At the time of the example, node qcd0706, was powered down. The output, exit status, and bulk of the 3.84 seconds reflect this "problem."

Example 2:

```
$ time rgang -nn "{,,}qcd0{1-8}{01-10}" echo hi
Traceback (innermost last):
      File "/usr/local/bin/rgang", line 688, in ?
         if __name__ == "__main__": main()
      File "/usr/local/bin/rgang", line 673, in main
```

```
        try: total_stat,ret_list = rgang( sys.argv[1:] )
      File "/usr/local/bin/rgang", line 531, in rgang
        sp_info = spawn_cmd( node_internal_info[mach_l_idx], opt, opts, args,
branch_nodes )
      File "/usr/local/bin/rgang", line 298, in spawn_cmd
        sp_info = spawn( opt['rsh'], sp_args, 0 )
      File "/usr/local/bin/rgang", line 237, in spawn
        pid = os.fork()
OSError: [Errno 11] Resource temporarily unavailable
Command exited with non-zero status 1
0.30user 0.49system 0:02.73elapsed 28%CPU (0avgtext+0avgdata
0maxresident)k 0inputs+0outputs (367major+24467minor)pagefaults 0swaps
```

In Example 2, an extra comma is added to the node specification, resulting in 240 nodes being specified. The default value for nway is 0, which causes nway to reset to the number of nodes specified, as mentioned above. This means that 240 rsh's are forked (in parallel) and the system cannot handle this! The next example shows one of the possible solutions.

Example 3:

```
$ time rgang -nn —nway 40 —skip qcd0706 "{,,}qcd0{1-8}{01-10}" echo hi
qcd0101  =  hi
qcd0102  =  hi
qcd0103  =  hi
qcd0104  =  hi
qcd0105  =  hi
qcd0106  =  hi
[edit]
qcd0807  =  hi
qcd0808  =  hi
qcd0809  =  hi
qcd0810  =  hi
1.44user 0.63system 0:02.71elapsed 76%CPU (0avgtext+0avgdata
0maxresident)k 0inputs+0outputs (5292major+8393minor)pagefaults 0swaps
```

In Example 3, the UNIX system limitations are avoided by using the nway option, and the skip option is demonstrated. The resulting status is now success.

Example 4:

```
$ time rgang -c -nn —nway=5 —skip qcd0706 "qcd0{1-8}{01-10}" \
     /boot/lsdel.out~ /tmp
qcd0101= qcd0102= qcd0103= qcd0104= [edit] qcd0809= qcd0810=
0.44user 0.31system 0:04.31elapsed 17%CPU (0avgtext+0avgdata
0maxresident)k 0inputs+0outputs (1601major+2320minor)pagefaults 0swaps
```

In Example 4, a file of size 1364734 bytes is copied from the home node, through a tree structure, to 79 nodes (connected to a big switch, all with 100Mb NICs) in 4.31 seconds. 1364734/(1024*1024) * 79 / 4.31 = 23.86 MBps

We have also implemented a tree-based tool, clstcon, which can send individual keystrokes rather than commands to all of the nodes in a cluster. When started, clstcon establishes a tree of shells across the cluster. The stdin of the root node propagates via the tree to all nodes. As an extreme example of the capabilities of the structure, an administrator can invoke a vi session across the cluster, simultaneously editing the files with the same name on all of the machines.

The rgang and clstcon utilities are great time savers.and cost. These programs have been released by Fremilab under the "Fermi Tools" umbrella. The terms and conditions are stated at *http://www.fnal.gov/fermitools/terms/TERMS_AND_CONDITIONS*

Conclusion

Within a few years, we will build and operate kiloclusters at Fermilab for work in theoretical physics. Because of our manpower limitations, we are actively seeking and developing scalable tools for the management of such large systems. For more information on our work or to try some of our tools, please see *http://qcdhome.fnal.gov/*.