ADMIN

Network & Security

Digital Special

HPC Techniques

Parallel I/O
Will smarter I/O improve your application's performance?

Lmod
Discover the power of environment modules

Benchmark Bootcamp
Using benchmarks to your advantage

Directive Encoding
Understanding compiler directives

A New World
Altair's Bill Nitzberg describes the new dual-license model for the PBS Pro workload manager.

US$ 7.95

WWW.ADMIN-MAGAZINE.COM

PBS Works 2017

HPC like never before

- New intuitive user interface – desktop, web, mobile
- Create appliances anywhere – on premises, cloud, bare metal
- Access and manage HPC with uncompromised security
- Optimize consumption and control cost
- Automate cloud bursting to handle peak workload

Learn more at **pbsworks.com/2017**

Altair | PBS Works™

ADMIN
Network & Security

HPC Techniques

Contact Info

Editor in Chief
Joe Casad, jcasad@linuxnewmedia.com

Managing Editor
Rita L Sooby, rsooby@linuxnewmedia.com

Localization & Proofreading
Amber Ankerholz, Amy Pettle, Ian Travis

Layout and Graphic Design
Dena Friesen, Lori White

Advertising
www.admin-magazine.com/Advertise
Ann Jesse, ajesse@linuxnewmedia.com
Phone: +1-785-841-8834

Publisher
Brian Osborn, bosborn@linuxnewmedia.com

Marketing Communications
Gwen Clark, gclark@linuxnewmedia.com

Customer Service / Subscription
For USA and Canada:
Email: cs@linuxnewmedia.com
Phone: 1-866-247-2802
(toll-free from the US and Canada)

www.admin-magazine.com

Printed in Germany

ADMIN is published by Linux New Media USA, LLC, 616 Kentucky St, Lawrence, KS 66044, USA.

Published in Europe by: Sparkhaus Media GmbH, Zieblandstr. 1, 80799 Munich, Germany

Dear Readers:

High-performance computing is really different. No matter what you have learned about coding or how many jobs you've held in the commercial software industry, working with the massively parallel HPC clusters used for cutting-edge science, engineering, and (increasingly) business applications takes a whole new skill set.

If you're new to HPC, you're probably realizing you have a lot to learn, and even if you're an HPC veteran, you probably learned long ago to keep your eyes open for new tips, tools, and techniques for improving HPC performance. At ADMIN magazine, we're all about helping you find the information you need. This special edition gets down in the trenches with some practical articles on working with HPC applications and clusters.

You'll also hear about an exciting new development for the HPC space: a new open source version of Altair's powerful PBS Pro workload manager.

Read on for a close-up look at real solutions for real problems in HPC.

Joe Casad
Editor in Chief
ADMIN magazine

Table of Contents

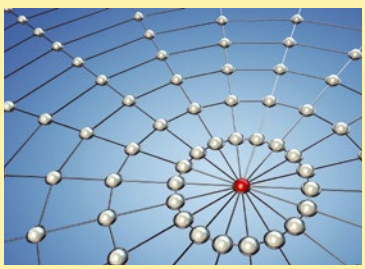
4 Parallel I/O
Even some parallel applications handle I/O serially, which can result in performance bottlenecks.

8 Virtuous Benchmarks
You'll save time and money on your HPC project if you choose your benchmarks carefully.

11 A New World
Altair recently released its popular PBS Professional workload manager under an open source license. We talk with Altair's Bill Nitzberg about PBS Pro and other recent developments in high-performance computing.

14 Directive Encoding
The OpenACC and OpenMP standards help you annotate code with compiler directives to take advantage of parallelism.

18 Lmod
Environment modules let you define the user environment and a set of tools for building your HPC application.



Improved Performance with Parallel I/O

Paths

Understanding the I/O pattern of your application is the starting point for improving its I/O performance, especially if I/O is a fairly large part of your application's run time. By Jeff Layton

Lately I've been looking at application timing to find bottlenecks or time-consuming portions of code. I've also been reading articles about improving applications by parallelization. Once again, this has led me to examine application I/O and, in particular, the fact that many applications handle I/O serially – even parallel applications – which can result in an I/O performance bottleneck.

I/O in serial applications is simple: one thread or process does it all. It doesn't have to read data and then pass it along to another process or thread. It doesn't collect data from other processes and threads and perform the I/O on their behalf. One thread. One I/O stream.

To improve application performance or to tackle larger problems, people have turned to parallelism. This means taking parts of the algorithm that can be run at the same time and doing so. Some of these code portions might have to handle I/O, which is where things can get complicated.

What it comes down to is coordinating the I/O from various threads/processes (TPs) to the filesystem. Coordination can include writing data to the appropriate location in the file or files from the various TPs so that the data files are useful and not corrupt. POSIX filesystems don't have mechanisms to help multiple TPs write to the filesystem; therefore, it is up to the application developer to code the logic for I/O with multiple TPs.

I/O from a single TP is straightforward and represents coding in everyday applications. Figure 1 shows one TP handling I/O for a single file. Here, you don't have to worry about coordinating I/O from multiple TPs because there is only one. However, if you want to run applications faster and for larger problems, which usually involves more than one TP, I/O can easily become a bottleneck. Think of Amdahl's Law [1].

Amdahl's Law says that the performance of a parallel application will be limited by the performance of the serial portions of the applications. Therefore, if your I/O remains a serial function, the performance of the application is driven by the I/O.

To better understand how Amdahl's Law works, I'll examine a theoretical application that is 80% parallelizable (20% is serial, primarily because of I/O). For one process, the wall clock time is assumed to be 1,000 seconds, which means that 200 seconds is the serial portion of the application. By varying the number of processes from 1 to 64, you can see in Figure 2 how the wall clock time on the *y*-axis is affected by the number of processes on the *x*-axis. The blue portion of each bar is the serial time, and the red portion represents time required by the parallel portion of the application. Above each bar is the speedup factor. The starting point on the left shows that the sum of the serial portion and the parallel portion is 1,000 seconds, with 20% serial (200 seconds) and 80% par-

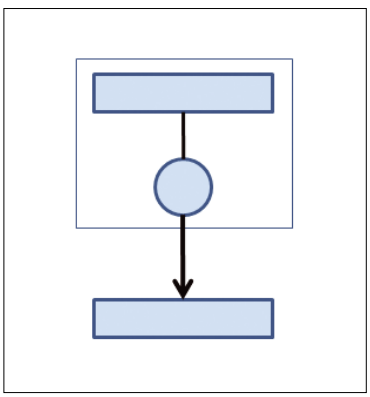


Figure 1: I/O from a single thread/process.

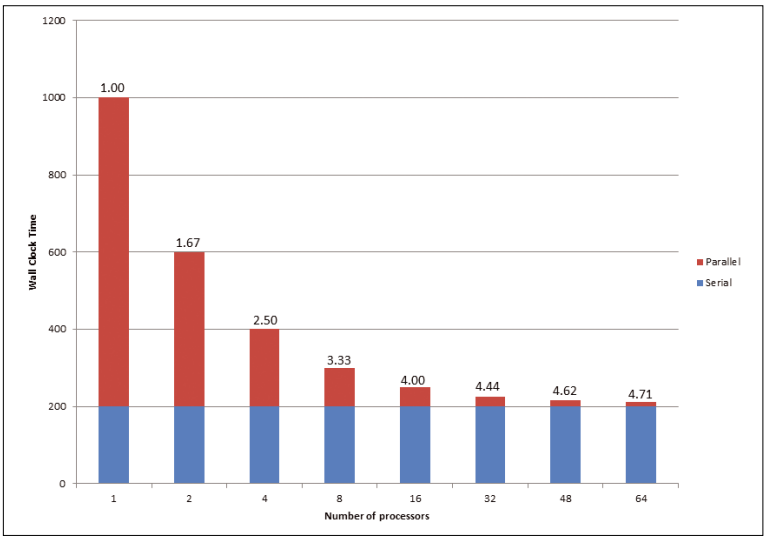


Figure 2: Influence of Amdahl's Law on a 20% serial application by number of processes.

allel (800 seconds), but with only one process. Amdahl's Law says the speedup is 1.00.

As the number of processes increase, the wall clock time of the parallel portion decreases. The speed-up increases from 1.00 with one process to 4.71 using 64 processes. Of course, the wall clock time for the serial portion of the application does not change; it stays at 200 seconds regardless of the number of processes.

As the number of processes increase, the total run time of the application approaches 200 seconds, which is the theoretical performance limit of the application, primarily because of serial I/O. The only way to improve application performance is to improve I/O performance by handling parallel I/O in the application.

Simple Parallel I/O

The first way to approach parallel I/O, and one that many applications use, is to have each TP write to its own file. The concept is simple, because there is zero coordination between TPs. All I/O is independent of all other I/O.

Figure 3 illustrates this common pattern, usually called "file-per-process." Each TP performs all I/O to its own file. You can keep them all in the same directory by using different file names or you can put them in different directories if you prefer.

As you start to think about parallel I/O, you need to consider several aspects. First, the filesystem should have the ability to keep up as parallelism increases. For the file-per-process pattern, if each TP performs I/O at a rate of 500MBps, then with four TPs the total throughput is 2GBps. The storage needs to be able to sustain this level of performance. The same is true of IOPS (input/output operations per second) and metadata performance. Parallelizing an application won't help if the storage and filesystem cannot keep up.

The second consideration is that the data files will be used outside your application. Does the data need to be post-processed (perhaps visualized) once it's created? Does the data read by the application need to be prepared by another application? If so, you need to make sure the other application creates the input files in the proper format. Think of the application workflow as a whole.

For the example in Figure 3, if you used four TPs and each handled their own I/O, you have four output files and probably four input files (you can allow each TP to read the same file). Any application that uses the output from the application will need to use all four files. If you stay with the file-per-process approach, you will likely have to stay with four TPs for any applications that use the data, which now introduces an artificial limitation into the workflow (four TPs).

One option is to write a separate application that reads the separate files and combines them into a single file (in the case of output data) or reads the input file and creates separate files for input (in the case of input data). Although this solution eliminates the limitation, it adds another step or two to the workflow.

Performing file-per-process is probably the easiest way to achieve parallel I/O, because it

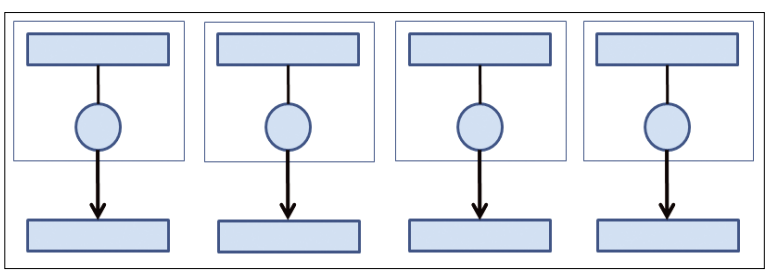


Figure 3: Parallel I/O via separate files (file-per-process).

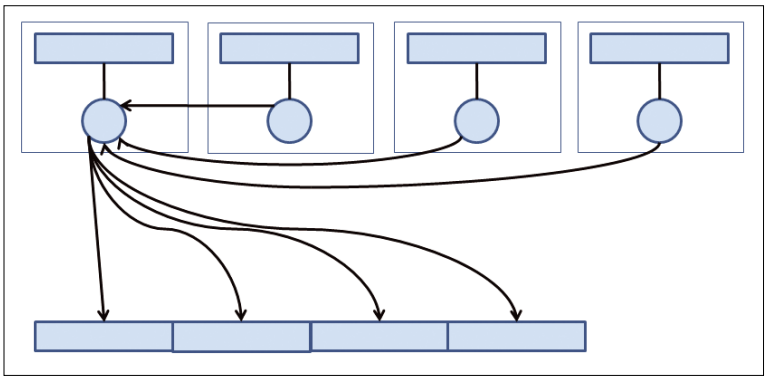


Figure 4: Multiple threads/processes doing I/O via a single thread/process.

involves less modification to the original serial or parallel application while greatly improving the I/O portion of the run time. However, as the number of TPs increases, you could end up with a large number of input and output files, making data management a pain in the neck.

Moreover, as the number of files increases, the metadata performance of the filesystem comes under increasing pressure. Instead of a few files, the filesystem now has to contend with thousands of files all doing a series of `open()`, `read()` or `write()`, `close()` [or even `lseek()`] operations at the same time.

Single Thread/Process I/O

Another option for the parallel I/O problem is to use one TP to handle all I/O. This solution really isn't parallel I/O, but it is a common solution that avoids the complications of having multiple TPs write to a single file. This option also solves the data management problem because everything is in a single file – both input and output – and eliminates the need for creating auxiliary applications to either split up a file into multiple pieces or combine multiple pieces into a single file. However, it does not improve I/O performance, because I/O does not take place in parallel.

You can see in Figure 4 that the first TP (the "I/O thread/process") takes care of I/O to the file(s). If any of the other TPs want to write data, they send it to the I/O TP, which then writes to the file. To read a file, the I/O TP reads the data and then sends it to the appropriate TP.

Using the single I/O TP approach can require some extra coding to read and write the data, but many applications use this approach because it simplifies the I/O pattern in the application: You only have to look at the I/O pattern of one TP.

Better Parallel I/O

A better approach for handling parallel I/O is shown in Figure 5. In this approach, every TP writes to the same file, but to different "section" of it. Because the sections are contiguous, you have no chance of one TP overwriting the data from a neighboring TP. For this approach to work, you need a common shared filesystem that all TPs can access (NFS anyone?).

One of the challenges of this approach is that the data from each TP has to have its own "section" of the file. One TP cannot cross over into the section of another TP (don't cross the streams), or you might end up with data corruption. The moral is, be sure you know what you are doing or you will corrupt the data file.

Also note that, most likely, if you write data using *N* number of TPs, you will have to keep using that many TPs for any applications later in the workflow. For some problems, this setup might not be convenient or even possible.

Developers of several applications have taken a different approach: using several TPs to process the I/O. In this case, each TP writes a certain part of the output file. Typically the number of I/O TPs is constant, which helps any preprocessing or post-processing applications in the workflow.

One problem with the single I/O TP or fixed number of I/O TPs approaches is that reading or writing data from a specific section often is not easy to accomplish; consequently, a solution has been sought that allows each TP to read/write data from anywhere in the file, hopefully, without stepping on each others' toes.

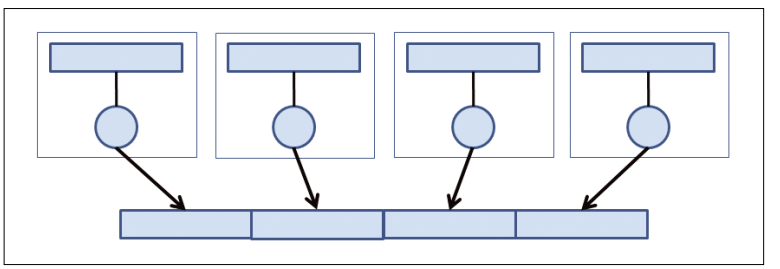


Figure 5: Multiple threads/processes performing I/O to a single file.

MPI-IO

Over time, MPI (Message Passing Interface) [2] became popular and researchers began thinking of how to handle parallel I/O for MPI applications better. In MPI-2, something called MPI-IO was adopted. MPI-IO is a set of functions that abstract I/O for MPI applications on distributed systems. It allows the application to perform I/O in parallel much the same way MPI sends and receives messages.

Typically each process in the MPI communicator participates in the I/O, but it's not required. How each process writes to a file is up to the developer. Although it is far beyond the scope of this article to discuss MPI-IO, a number of tutorials [3], documents [4], and even books [5] online can help you get started.

High-Level Libraries

The last approach to parallel I/O I'm going to mention is high-level libraries that you can use for storing data (read and write), These libraries take care of the parallel I/O "under the covers," so to speak. Two options are worth mentioning: HDF5 and Parallel NetCDF.

Parallel HDF5

HDF5 [6] is a file format that can be used to store large amounts of data in an organized fashion. HDF stands for "Hierarchical Data Format" and the "5" indicates the particular file format. HDF5 files are portable, so you can write an HDF5 file on one system and read it on another. Moreover, a number of languages have HDF5 APIs, including C, C++, Fortran, Python, Perl, Julia, Go, Matlab, R, Scilab, Octave, and Lua, to name a few.

Parallel HDF5 [7] uses MPI-IO to handle I/O to the storage system. To get the best performance reading and writing to HDF5 files, you can tune various aspects of MPI-IO, as well as HDF5 parameters for the underlying filesystem.

Parallel NetCDF

Another portable file format is NetCDF [8]. The current version 4 allows the use of the HDF5 file format. APIs for NetCDF include C, C++, Fortran, Python, Java, Perl, Matlab, Octave, and more. As with HDF5, NetCDF has a parallel version, Parallel-NetCDF [9], which also uses MPI-IO. This version is based on NetCDF 3 and was developed by Argonne Labs. To implement parallel I/O with NetCDF 4, you need to use HDF5 capability and make sure HDF5 was built with MPI-IO.

Recommendations

If you have an application that handles I/O in a serial fashion and the I/O is a significant portion of your run time, you could benefit by modifying the application to perform parallel I/O. The fun part is deciding how you should do it.

I recommend you start very simply and with a small-ish number of cores. I would use the file-per-process approach in which each TP performs I/O to its own file. This solution is really only suitable for small numbers of TPs, but it is fairly simple to code; be sure to have unique file names for each TP. This approach places more burden on the pre-processing and post-processing tools, but the application itself will see better I/O performance.

The second approach I would take is to use a high-level library such Parallel HDF5. You can use MPI-IO underneath the library to get improved I/O performance, but it might require some tuning. The benefit of using a high-level library is that you get a common, portable format across platforms with some possible I/O performance improvement.

After using high-level libraries, I would say that using MPI-IO or confining I/O to one TP are your choices. Writing applications for MPI-IO can be difficult, but it also can reap the biggest I/O performance boost. Having one TP perform all of the I/O can be a little complicated as well, but it is a very common I/O pattern for parallel applications.

Don't be afraid of jumping into parallel I/O with both feet, because you can get some really wonderful performance improvements. ■

Info

[1] Amdahl's Law: [http://www.admin-magazine.com/HPC/Articles/Failure-to-Scale]

[2] MPI: [https://computing.llnl.gov/tutorials/mpi/]

[3] MPI and MPI-IO training tutorial: [https://www.hpc.ntnu.no/display/hpc/MPI+and+MPI+IO+Training+Tutorial]

[4] MPI-IO: [http://beige.ucs.indiana.edu/I590/node86.html]

[5] Liao, Wei-keng, and Rajeev Thakur. MPI-IO. In: *High Performance Parallel I/O*, chapter 13. Chapman and Hall/CRC, 2014, pp. 157-167, [http://www.mcs.anl.gov/papers/P5162-0714.pdf]

[6] HDF5: [http://www.hdfgroup.org/]

[7] Parallel HDF5: [https://www.hdfgroup.org/HDF5/PHDF5/]

[8] NetCDF: [https://en.wikipedia.org/wiki/NetCDF]

[9] Parallel-NetCDF: [https://en.wikipedia.org/wiki/NetCDF#Parallel-NetCDF]

Lead Image © Tom De Spiegelaere, 123RF.com

Using benchmarks to your advantage

Node Check

A collection of single- and multinode performance benchmarks is an excellent place to start when debugging a user's application that isn't running well. By Jeff Layton

From my perspective as a user, customer, developer, administrator, and vendor, one of the most contentious issues in the HPC industry has been benchmarks.

As a user and customer, I used benchmarks to get an idea of performance and to compare product metrics, such as performance/price or performance/watt. After the system is installed, the benchmarks are often re-run to make sure the system meets the vendor's guarantees. On the vendor side, I used benchmarks to improve my understanding of how new systems performed, so I could make good recommendations to customers.

Because of the enormous amount of effort required for benchmarks, both sides – customer and vendor – view benchmarks as a necessary evil. Neither side really wants them; nonetheless, they use them. Perhaps I can find a way to use them that isn't so evil. To begin this quest, I'll examine the benchmarks typically run when installing a system.

Installation Benchmarks

During installation, the system is reconstructed on the customer site, which includes racking and cabling the hardware and installing or checking the system software.

Once the system is up and running, benchmarks are run to determine two things: Are the nodes and network functioning correctly? Is system performance as promised?

In my experience, to accomplish these two goals, you should run a series of benchmarks that start with single-node runs and progress to groups of nodes of various sizes.

Single-Node Runs

I like to start with the individual nodes and then work up, so I begin by running the exact same tests on all of the nodes as close to the same time as possible. The tests should run fairly quickly yet stress various components of the system. For example, they should definitely stress the processor(s) and memory, especially the bandwidth. I would recommend running single-core tests and tests that use all of the cores (i.e., MPI or OpenMP).

A number of benchmarks are available for you to run. The ones I like are the NAS Parallel Benchmarks (NPB) [1]. NPB is a set of benchmarks that cover a wide range of applications, primarily from the CFD (Computational Fluid Dynamics) field. I've found they really stress the CPU, memory bandwidth, and network in various ways. OpenMP and MPI versions of NPB "classes" allow you to run different data sizes. Plus, they are very easy to build and run, and the output is easy to interpret.

The NASA website [2] provides the following details on the NPB benchmarks.

Five kernel benchmarks:

- IS – Sorts small integers using the bucket sort. Typically uses random memory access.
- EP – Embarrassingly parallel application. Generates independent Gaussian random variates using the Marsaglia polar method.
- CG – Estimates the smallest eigenvalue of a large sparse symmetric positive-definite matrix using the inverse iteration with the conjugate gradient method as a subroutine for solving systems of linear equations. Uses irregular memory access and communication.
- MG – Approximates the solution of a three-dimensional discrete Poisson equation using the V-cycle multigrid method on a sequence of meshes. Exhibits both long- and short-distance communication and is memory intensive.
- FT – Solves a three-dimensional partial differential equation (PDE) using the fast Fourier

transform. Uses a great deal of all-to-all communication.

Three pseudo-applications:

- BT – Solves a synthetic system of nonlinear PDEs using a block tri-diagonal solver.
- SP – Solves a synthetic system of nonlinear PDEs using a scalar penta-diagonal solver.
- LU – Solves a synthetic system of nonlinear PDEs using symmetric successive over-relaxation (SSOR). Also referred to as a Lower-Upper Gauss–Seidel solver.

These tests have both OpenMP and MPI versions, and a "multi-zone" version of the pseudo-applications can be run in a hybrid mode (i.e., MPI/OpenMP).

The benchmark classes in Table 1 indicate the size of the problem being examined and correlate with the amount of memory used and the amount of time needed to complete. NPB has been released three times, each undergoing several versions as bugs were found or improvements were introduced. As of this writing, the latest version is 3.3.1 for both NPB and NPB-MZ (multizone).

Benchmark results are usually expressed in terms of how much (wall clock) time it takes to run and in GFLOPS (10^9 floating point operations per second) or MFLOPS (10^6 floating point operations per second). For example, Listing 1 presents the output of the MG benchmark (NPB 3.3.1, GCC compilers, OpenMPI, single socket with four cores with four hyperthreading cores for eight total cores, Class C).

The output says it took 35.44 seconds to run, using a total of 4,393.44 MOPS (Mop/s in Listing 1, million operations per second; =4.393 GFLOPS).

For testing (benchmarking), I select a subset of the NPB benchmarks and classes, execute single-node runs (either OpenMP or MPI) on all of the nodes roughly at the same time, and name the output files to match the node name. To collect the output from all of the runs, I use simple Bash or Python scripts.

With this data in hand, I first look for performance outliers. To begin, I compute the average (arithmetic mean) and standard deviation of all of the results for each test. If the standard deviation is a significant percentage of the average, I then plot the data on a graph of performance versus node number, which I inspect visually for outliers.

From the plot, I can mark some nodes as outliers that need to be re-tested and possibly triaged. Next, I remove the data of the outlier nodes from the totals and recompute the average and standard deviation, repeating the outlier identification process. At some point, one hopes the standard deviation becomes a small percentage of the average, so I can stop the testing process with a set of good nodes and a set of outlier nodes.

For example, I might start with a performance standard deviation target of +/-5 percent of the average. (Note that 5 percent is an example, not a hard and fast number.) If the computed standard deviation is greater than 5 percent, I will plot the results and start choosing nodes outside of this deviation. Next, I recompute the average and standard deviation of the reduced set and repeat until I reach the target 5 percent deviation.

With the set of outlier nodes, I re-run the benchmarks one or two more times to see if the performance changes. If it does not, then I triage the nodes (up to and including replacement).

The last step is probably one of the most critical steps you can take, and it goes to the heart of this article. Be sure to store the single-node results somewhere you can easily retrieve them. Also, store the the source, and even the binaries, with the information on how you built the code, including software versions.

Table 1: NPB Benchmark Classes

Class	Test Size	Application
S	Small	Quick tests
W	Workstation	From the 1990s
A, B, C	Standard	4x size increases going from one class to the next
D, E, F	Large	~16x size increases from each of the previous classes

Listing 1: MG Benchmark Output

```
NAS Parallel Benchmarks 3.3 -- MG Benchmark

No input file. Using compiled defaults
Size:  512x 512x 512  (class C)
Iterations:   20
Number of processes:     8

Initialization time:           5.245 seconds

 iter    1
 iter    5
 iter   10
 iter   15
 iter   20

Benchmark completed
VERIFICATION SUCCESSFUL
L2 Norm is  0.5706732285739E-06
Error is    0.1345119360807E-12


MG Benchmark Completed.
Class           =                        C
Size            =            512x 512x 512
Iterations      =                       20
Time in seconds =                    35.44
Total processes =                        8
Compiled procs  =                        8
Mop/s total     =                  4393.44
Mop/s/process   =                   549.18
Operation type  =            floating point
Verification    =                SUCCESSFUL
Version         =                    3.3.1
Compile date    =              28 Nov 2014

...
```

Small Node Groups

After the single-node runs are done, I test small groups of nodes. You can either arbitrarily pick the number of nodes per group to test, or you can group the nodes together so that they all belong to a single switch. Generally, I try to run four nodes per group to keep things simple. In these groups, I run tests with both a single core per node and all the cores per node, allowing me to stress the nodes in different ways. The goal of small-node-group testing is to start introducing network performance as an overall parameter.
For these runs, you have to use the MPI version of the NPB tests, and I would run the same tests as used in the single-node runs.
I recommend running two different classes for these small node groups, beginning with A or B, to stress the network by taking a small problem and spreading it across a number of processes. However, real systems are seldom run this way, because it is not an efficient use of the system. Therefore, I would also run the largest class problem possible to stress the memory, CPU, and network.
After running these tests, you again perform a statistical analysis on the results in the exact same manner as described for the single-node runs: compute the average and standard deviation of the tests, look for outliers in the data, run more tests on those groups, and perhaps triage the nodes if needed.
I would also recommend comparing the nodes in this outlier groups to the outliers in the single-node tests to look for correlation.
As with the single-node tests, be sure to store the results somewhere you can easily retrieve them, along with the source and binaries and how you built the code, including versions.

Larger Node Groups

After running small groups of nodes you can run larger groups by combining the smaller groups. The most important thing to remember is to store the results once you are done.
You can repeat the process of testing larger and larger node groups until you reach the entire cluster. Sometimes this is useful if you are trying to do a TOP500 run, because you can leave slower nodes out of the run that hurt the final result.
After you finish all these tests you will have a fairly extensive database of benchmark results; it includes the results from standard benchmarks for all of the individual nodes and groups of nodes, as well as a history of outlier nodes relative to the others.

Database of Results

The most common problem I encountered as an admin is user applications that do not run or run poorly. One of the first things to do in tackling these problems is test the nodes that seem to be causing the performance problems. To do this, you need to know what kind of performance to expect from the nodes. Don't forget that you have a very nice database of test results you can use for this testing.
Of course, these tests might not "tickle" the node(s) in the same way a user application does, but at least you have a starting point.
In addition to debugging the node itself, the database results can help track down network problems.
With the node group tests from the database in hand, you can re-run the small node group tests across the set of nodes you suspect are not performing well and see how the results compare with the database.
Admins also update system software from time to time (e.g., a security update or a new version of a compiler or library). To determine whether the nodes are performing well after the update, you can simply re-run the tests and compare the results to the database.
After a firmware upgrade on nodes or switches, I definitely recommend re-running all of the tests – from single-nodes to larger node groups. Again, don't forget to store the new results as a new baseline. If the results are worse and triaging does not turn up much, you might have to roll back the firmware version while you debug the updated firmware with the manufacturer(s).
A great way to use these benchmark results is to re-run the tests on nodes periodically by creating some simple jobs, running them, and recording and comparing the results. A simple tool can parse the benchmark results and throw them into the database for comparison with the old results, and you can even use statistical methods in the comparison.

Summary

Debugging a user's application when it isn't running well is always difficult; however, you have an advantage if you have a set of baseline performance benchmarks in your back pocket.
An excellent way to start checking for problems is to check the nodes they use. In particular, I would briefly take the nodes out of production and check their performance by repeating the exact same tests used to create the database and comparing the results to the database. ■

Info

[1] NAS Parallel Benchmarks: [http://en.wikipedia.org/wiki/NAS_Parallel_Benchmarks]
[2] NASA NPBs: [http://www.nas.nasa.gov/publications/npb.html]

An open source license for PBS Pro

A New World

The powerful PBS Pro workload manager gets a new open source license. By Joe Casad

ADMIN Magazine: PBS Professional has been a leading workload manager in the HPC space for years. What made you decide to go with an open source licensing model?

Bill Nitzberg: The HPC world is really two separate HPC worlds: the public sector (focused on research and academia) and the private sector (focused on commercial endeavors). On the research side, organizations are eager to take risks and be early adopters, all in the name of exploration; plus, researchers are natural collaborators. On the commercial side, organizations are more risk averse; they want proven solutions (not half-working, bleeding-edge tools); in addition, they are natural competitors. These differences have meant that the public sector has an affinity for open source software and the private sector has a preference for (sometimes even demands) commercially licensed software. This dichotomy has really hindered innovation, by restricting the flow between these two worlds. This is a huge lost opportunity for the whole HPC world. By dual-licensing PBS Professional – offering it with an open source option as well as with a commercial license option – we hope to marry these separate HPC worlds, to re-target efforts towards pushing the envelope instead of wasting effort re-implementing duplicate capabilities, and to really advance the state of the art in scheduling, bringing together the public sector's innovations with the private sector's enterprise know-how.

AM: For our readers who aren't familiar, what is PBS Professional and what does it do? How does it differ from other similar tools in the market?

BN: PBS Professional is "job scheduler" or "workload manager" software for HPC clusters and clouds. It is infrastructure middleware that efficiently optimizes the allocation and use of HPC resources (CPUs, memory, GPUs, software licenses, electric power, …) ensuring that resources are used wisely and enterprise policies are met. Engineers and researchers submit "jobs" (engineering simulations, weather models, gene matching, …) and PBS Pro schedules the right job at the right time on the right resources, automatically, handling prioritization, traffic control, fault recovery, notifications, and reporting. System administrators configure site-wide system use policies and can easily manage complex HPC systems with thousands of users and millions of cores, letting PBS Pro ensure maximum utilization, with minimum interruption. Award-winning PBS Professional is fast and powerful and is designed to improve productivity, optimize utilization and efficiency, and simplify HPC administration. It automates job scheduling, management, monitoring, and reporting. PBS Pro has been proven for over 20 years at thousands of global sites; it is the trusted solution for complex TOP500 systems and small cluster owners alike.

AM: Open source means community. Vendors often open-source their products to encourage participation from third-party developers and volunteers. What kinds of resources do you have now for developers, testers, and other volunteers to get involved with PBS Pro? What kinds of steps are you taking to encourage more participation?

BN: Altair has made a big investment into the open source community and continues to do so. We have opened the full core of PBS Pro, not just a weak subset or an older version. We have reorganized to behave as one of the many contributors and have added staff to support the community as well. We have implemented community-accepted practices and tools, such as OSI-approved licenses and using GitHub and JIRA. We are focused on building a viable and sustainable

community that is aggressively open and inclusive while maintaining a high level of respect and professionalism towards one another.
PBS Professional is a community effort providing a variety of ways to engage. We have created a forum to facilitate communication between users and developers alike. The contributor's portal provides a huge amount of step-by-step instructions, details on roadmaps and processes, and more.

AM: PBS Pro will actually have a dual-licensing model. Tell us about the other side. Do you plan to continue to maintain a closed-source version of PBS Professional? What's different about the premium edition? What are some scenarios in which you would recommend the commercial version?

BN: Our goal is to have a common core but to continue to provide a commercially hardened version for customers who need a bullet-proof enterprise edition, and where commercial license terms are desired or required. For organizations whose infrastructure has depended on PBS Pro for many years, we want to ensure that there is no change and that they still get the same hardened code, fantastic support, and critical updates that they have been getting for many years. Therefore, the commercial version will have significantly more testing and additional quality assurance.

AM: How do you see the HPC space in general, compared to where it was a few years ago? And where do you see it going in the future?

BN: The HPC space has become interesting again… it feels as if we are in the middle of a Cambrian tech-explosion in both hardware and software. On the hardware side, what was very homogeneous a few years ago (clusters of Intel architecture systems where the big differences were pretty minor – Ethernet vs. Infiniband and how much memory to put in each node) is now a hotbed of innovation with Xeon Phi, GPUs, ARM, POWER, SSDs; even FPGAs are back. On the software side, Linux had become the de facto HPC operating system and MPI + OpenMP was the dominant programming model. Although the Linux kernel remains the HPC operating system of choice, the variety of Linux distributions has greatly expanded, and now includes variants provided by cloud providers; the use of containers (aka Docker) has allowed each individual application to precisely craft what amounts to an application-specific Linux distribution. Programming models are also much more interesting, making use not only of cloud APIs but also, in a major way, microservices, containers, and the elephant in the room – big data with Hadoop (although even that's feeling old), not to mention Spark. There's more every day.
This explosion of new technologies is being driven by collaboration, and the collaboration is powered by open source. Most of the new tools mentioned above are, in fact, open source. The collaborative power of open source not only helps explore new ideas, but it's accelerating adoption. (This is where our dual-license support naturally fits in, and with PBS Pro as part of the Linux Foundation's OpenHPC initiative, it truly feels as if there is the potential to change the landscape of the HPC supercomputing space by collaborating on a common set of tools that anyone can use.)

AM: High-performance computing used to be the preserve of college professors and high-end engineering and simulation teams. Now we hear a lot about Big Data: corporations using HPC techniques to analyze data and look for patterns. Do you see PBS Professional, and Altair's products in general, finding their way into this Big Data space, or will your primary focus remain with science and engineering?

BN: Science and engineering are being driven more and more by data in addition to computation, so even in the traditional HPC areas, Big Data is becoming as important as Big Compute. So, yes, Altair sees Big Data as a key driver for our technologies (both inside and outside engineering). I believe the common definition of HPC will include Big Data in the near future. (In fact, my definition of HPC is "pushing the limits of computing slightly beyond where it works really well," and, as such, Big Data fits right in there today.) Interestingly, some sectors of HPC, such as weather and climate, have always had a Big Data problem – huge datasets, streaming in real time, messy accuracy – but they have been around long enough that they have captured their big data into efficient HPC workflows. Even modern engineering simulations now include Big Data aspects; for example, with Altair's HyperStudy design exploration tool, a single engineer on a single aspect of one design can end up with datasets topping 500 runs. The new users of data in other industries are still grappling with how to best manage it. At Altair, we're focusing on two parts of the convergence: one is scheduling combined workloads (mixing Big Data and Big Compute) using PBS Pro; the other is analytics on Big Data using Altair's new Envision business intelligence platform. Envision is a modern cloud-based

analytics solution and is the power behind PBS Analytics and Altair's Software Asset Optimization (SAO), both of which are a part of the PBS Works suite. PBS Analytics allows administrators to visualize historical HPC resource usage for optimized returns on investments. Altair's SAO provides organizations the ability to measure and analyze utilization of their software assets to ensure that technology investments are used in the best way possible.

AM: The last time we talked, Altair had just launched HyperWorks Unlimited. How's the cloud business going? How will the licensing changes and other developments with the PBS family of tools affect the cloud operation?

BN: Our cloud business is ramping up. We now have HyperWorks Unlimited appliances that are available on the private and public clouds, and we have announced two new cloud solutions: PBSCloud.io and Inspire Unlimited.
HyperWorks Unlimited is a state-of-the art appliance available in both physical and virtual formats, offering unlimited use of all Altair software. Altair's cloud appliances address the unique needs of enterprises by providing access to an HPC infrastructure at an affordable cost. This allows engineers and scientists access to hardware, software, and HPC support that enables robust product designs.
PBSCloud.io is an SaaS platform that easily creates, manages, deploys, and monitors HPC appliances for public clouds, private clouds, and bare metal. Solution architects are able to define fully engineered HPC appliances whether it is in the cloud or on on-premise hardware. A vast variety of middleware is supported, including applications from Altair, third-party solutions, and in-house software. In addition, engineers and scientists can easily deploy and use appliances via a self-service model.
Inspire Unlimited is the cloud platform that will host solidThinking Inspire. Inspire is a topology optimization software that allows engineers and designers to create and investigate structurally efficient concepts quickly and easily. Other solidThinking solutions including Evolve and Click2Cast will eventually become a part of it, as well.

AM: Is there anything Altair is working on now that you're particularly excited about? A new product? Or new and interesting use cases that have evolved around the Altair toolkit? What should Altair-watchers be watching for?

BN: The 2017 release of PBS Works is really great – fast, scalable, beautiful – it includes everything from totally new modules (like web-based real-time workload monitoring) to small touches (like native iOS notifications). One really exciting new module is the workload simulator. The simulator allows a system administrator to test and verify infrastructure changes before committing to the changes, and especially before putting those changes into production (Figure 1). For example, let's say some jobs are taking too long to complete because they appear to languish in the queues while waiting for resources. You believe that adding resources will solve the problem, but how do you figure out exactly what to buy, and then how do you convince your management that it's the right thing to do? The simulator lets you run a study based on your site's real workload (for example, last quarter's workload), varying the number of resources added. Using it, you can show the effect of adding big memory nodes vs. GPU nodes vs. software licenses vs. whatever on job waiting times. With the simulator, you can not only determine that you should buy 12 big memory nodes plus one GPU node to ensure your high-priority jobs get done overnight, but you can take the charts to your management to justify the budget and explain exactly what is needed and why and how much of an impact it will have. ■

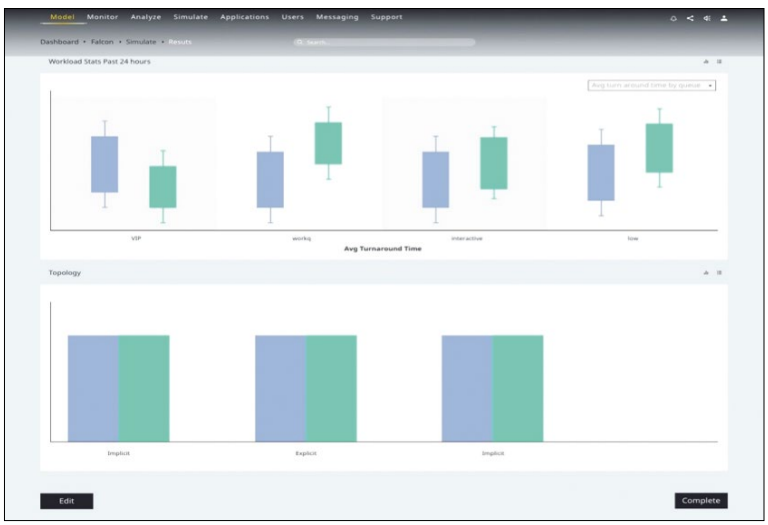
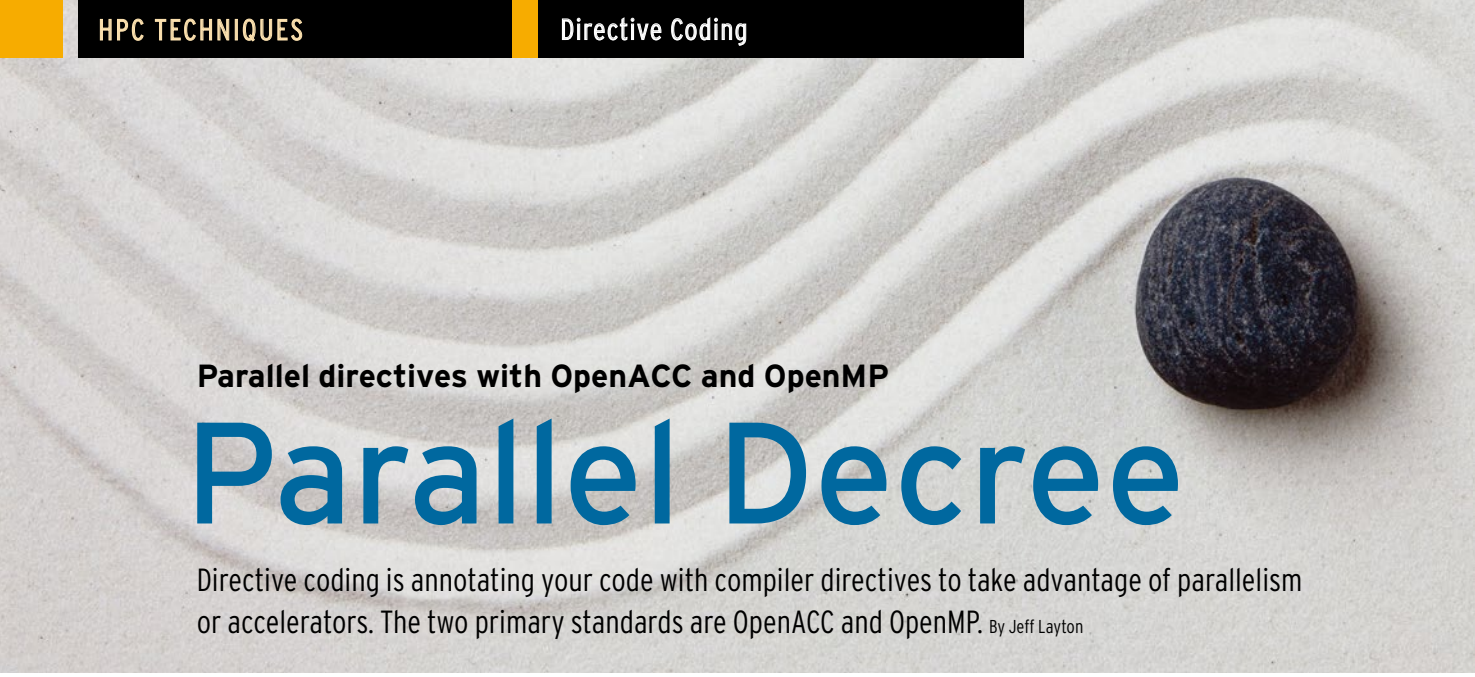


Figure 1: Proposed changes can be verified via simulation before being deployed in production; the top graphs show how VIP job turnaround improves (blue actual vs. green simulated with proposed changes), and it shows that other job types show a slight degradation in turnaround.



Parallel directives with OpenACC and OpenMP

Parallel Decree

Directive coding is annotating your code with compiler directives to take advantage of parallelism or accelerators. The two primary standards are OpenACC and OpenMP. By Jeff Layton

Ellie Arroway: *You found the primer.*

S. R. Hadden: *Clever girl! Lights. … Pages and pages of data. Over 63 thousand in all, and on the perimeter of each …*

Ellie: *… alignment symbols, registration marks, but they don't line up.*

Hadden: *They do, if you think like a Vegan. An alien intelligence is going to be more advanced. That means efficiency functioning on multiple levels and in multiple dimensions.*

Ellie: *Yes! Of course. Where is the primer?*

Hadden: *You'll see. Every three-dimensional page contains a piece of the primer; there it was all the time, staring you in the face. Buried within the message itself, is the key …*

– Contact (1997)

I've always loved this interaction from the movie *Contact* [1]. To me, it illustrates that you constantly have to think about all aspects of a problem and can't focus on just one thing too long. You have to spin the problem, turn it around, and turn it inside out to understand the problem and solve it – or at least take a step in that direction.

Two current trends in the HPC world are intersecting: using more than one core and using accelerators. They can result in lots of opportunities, but sometimes you need to turn a problem around and examine it from a different perspective to find an approach to solving the problem that comes from the intersection of possible solutions. In the HPC world, opportunities can mean faster performance (it does stand for "high performance," after all), easier or simpler programming, more portable code, or perhaps all three. To better understand the interaction, I'll examine the trend of helping coders get past using only a single core.

Using More than One Core

XSEDE [2] is an organization, primarily of colleges and universities, that integrates resources and services, mostly around HPC, and makes them easier to use and share. XSEDE's services and tools allow the various facilities to be federated and shared. You can see a list of the resources [3] on their website. It's definitely not a small organization, having well over 12x10^15 floating-point operations per second (12PFLOPS) of peak performance in aggregate. At the recent XSEDE conference [4] during a panel session [5], it was stated that 30% of the jobs run through XSEDE in 2012 only used a single core. From another presentation [6], only 70% used 16 cores (about a single node). There at least has to be an easy way to accelerate the single-core jobs to use a good percentage of the cores in a single node. Perhaps some simple "hints" or directives can be added to code to tell the compiler that it can create a binary that takes advantage of all of the computational capability available.

Accelerators

At the same time, HPC has had an insatiable appetite for more performance. CPUs have evolved to include several tiers of cache, from L1 to L2 to L3 (and even L4), before going to main memory. Current CPUs also have several cores per processor. Although CPU improvements have brought wonderful gains in performance, the desire for even more performance has been strong, leading to the adoption of co-processors, which take on some of the computational load to improve application performance. These coprocessors, also referred to as "accelerators," can take many shapes: graphical processing units (GPUs), many-core CPU processors like Intel's Xeon Phi, digital signal processors (DSPs), and even field-programmable gate array (FPGAs).

All of this hardware has been added in the name of better performance. Meanwhile, applications and tools have evolved to take advantage of the extra hardware, with applications like OpenMP [7] using the hardware on a single node or the message-passing interface (MPI) [8] taking advantage of the extra processing power across independent nodes.

Certain applications or parts of applications can be rewritten to use these accelerators, greatly increasing their performance. However, each accelerator has its own unique properties, so applications have to be written for that specific accelerator. How about killing two birds with one technology? Is it possible to have a simple way to help people write code that uses multiple cores or accelerators or both? Wouldn't it be nice to have compiler directives that tell compilers what sections of code could be built for accelerators, including extra CPU cores, and build the code for the targeted accelerator? Turns out a couple of directives are available.

OpenACC

The first set of compiler directives is called OpenACC [9], which was started by Cray, CAPS, Nvidia, and PGI (Portland Group) and is somewhat similar to OpenMP in that you annotate your code with a series of comments that the compiler interprets as directives to build sections of code for the accelerator. OpenACC originally targeted GPUs from Nvidia and AMD (ATI), but it was expanded to target other accelerators, possibly including other CPU cores (multicore processors), at some point.

OpenACC has two major versions: Version 1.0 of the standard [10], announced November 14, 2011, included a number of directives for coding accelerators (initially GPUs), after which, OpenACC compilers very quickly became available. Version 2.0 [11] was finalized on June 2013, although a preview was posted [12] on November 12, 2012. It added some new capabilities and expanded functionality acquired since version 1.0 was released. Work is ongoing for the next OpenACC standard, as well, and OpenACC 2.6 is tentatively scheduled for completion by mid-2017.

As mentioned previously, OpenACC is a set of directives you add to your code as comments. An OpenACC compiler will interpret these as directives, but if the compiler is not OpenACC-ready, it will think they are just comments and ignore them. This makes it easy to create portable code that can be used with a variety of compilers. The directives cover a fairly large range of capability that tells compilers to create code that does a wide range of tasks, including:

- Initiate the accelerator startup/shutdown.
- Manage program, data transfer, or both between the CPU and the accelerator. (Note: At this time, OpenACC assumes that the memory in the accelerator is distinct from the CPU, requiring data transfer.)
- Manage the work between the accelerator and the CPU.

On the basis of the directives, the compilers generate the best code possible, but it is up to the programmer to tune their code to take advantage of the accelerators. Because accelerators target parallel computations, you can imagine that application code sections that target accelerators include both coarse-grained and fine-grained parallelism. Coarse-grained parallelism allows multiple executions in the accelerator at the same time, whereas fine-grained parallelism includes threads of execution within a single execution unit, such as SIMD and vector operations. Moreover, accelerators are good candidates for work-sharing loops, or "kernel" regions, wherein one or more loops are executed as kernels. (Generically, a kernel is a small section of code.)

The syntax for directives is pretty simple. For Fortran, the directive looks like:

```
!$acc directive [clause [, clause] ...]
```

From free-format Fortran 90 onward, `!` is a comment. For C, the directive looks like:

```
#pragma acc directive [?
 clause [, clause] ...]
```

OpenACC directives fall into several categories:

- Accelerator `parallel` region/`kernels` directives
- Loop directives
- Data declaration directives
- Data region directives
- Cache directives
- Wait/update directives
- Environment variables

Although I won't go through all of the directives and clauses, I'll look at a couple to get a feel for what they look like and do.

The first directive or construct is a `parallel` construct. In Fortran, the code looks something like this:

```
!$acc parallel [clause [, clause] ...]
< structured code block >
!$acc end parallel
```

Notice that in Fortran you have to insert a directive that tells where the parallel region ends. In C, the code looks like this:

```
#pragma acc directive [?
 clause [, clause] ...]
< structured code block >
```

This directive tells the compiler to create code where gangs of worker threads are executing the "structured code block" in parallel on the accelerator. [Note: In CUDA, a "gang of workers" is a CUDA block of threads.] One worker in each gang begins executing the code in the structured block. The number of gangs and the number of workers in each gang remain constant for the duration of the parallel region.
The second directive is a `kernels` directive. In Fortran, the directive looks like this:

```
!$acc kernels [clause [, clause] ... ]
< structured code block>
!$acc end kernels
< structured code block >
```

In C, the same thing looks like:

```
#pragma acc kernels [clause [, clause] ...]
< structured code block >
```

The `kernels` directive tells the compiler that the structured code block has a region that can be compiled into a sequence of kernels for execution on the accelerator. It is similar to the `parallel` directive, but the loops in the kernels will be independent kernels rather than one large kernel. These independent kernels and associated data transfers may overlap with other kernels. A simple example of this directive (in Fortran) is shown in Listing 1. With the simple `kernels` directive, the compiler creates a kernel from the first loop and a second kernel from the second loop (i.e., they are independent). These kernels can then be run on the accelerator.
You can find a lot of introductory materials on the web about OpenACC, as well as some YouTube videos [13] that walk through OpenACC with examples.

Listing 1: Fortran kernels Directive

```
!$acc kernels
    do i=1,n
       a(i) = 0.0
       b(i) = 1.0
       c(i) = 2.0
    end do

    do i=1,n
       a(i) = b(i) + c(i)
    end do
!$acc end kernels
```

OpenMP

OpenMP [14] was the first set of directives developed that helped the compiler find regions of code that could be run in parallel on shared memory systems. The last bit, "shared memory systems," is important. OpenACC handles data movement to and from accelerators that each can have their own memory, whereas OpenMP has to use shared memory. Today's multicore and multisocket systems are shared memory, so that's usually not an issue.
OpenMP started in 1997 with version 1.0 for Fortran. Version 4.0 of the specification, released in July 2013, has some directives that allow for the use of accelerators. OpenMP 4.5, released November 2015, added the ability to divide iterations of a loop into tasks and to express dependencies in parallelized loops, among other improvements. (Version 4.5 was released after this article was originally written, so all examples use v4.0 capabilities.) The nonprofit OpenMP consortium, which manages OpenMP, is also working on new directives that work with accelerators [15]. Remember that in the future, OpenACC may be targeting CPUs as well as accelerators. Because OpenMP is targeting shared memory, it can use threads, which are created by the "master" thread and forked to run on different processors, thereby running certain portions of the code in parallel. By default, each thread executes its section of code independently; therefore, you can create "work sharing" by dividing a task among threads so that each thread can run its portion of the code. In this way, you can create both task and data parallelism.
OpenMP uses several directives:

- `PARALLEL`
- `DO`/`PARALLEL DO` and `SECTIONS` (primarily for work sharing)
- `SHARED` and `PRIVATE` clauses for sharing data (or not) between threads
- `CRITICAL`, `ATOMIC`, and `BARRIER` directories that coordinate and synchronize threads
- Run-time functions and environment variables (not directives, but functions that OpenMP makes available).

The form of the directives is very similar to OpenACC. For C, the directives look like this:

```
#pragma omp construct [clause [clause] ...]
```

For free-format Fortran, the directives look like:

```
!$omp construct [clause [, clause] ...]
```

A simple Fortran example of the `PARALLEL` directive is shown in Listing 2. The `PARALLEL` directives tell the compiler to create code so that the `write(*,*)` statement is executed by each thread. You control the number of threads with an environment variable, `OMP_NUM_THREADS`, that you set to the number of threads you want.
OpenMP also has a concept of work sharing using the `DO` directive, which specifies that iterations of the loop immediately following the directive must be executed in parallel. This assumes that a parallel region has been initiated with the `PARALLEL` directive. The `DO` construct

can get a little complicated, but the simple example in Listing 3 adds two vectors together and stores them in a third vector.

The OpenMP portion of the code creates a parallel region and indicates that the variables `A`, `B`, `C`, and `chunk` are shared between the threads but the loop variable `i` is specific to each thread (i.e., each thread has its own copy). Next is the `DO` directive that tells the compiler that the iteration of the loop will be distributed dynamically in `chunk`-sized sections.

As with OpenACC, you can find many tutorials and examples on the web, including YouTube. I encourage you take a look at them, because the reward is the possibility of greatly reducing the run time of your application.

Parting Comments

It's very interesting that 30% of the jobs at XSEDE, an organization that supplies HPC to university researchers, would only use a single core – probably because the researchers have never been taught the concepts of parallel programming or how easy it is to achieve. I have to admit that I'm guilty of writing single-thread (serial) code when I need something quick and dirty. However, if the code takes a fairly long time to run or if I have to run it many times, I will reach for compiler directives to parallelize the code easily.

At the same time, in an effort to get more speed from HPC systems, accelerators (co-processors) are becoming more and more prevalent. GPUs, DSPs, multicore (Phi), and FPGAs are all being used to improve performance. Coding for these isn't always easy; the directive-based approach called OpenACC allows you to code for accelerators more easily. Right now, OpenACC is focused on GPUs, but the promise is there for it to abstract other accelerators. A rule of thumb that you can use for the moment is that, if you want to parallelize the code on a single system that has multiple cores, then OpenMP is your likely tool. If you want to use GPUs, then OpenACC is your friend. You can combine both of them in the same code if you desire.

The GNU set of compilers [16] have had OpenMP capability for some time. In version 5.1 of the compilers, OpenACC support had been added as well, although I'm not sure whether all of OpenACC 2.0 is supported. Now you have the opportunity to try directive programming for multicore processors, for GPUs, or for both.

One last comment: I'm sure people are reading this and thinking, "Why hasn't OpenACC and OpenMP merged?" My answer is that I don't know, but I do know the two organizations talk to each other. Maybe someday they will create one set of directives. ■

Info

[1] Contact (movie): [http://www.imdb.com/title/tt0118884/?ref_=nv_sr_1]

[2] XSEDE: [https://www.xsede.org/]

[3] XSEDE resources overview: [https://www.xsede.org/resources/overview]

[4] XSEDE conference: [https://conferences.xsede.org/]

[5] XSEDE15 plenary panel: [http://www.hpcwire.com/2015/07/31/xsede-panel-highlights-diversity-of-nsf-computing-resources/]

[6] Comet: [http://cdn.opensfs.org/wp-content/uploads/2015/04/SDSC-Data-Oasis-GEn-II_Wagner.pdf]

[7] OpenMP: [https://en.wikipedia.org/wiki/OpenMP]

[8] MPI: [https://en.wikipedia.org/wiki/Message_Passing_Interface]

[9] OpenACC: [https://en.wikipedia.org/wiki/OpenACC]

[10] Parallel programming standard: [http://www.openacc.org/node/93]

[11] OpenACC v2.0: [http://www.openacc.org/sites/default/files/OpenACC%202%200.pdf]

[12] OpenACC v2.0 preview: [http://www.openacc.org/node/173]

[13] OpenACC YouTube videos: [https://www.youtube.com/results?search_query=OpenACC]

[14] OpenMP specifications: [http://openmp.org/specifications/]

[15] OpenMP for Accelerators: [https://cterboven.files.wordpress.com/2012/09/omp4-openmp_for_accelerators.pdf]

[16] GCC: [http://www.gnu.org/software/gcc/]

The Author

Jeff Layton has been in the HPC business for almost 25 years (starting when he was 4 years old). He can be found lounging around at a nearby Frys enjoying the coffee and waiting for sales.

Listing 2: Fortran PARALLEL Directive

```
    program hello
    implicit none

!$OMP PARALLEL
    write(*,*)'hello world'
!$OMP END PARALLEL

    stop
    end
```

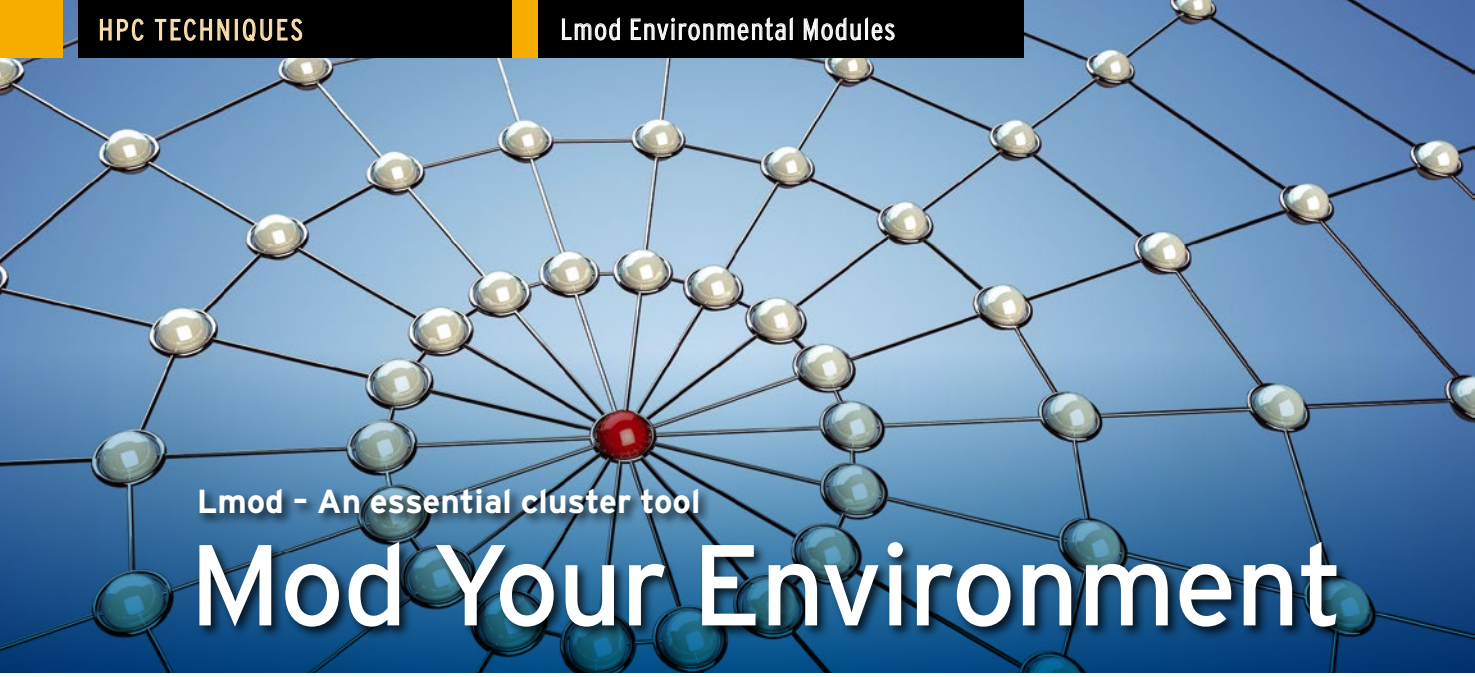
Listing 3: Simple DO Construct

```
    program Vec_Add
    integer n, chunksize, chunk, i
    parameter (n=1000)
    parameter (chunksize=100)
    real :: A(i), B(i), C(i)

! Some initializations
    do i = 1, n
       A(i) = i * 1.0
       B(i) = A(i)
    enddo

    chunk = chunksize
!$OMP PARALLEL SHARED(A,B,C,chunk) PRIVATE(i)
!$OMP DO SCHEDULE(DYNAMIC,chunk)
    do i = 1, n
       C(i) = A(i) + B(i)
    enddo
!$OMP END DO
!$OMP END PARALLEL

    end program
```



Lmod – An essential cluster tool

Mod Your Environment

Lmod is an indispensable tool for high-performance computing. It allows users to control their build and execute environment with ease. By Jeff Layton

One of the key tools for any cluster is environment modules, which allow you to define your user environment and the set of tools you need or want, to build and execute your application. The modules feed into a resource manager (job scheduler), where you can recreate the same environment to run the application that you used to build the application. One implementation of Environment Modules, Lmod [1], is under constant development and has some unique features that can be very useful in high-performance computing (HPC). It can even be useful on your own desktop if you write code and want to use a variety of tools and libraries. I use Lmod on my desktop and laptop to try new compilers or new versions of compilers or libraries.

Fundamentals

Programmers use a number of compilers, MPI libraries, compute libraries, and other tools to write applications. For example, one person might code with OpenACC [2], targeting GPUs, and Fortran, requiring the PGI compilers along with Open MPI, whereas another person might use the GNU 6.2 compilers with MPICH. The first user might use PETSc [3] to solve their problem; the second user might want to use OpenBLAS [4] for their code. Tools such as Lmod that allow users and developers to specify the exact set of tools they want or need is key to operating an effective HPC system. "Effective" can mean better performance (choosing the tools that allow your code to run as fast as possible), more flexibility (choice of tools that match the specific case), or ease of configuration of the environment for specific tools.

For example, assume you have three versions of the GNU compiler, 4.8, 5.4, and 6.2; the latest Intel and PGI compilers; and MPICH 3.3 and Open MPI 2.1. Furthermore, let's assume the users need two versions of PETSc and two versions of OpenBLAS. Altogether you have 40 possible combinations (five compilers, two MPI libraries, two PETSc versions, and two OpenBLAS versions). This is a very large number of combinations, and making it easy for users to use whatever combination they want is extremely difficult and very time consuming. At this point you have two choices: You can reduce the number of combinations (e.g., get rid of some of the compilers and perhaps one of the MPI libraries), or you can use Environment Modules so users can choose the combination of compiler, MPI library, and computational library they want or need. As a user, I might build a parallel application using GNU 4.8, the default compiler for CentOS 7.3, and MPICH; however, the GNU 6.2 compilers have some unique features that I might want use to try building the same application. Environment modules allow the user to select the tools used for production while allowing the use of a different toolset for development.

The secret to environment modules is manipulating the environment variables. Users can manipulate environment variables such as `$PATH`, `$LD_LIBRARY_PATH`, and `$MANPATH` and make changes to these variables according to the tool combinations desired. Changing the toolset changes these environment modules accordingly. It's fairly simple conceptually, but it's not always easy in practice.

Lmod

Lmod is an environment modules system that provides simple commands for manipulating your environment for the selection of tools and libraries. A set of "modules" is written to modify `$PATH`, `$LD_LIBRARY_PATH`, and `$MANPATH` and create needed environment vari-

ables for the specific tool or library that is needed. These modules also define dependencies, so you don't assemble conflicting tools. By "loading" or "unloading" these modules, you can change your environment to use what you need. If you "purge" all of your modules, they are all unloaded, and `$PATH`, `$LD_LIBRARY_PATH`, and `$MANPATH` are returned to the values present when you logged into the system. Lmod provides a complete set of tools for using and manipulating these modules. For example, you can list available modules, load and unload modules, purge all modules, swap modules, list loaded modules, query modules, ask for help on modules, show modules, and perform many other related tasks. Other options aren't used as frequently but are there if you need them.

One of the coolest features of Lmod is its ability to handle a module hierarchy, so that Lmod will only display modules that are dependent on loaded modules, preventing you from loading incompatible modules. This feature can help reduce unusual errors with mismatched modules that are sometimes very difficult to diagnose. I'll explain more about module hierarchy in a later section, because it is a very important feature in Lmod.

One of the first widely used environment module tools is Environment Modules Tcl/C [5], so-called because the code is written primarily in C and the modules in Tcl [6]. Lmod is a different implementation of environment modules and retains the ability to read and use modules written in Tcl. It adds the ability to read and use modules written in Lua [7], a popular language in its own right and a very embeddable language for applications.

Module files are not that difficult to create. In the case of Lua, there are a few functions that you use to manipulate `$PATH`, `$LD_LIBRARY_PATH`, and `$MANPATH` for the targeted application or library. Other environment variables can also be easily created. The module contains the dependencies so Lmod can track that information. You can also include help information in the module. You place these files in a directory hierarchy and add a couple of commands in the module so that Lmod knows the tool dependencies.

In the next section, Lmod Hierarchical Modules are discussed and explained, focusing on how to organize module files and how to limit the visibility of dependent module files. I'll use an example from my own laptop (a Lenovo G50-45) to help illustrate this process. Additionally, I've tried to add comments about Lmod best practices, some of which I've gathered from email discussions with the developer of Lmod, Dr. Robert McLay, on the Lmod users mailing list and others from Lmod documentation and presentations. I hope these help with your Lmod deployment.

Lmod Hierarchical Modules

One of the key capabilities of Lmod is module hierarchy. With Tcl/C Environment Modules, you can load pretty much any modules you want, even if they are not compatible, whereas Lmod doesn't allow you to see all possible modules, so you might not be aware of what is available might conflict with your modules. However, the Lmod `module spider` command lets you see all possible modules. Figure 1 illustrates the module hierarchy of the module files for my laptop. Anything marked with "(f)" is a file. Everything else is a directory.

At the top of the diagram, `/usr/local/modulefiles` is the Lmod default directory where all module files are stored. For single systems, this is fine. For clusters, the directory `/usr/local` would be NFS-exported to all of the compute nodes, or you could have installed Lmod to another NFS-exported directory. Below the root directory are three main subdirectories: `Core`, `compiler`, and `mpi`. These directories indicate the *dependencies* of the various modules. For example, everything in the `compiler` directory depends on a specific compiler (e.g., GCC 6.2). Everything in the `mpi` directory is dependent on a specific MPI and compiler combination. Everything in the `Core` directory does not depend on anything but the OS.

By default, Lmod reads module files in `/usr/local/modulefiles/Core`, so a best practice is to put any module files in this directory that do not depend on either a compiler or an MPI. This means you also put the compiler module files in the `Core` directory.

The `gcc` subdirectory under `Core` is where all of the module files for the GNU family of compilers are stored. A best practice from the developer of Lmod, Dr. Robert McLay [8] at the University of Texas Advanced Computing Center (TACC), is to make all subdirectories beneath `Core`, `compiler`, and `mpi` lowercase. In McLay's own words, "Lmod is designed to be easy to use interactively and be easy to type. So I like lowercase names wherever possible." He continues: "I know of some sites that try very hard to match the case of the software: Open MPI, PETSc, etc. All I can say is that I'm glad I don't have to work on those systems."

In the `gcc` subdirectory is a module file named "6.2." You will have a module file corresponding to every GCC compiler version. If you have versions 5.1, 6.2, and 7.0, then you should have three module files in

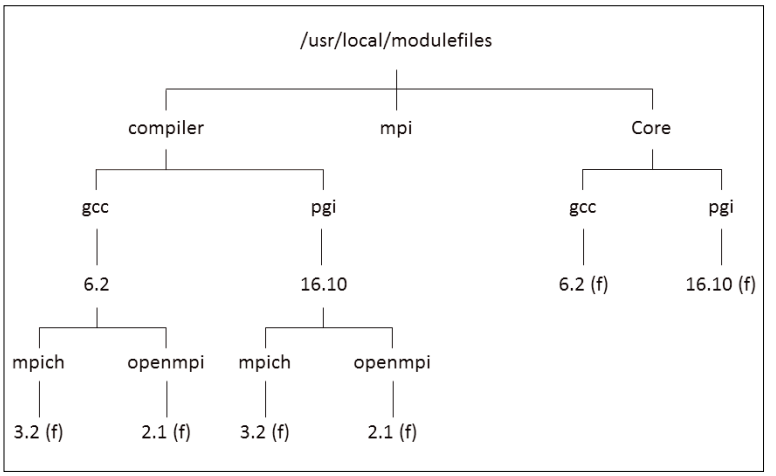


Figure 1: Example module file layout.

`/usr/local/modulefiles/Core/gcc` corresponding to these versions. In the case of GCC version 6.2, as shown in Figure 1, a module file named `6.2.lua` is labeled *6.2 (f)*, with the *(f)* indicating that it is a file and not a directory. This file contains the details for version 6.2 of the GNU compilers. The extension `.lua`, although not shown in Figure 1, indicates that the module file is written in Lua; however, it could be written in Tcl.

Notice that for a different set of compilers (e.g., those from PGI [9]), you would create a directory under `/use/local/modulefiles/Core` (i.e., `/usr/local/modulefiles/Core/pgi`) and then place the module files corresponding to the specific versions in this subdirectory.

All Lmod commands start with `module` followed by options. For example, you can find what modules are available using the `avail` option with the `module` command (Listing 1). Notice that compiler modules are "available" to be loaded. After loading a module, I `list` the modules loaded:

```
$ module load gcc/6.2
$ module list

Currently Loaded Modules:
  1) gcc/6.2
```

In this case, the gcc/6.2 compiler. The `compiler` module files modify the Lmod environment variables to point to the appropriate compiler and use commands to tell Lmod what MPI libraries are available that have been built with the loaded compiler. Therefore, only the MPI tools that depend on the loaded compiler are available to the user.

If you now type `module avail`, you get the response shown in Listing 2. Notice the two subdirectories under `/usr/local/modulefiles/compiler`, one for each compiler "family." Under the `gcc` compiler family is another subdirectory for each version of the GCC compiler that has modules. In this case, it's only version 6.2.

Under that subdirectory are all of the modules for applications and/or libraries that are dependent on the GCC 6.2 compiler. In Figure 1 there are two libraries that are dependent upon the gcc/6.2 module and are listed as subdirectories, `mpich` and `openmpi`. Under these directories are the *module files* corresponding to the specific MPI library version. These modules are denoted with an *(f)* next to their name in Figure 1.

Loading the MPICH 3.2 module should modify the `$PATH`, `$LD_LIBRARY_PATH`, and `$MANPATH` environment variables, as well as add some environment variables specific to MPICH. You can check by looking at the paths to the `mpicc` and `mpif77` scripts:

```
$ module load mpich/3.2
$ module list

Currently Loaded Modules:
  1) gcc/6.2   2) mpich/3.2

$ which mpicc
~/bin/mpich-3.2/bin/mpicc
$ which mpif77
~/bin/mpich-3.2/bin/mpif77
```

Notice that `mpicc` and `mpif77` point to the correct scripts (you can tell by the path).

An important key to making everything work correctly is in the module files. To better understand these module files, I'll take a deeper look at them.

Listing 1: `module avail` Before Loading

```
[laytonjb@laytonjb-Lenovo-G50-45 ~]$ module avail

------------------------ /usr/local/modulefiles/Core -------------------------
   gcc/6.2    pgi/16.10

-------------------- /usr/local/lmod/lmod/modulefiles/Core --------------------
   lmod/6.5   settarg/6.5

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 ..." to search for all possible modules
matching any of the "keys".
```

Under the Module File Hood

Lmod handles everything quite well so far. Modules can be loaded, unloaded, deleted, purged, and so on; however, Lmod executes whatever commands you put in the

module file, so you need to build good ones. To understand what's happening with these files, let's examine `gcc/6.2` compiler module that is a file underneath the "Core" branch of the module hierarchy as an example in Listing 3.

For this module, the GNU 6.2 compilers were installed in my home directory. Because this system is my laptop, I'm not too worried about where the compilers are installed. However, for clusters, I would install them in either `/usr/local/`, `/opt/`, or some directory that could be NFS-shared to all of the cluster nodes.

The module file can be broken down into several sections. The first part of the file is the `help` function, which is printed to `stdout` when you ask for help with the module. The next section defines the major environment variables `$PATH`, `$LD_LIBRARY_PATH`, and `$MANPATH`. Notice that the `prepend_path` function is used to put the compiler "first" in these environment variables.

The third major section is where the specific environment variables for the compiler are defined. For this module, the variables are pretty straightforward: `CC`, `cc`, `f90`, `F90`, and so on. These are specific to the compiler and are defined using the `pushenv` function, which pushes the variables into the environment. It also uses the `pathJoin` function, which helps creates the correct paths for these variables.

The last section under *Setup Modulepath for packages built by this compiler*, is very important and is key to Lmod. Two environment variables are defined for Lmod: `$MODULEPATH` and `$MODULEPATH_ROOT`.

The line

```
local mdir = pathJoin(?
  mroot,"compiler/gcc", version)
```

creates a local variable named `mdir`, which is a concatenation of the variable `mroot` (`$MODULEPATH_ROOT`) and `compiler/gcc` which is the branch of the `modulefile` hierarchy where the file is located. This tells Lmod that subsequent `module avail`

Listing 2: `module avail` After Loading

```
[laytonjb@laytonjb-Lenovo-G50-45 ~]$ module avail

------------------ /usr/local/modulefiles/compiler/gcc/6.2 -------------------
   mpich/3.2      openmpi/2.1

------------------------ /usr/local/modulefiles/Core -------------------------
   gcc/6.2 (L)    pgi/16.10

------------------- /usr/local/lmod/lmod/modulefiles/Core --------------------
   lmod/6.5       settarg/6.5

  Where:
   L:  Module is loaded

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 ..." to search for all possible modules
matching any of the "keys".
```

Listing 3: `gcc/6.2` Module

```
-- -*- lua -*-
-----------------------------------------------------------------------
-- GNU 6.2 compilers - gcc, g++, and gfortran. (Version 6.2.0)
-----------------------------------------------------------------------

help(
[[
This module loads the gcc-6.2.0 compilers (6.2.0). The
following additional environment variables are defined:

CC   (path to gcc compiler wrapper      )
CXX  (path to g++ compiler wrapper      )
F77  (path to gfortran compiler wrapper )
F90  (path to gfortran compiler wrapper )

See the man pages for gcc, g++, gfortran (f77, f90). For
more detailed information on available compiler options and
command-line syntax.
]])

-- Local variables
local version = "6.2"
local base = "/home/laytonjb/bin/gcc-6.2.0/"

-- Whatis description
whatis("Description: GNU 6.2.0 compilers with OpenCoarrays")
whatis("URL: www.gnu.org")

-- Take care of $PATH, $LD_LIBRARY_PATH, $MANPATH
prepend_path("PATH", pathJoin(base,"bin"))
prepend_path("PATH", pathJoin(base,"sbin"))
prepend_path("PATH", pathJoin(base,"include"))
prepend_path("LD_LIBRARY_PATH", pathJoin(base,"lib"))
prepend_path("LD_LIBRARY_PATH", pathJoin(base,"lib64"))
prepend_path("MANPATH", pathJoin(base,"share/man"))

-- Environment Variables
pushenv("CC", pathJoin(base,"bin","gcc"))
pushenv("CXX", pathJoin(base,"bin","g++"))
pushenv("F90", pathJoin(base,"bin","gfortran"))
pushenv("F77", pathJoin(base,"bin","gfortran"))
pushenv("FORT", pathJoin(base,"bin","gfortran"))
pushenv("cc", pathJoin(base,"bin","gcc"))
pushenv("cxx", pathJoin(base,"bin","g++"))
pushenv("f90", pathJoin(base,"bin","gfortran"))
pushenv("f77", pathJoin(base,"bin","gfortran"))
pushenv("fort", pathJoin(base,"bin","gfortran"))

-- Setup Modulepath for packages built by this compiler
local mroot = os.getenv("MODULEPATH_ROOT")
local mdir = pathJoin(mroot,"compiler/gcc", version)
prepend_path("MODULEPATH", mdir)

-- Set family for this module
family("compiler")
```

commands should look at the `compiler/gcc` subdirectory under the `compiler` subdirectory on the main `modulefiles` tree (the left hand branch). In particular, Lmod will only display modules in the `/usr/local/modulefiles/compiler/gcc/6.2` path. As the writer of the modules, you control where the module files that depend on the compilers are located. This step is the key to module hierarchy. You can control what modules are subsequently available by manipulating the `mdir` variable. This key attribute of Lmod gives you great flexibility.

The very last line in the module file, the statement `family("compiler")`, although optional, makes things easier for users (i.e., a best practice). The function `family` tells Lmod to which family the module belongs. A user can only have one module per family loaded at a time. In this case, the family is `compiler`, so that means no other compilers can be loaded. Adding this line helps users prevent self-inflicted problems from loading a module that is incompatible with the loaded compiler. Even though the statement is somewhat optional, I highly recommend using it.

If the `gcc 6.2` compiler module is loaded, then the diagram of the module layout should look something like Figure 2. The green labels indicate the module path resulting from the currently loaded module (the `gcc 6.2` compiler module). The red labels indicate the path to the modules that depend on the loaded compiler module – in this case, the MPI modules, `mpich/3.2` and `openmpi/2.1`. Note that the MPI modules are under the `compiler` directory because they depend on the `compiler` module that is loaded.

In the previous section, I loaded the `mpich-3.2` module (Listing 4) that was built with the `gcc/6.2` compilers. If you compare this module file to the compiler module file, you will see many similarities. The classic environment variables, `$PATH`, `$LD_LIBRARY_PATH`, and `$MANPATH`, are modified, and certain environment variables are defined. Because you want the MPI tools associated with the module to be "first" in `$PATH`, the `prepend_path` Lmod module command is again used.

Toward the end of the file, examine the code for the module path. The local variable `mdir` points to the "new" module subdirectory, which is `mpi/gcc/6.2/mpich/3.2`. (Technically, the full path is `/usr/local/modulefiles/mpi/gcc/6.2/mpich/3.2` because `$MODULEPATH_ROOT` is `/usr/local/modulefiles`.) In this subdirectory, you should place all modules that point to tools that have been built with both the `gcc/6.2` compilers and the `mpich/3.2` tools. Examples of module files that depend on both a compiler and an

Listing 4: `mpich-3.2` Module

```
-- -*- lua -*-
-----------------------------------------------------------------------
-- mpich-3.2 (3.2) support. Built with gcc-6.2 (6.2.0)
-----------------------------------------------------------------------

help(
[[
This module loads the mpich-3.2 MPI library built with gcc-6.2.
compilers (6.2.0). It updates the PATH, LD_LIBRARY_PATH,
and MANPATH environment variables to access the tools for
building MPI applications using MPICH, libraries, and
available man pages, respectively.

This was built using the GNU compilers, version 6.2.0.

The following additional environment variables are also defined:

MPICC   (path to mpicc compiler wrapper   )
MPICXX  (path to mpicxx compiler wrapper  )
MPIF77  (path to mpif77 compiler wrapper  )
MPIF90  (path to mpif90 compiler wrapper  )
MPIFORT (path to mpifort compiler wrapper )

See the man pages for mpicc, mpicxx, mpif77, and mpif90. For
more detailed information on available compiler options and
command-line syntax. Also see the man pages for mpirun or
mpiexec on executing MPI applications.
]])

-- Local variables
local version = "3.2"
local base = "/home/laytonjb/bin/mpich-3.2"

-- Whatis description
whatis("Description: MPICH-3.2 with GNU 6.2 compilers")
whatis("URL: www.mpich.org")

-- Take care of $PATH, $LD_LIBRARY_PATH, $MANPATH
prepend_path("PATH", pathJoin(base,"bin"))
prepend_path("PATH", pathJoin(base,"include"))
prepend_path("LD_LIBRARY_PATH", pathJoin(base,"lib"))
prepend_path("MANPATH", pathJoin(base,"share/man"))

-- Environment Variables
pushenv("MPICC", pathJoin(base,"bin","mpicc"))
pushenv("MPICXX", pathJoin(base,"bin","mpic++"))
pushenv("MPIF90", pathJoin(base,"bin","mpif90"))
pushenv("MPIF77", pathJoin(base,"bin","mpif77"))
pushenv("MPIFORT", pathJoin(base,"bin","mpifort"))
pushenv("mpicc", pathJoin(base,"bin","mpicc"))
pushenv("mpicxx", pathJoin(base,"bin","mpic++"))
pushenv("mpif90", pathJoin(base,"bin","mpif90"))
pushenv("mpif77", pathJoin(base,"bin","mpif77"))
pushenv("mpifort", pathJoin(base,"bin","mpifort"))

-- Setup Modulepath for packages built by this compiler/mpi
local mroot = os.getenv("MODULEPATH_ROOT")
local mdir = pathJoin(mroot,"mpi/gcc", "6.2","mpich","3.2")
prepend_path("MODULEPATH", mdir)

-- Set family for this module (mpi)
family("mpi")
```

MPI tool are applications or libraries such as PETSc.
Also notice that the `mpich/3.2` module also uses the `family()` function, so the user cannot load a second MPI module. You could even have a `family()` function for libraries such as PETSc.

Module Usage for the Admin

In an article from a couple of years ago [10], I presented a way to gather logs about Tcl/C environment module usage. It was a bit of a kludge, but it did allow me to gather the data I needed. With Lmod, this ability was brought to the forefront.
Tracking module usage is conceptually fairly easy, but a number of steps are involved. Having this information can be amazingly important, because it allows you to track which tools are used the most. (I associate one tool with one module.) If you have various versions of a specific tool, it allows you to track the usage of each so that you can either deprecate an older version or justify keeping it around and maintaining it. You can also see which modules are used as a function of time, which helps you understand when people run their jobs and what modules they use.

Summary

Although I've written about Lmod before, I continue to come back to it because it is so useful. It greatly helps users sort out their environment so that they don't accidentally load conflicting libraries and tools. The first time you have to debug a user's code when they have mixed MPI implementations, you will be thankful for Lmod. Environment modules in general, and Lmod specifically, allow you to keep multiple versions of the same package on a system to service applications that have been built with older versions of a compiler, MPI, or library, or even old libraries that are needed. I even saw a posting to the Open MPI mailing list [11] asking about LAM-MPI, even though it basically has been dead for a decade. You would be surprised how long applications stick around and bring their dependencies with them. Because Lmod can read Tcl module files in addition to Lua (the preferred language), you can move easily from Tcl/C Environment Modules to Lmod. As you can see from the Lua module file examples here, the syntax is very clean and simple, making them very easy to read. Finally, Lmod is developing tools to audit module usage. This information is amazingly useful, as pointed out in two articles from Harvard [12] [13]. The author gives a very good explanation of how to set up Lmod to collect module usage data, put it into a database, and mine that database – which is very cool stuff indeed. ■

Info

[1] Lmod: [https://www.tacc.utexas.edu/research-development/tacc-projects/lmod]
[2] OpenACC: [http://www.openacc-standard.org/]
[3] PETSc: [http://www.mcs.anl.gov/petsc/]
[4] OpenBLAS: [http://www.openblas.net/]
[5] Environment Modules: [http://modules.sourceforge.net/]
[6] Tcl: [https://www.tcl.tk/]
[7] Lua: [https://www.lua.org/]
[8] Dr. Robert McLay: [https://www.tacc.utexas.edu/about/directory/robert-mclay]
[9] PGI: [http://www.pgroup.com/index.htm]
[10] "Gathering Data on Environment Modules" by Jeff Layton: [http://www.admin-magazine.com/HPC/Articles/Gathering-Data-on-Environment-Modules]
[11] "Looking for LAM-MPI Sources to Create a Mirror," Open MPI User's Mailing List Archives: [http://www.open-mpi.org/community/lists/users/2015/06/27079.php]
[12] "Scientific Software as a Service Sprawl," part 1, by James Cuff: [http://blog.jcuff.net/2012/07/scientific-software-as-service-sprawl.html]
[13] "Scientific Software as a Service Sprawl," part 2, by James Cuff: [http://blog.jcuff.net/2012/07/part-two-scientific-software-as-service.html]

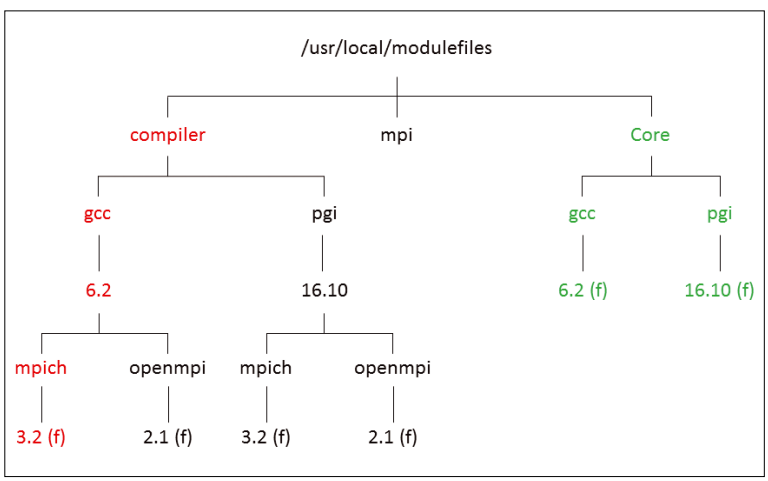
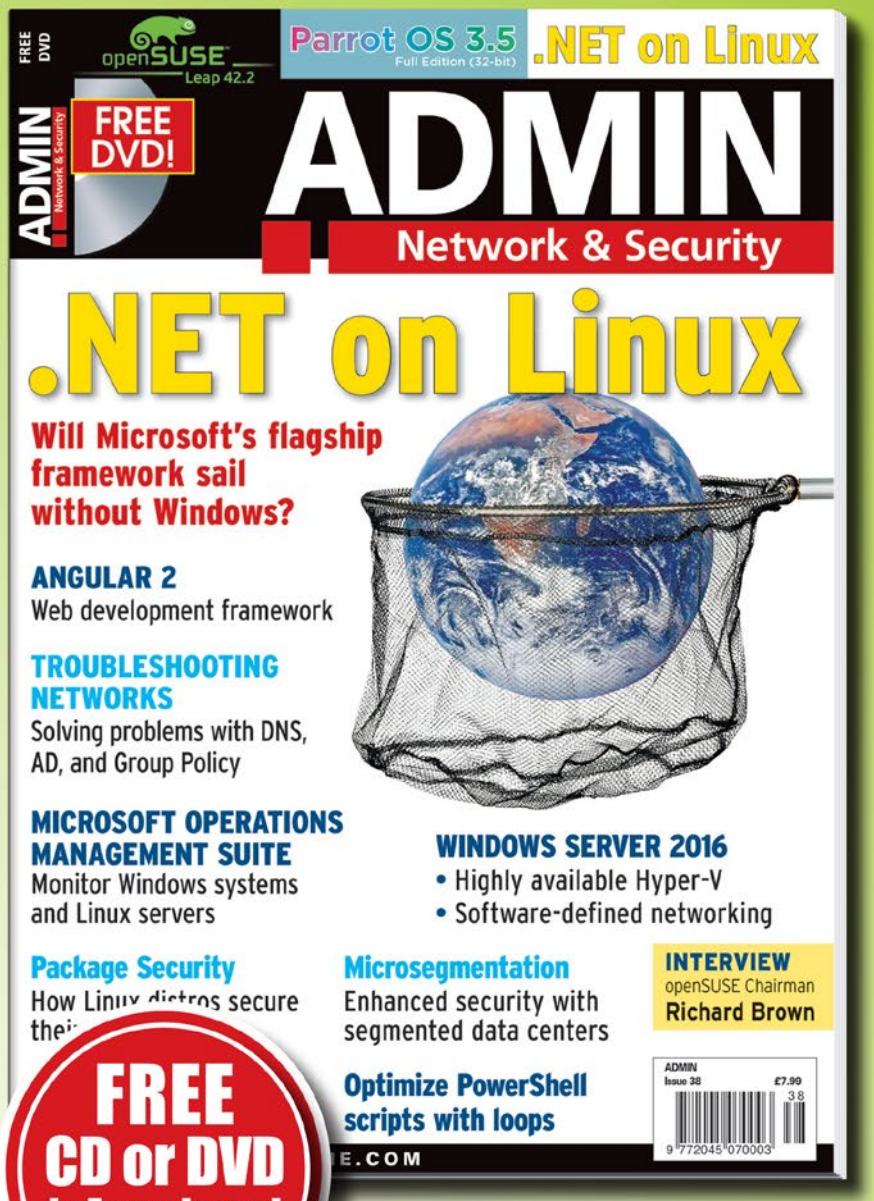


Figure 2: Active path after `gcc/6.2` is loaded.

REAL SOLUTIONS FOR REAL NETWORKS



FREE CD or DVD in Every Issue!

Each issue delivers technical solutions to the real-world problems you face every day.

Learn the latest techniques for better:

- network security
- system management
- troubleshooting
- performance tuning
- virtualization
- cloud computing

on Windows, Linux, Solaris, and popular varieties of Unix.

6 issues per year!

ORDER ONLINE AT: shop.linuxnewmedia.com