MPI and Password Cracking

Author: Jason R. Davis
Site: TMTO[dot]ORG
Date: May 28th, 2009

Table of Contents:

Introduction

After sauntering through the web for many hours searching for GOOD documentation on this topic, I realized that it simply does not exist. I've been using MPI, both OpenMPI and MPICH2, for a little over two years and have become familiar with implementing it effectively in production environments. I also have experience with setting up scalable password cracking utilities that operate through MPI. With that said, I couldn't sit on this knowledge anymore, and went through the laborious task of documenting it for public release. I will be updating this documentation in the future to include instructions on how to scale this to a full cluster. For now, it'll let you utilize all the cores on a single PC for cracking, instead of just one core.

The goals:

- Setup a cluster like directory structure that is scalable
- Install OpenMPI
- Install John the Ripper MPI
- Run a simple test and crack a MD5 hash

Directory Structure Methodology

This is almost the most important part of building a system that is scalable and won't break if you upgrade to a newer version of an application. I am going to quickly outline how I build a directory structure for a single machine. That structure is then scalable to a cluster.

- /apps/
  - /apps/myapp1/
    - /apps/myapp1/v1.x/
      - /apps/myapp1/v1.x/install/
      - /apps/myapp1/v1.x/x86_32/
      - /apps/myapp1/v1.x/x86_64/
    - /apps/myapp1/v2.x/
      - /apps/myapp1/v2.x/install/
      - /apps/myapp1/v2.x/x86_32/
      - /apps/myapp1/v2.x/x86_64/
    - /apps/myapp1/latest → v2.x/
    - /apps/myapp1/stable → v1.x/
    - /apps/myapp1/env/

/apps/
This is the base directory for all our cluster applications.

/apps/myapp1/
For each application we create a subdirectory of our base directory, in this case "myapp1"

/apps/myapp1/v1.x/
The **stable** version of the application will go below this directory structure. This is where the production application will reside. You will use the actual version number of the application for the directory name. You will not use "v1.x".

/apps/myapp1/v1.x/install/
The source code for this specific version will be downloaded, unpacked, and compiled from this directory.

/apps/myapp1/v1.x/x86_32/
If the application was compiled on a 32-bit processor, then this is where the application will be installed and ran from.

/apps/myapp1/v1.x/x86_64/
If the application was compiled on a 64-bit processor, then this is where the application will be installed and ran from.

/apps/myapp1/v2.x/
The **latest** version of the application will go below this directory structure. This is where the alpha, beta, or unstable release will reside. This includes applications that need to be tested and are not ready for production. You will use the actual version number of the application for the directory name. You will not use “v2.x”.

/apps/myapp1/v2.x/install/
The source code for this specific version will be downloaded, unpacked, and compiled from this directory.

/apps/myapp1/v2.x/x86_32/
If the application was compiled on a 32-bit processor, then this is where the application will be installed and ran from.

/apps/myapp1/v2.x/x86_64/
If the application was compiled on a 64-bit processor, then this is where the application will be installed and ran from.

/apps/myapp1/latest → v2.x
This is a symbolic link “latest” pointing to the directory named after the unstable installed version.

/apps/myapp1/stable → v1.x
This is a symbolic link “stable” pointing to the directory named after the stable installed version

These symbolic links will allow you to upgrade software and not need to rebuild the environment to accommodate the change. Simply modify the symbolic link to point to the newer version, still under the name of latest. If I have “/apps/myapp1/v1.x/bin/” in the $PATH variable for all users .bashrc, then I would have to update all the user .bashrc files if I wanted to upgrade everyone to “/apps/myapp1/v2.x/bin/”. By using the “stable” and “latest” symbolic links, you only need to put “/apps/myapp1/stable/bin/” in each users .bashrc file. And when you want to migrate all users to “/apps/myapp1/v2.x/bin/” simply modify the “stable” symbolic link to point to it. All the users will automatically start using the newest version on a simple change to the symbolic links. This is very important for scalability and allows you to test new releases without affecting the other users by putting “/apps/myapp1/latest/bin/” into your $PATH.

/apps/myapp1/env/
This is where scripts and files that modify the user environment will be placed

Hardware and Software

For this process I've chosen Gentoo as the GNU/Linux distribution. I've installed it onto a server containing a single "Intel(R) Xeon(R) E5405 @ 2.00GHz" which happens to be a Quad Core processor. Which is why I use "-np 4" in the mpirun commands. Also note, I am using a 64-bit OS and software. Pay attention to your version and make the minor changes to the commands to accommodate that.

Installing OpenMPI

OpenMPI v1.3.2 (Latest) / OpenMPI v1.3.1 (Stable)

```
~ # mkdir /apps
~ # mkdir /apps/openmpi
```

Install v1.3.2:

```
~ # mkdir /apps/openmpi/v1.3.2
~ # ln -s /apps/openmpi/v1.3.2 /apps/openmpi/latest
~ # mkdir /apps/openmpi/v1.3.2/install
~ # mkdir /apps/openmpi/v1.3.2/x86_32
~ # mkdir /apps/openmpi/v1.3.2/x86_64
~ # cd /apps/openmpi/v1.3.2/install
~ # wget http://www.open-mpi.org/software/ompi/v1.3/downloads/openmpi-1.3.2.tar.gz
~ # tar -xf openmpi-1.3.2.tar.gz
~ # cd openmpi-1.3.2
```

For 32-bit systems:

```
~ # ./configure --prefix= /apps/openmpi/v1.3.2/x86_32
```

For 64-bit systems:

```
~ # ./configure --prefix=/apps/openmpi/v1.3.2/x86_64
```

```
~ # make
~ # make install
```

Install v1.3.1:

```
~ # mkdir /apps/openmpi/v1.3.1
~ # ln -s /apps/openmpi/v1.3.1 /apps/openmpi/stable
~ # mkdir /apps/openmpi/v1.3.1/install
~ # mkdir /apps/openmpi/v1.3.1/x86_32
~ # mkdir /apps/openmpi/v1.3.1/x86_64
~ # cd /apps/openmpi/v1.3.1/install
~ # wget http://www.open-mpi.org/software/ompi/v1.3/downloads/openmpi-1.3.1.tar.gz
```

~ # tar -xf openmpi-1.3.1.tar.gz
~ # cd openmpi-1.3.1

For 32-bit systems:
~ # ./configure --prefix= /apps/openmpi/v1.3.1/x86_32

For 64-bit systems:
~ # ./configure --prefix=/apps/openmpi/v1.3.1/x86_64

~ # make
~ # make install

~ # mkdir /apps/openmpi/env
~ # cd /apps/openmpi/env
~ # echo 'export PATH=${PATH}:/apps/openmpi/latest/x86_64/bin' > prepenv_latest
~ # echo 'export PATH=${PATH}:/apps/openmpi/stable/x86_64/bin' > prepenv_stable

Edit /etc/env.d/00basic and modify the LDPATH to include the OpenMPI libraries:

LDPATH="/usr/local/lib:/apps/openmpi/latest/x86_64/lib:/apps/openmpi/stable/x86_64/lib"

~ # env-update && source /etc/profile

Running a OpenMPI Sanity Check:

~ # source /apps/openmpi/env/prepenv_latest
~ # mpirun -np 4 hostname

The last command above will print the computers hostname four times. Not less, not more, four times. If it does, then your "latest" OpenMPI version is working.

~ # env-update && source /etc/profile
~ # source /apps/openmpi/env/prepenv_stable
~ # mpirun -np 4 hostname

The last command above will print the computers hostname four times. Not less, not more, four times. If it does, then your "stable" OpenMPI version is working.

At this point you should have a functioning OpenMPI install. Two versions, latest and stable, are available based upon which file you source. The reason I force users to login and "source" the "prepenv_xxxxxx" file is because I don't want to add bloat to user profiles. I despise $PATH variables that are 30+ directories long, etc. Less is more in this case. I don't add any information to users .bashrc in this case. Some users actually put the "source

/apps/openmpi/env/prepenv_stable" command in their .bashrc, which makes OpenMPI available at the time they login by setting the $PATH.

Installing John the Ripper MPI

Note: You cannot install John the Ripper MPI unless you have a functioning MPI environment. So do not continue unless you have successfully installed OpenMPI and completed the sanity check.

John the Ripper MPI v1.7.3.1 (Latest)

```
~ # mkdir /apps/jtr
```

Install v1.7.3.1:

```
~ # mkdir /apps/jtr/v1.7.3.1
~ # ln -s /apps/jtr/v1.7.3.1 /apps/jtr/latest
~ # mkdir /apps/jtr/v1.7.3.1/install
~ # mkdir /apps/jtr/v1.7.3.1/x86_32
~ # mkdir /apps/jtr/v1.7.3.1/x86_64
~ # cd /apps/jtr/v1.7.3.1/install
~ # wget http://www.bindshell.net/tools/johntheripper/john-1.7.3.1-all-2-mpi8.tar.gz
~ # tar -xf john-1.7.3.1-all-2-mpi8.tar.gz
~ # cd john-1.7.3.1-all-2-mpi8/src
```

For 32-bit systems:

```
~ # make linux-x86-sse2
```

For 64-bit systems:

```
~ # make linux-x86-64
```

```
~ # cd /apps/jtr/v1.7.3.1/install/john-1.7.3.1-all-2-mpi8/run
~ # cp * /apps/jtr/v1.7.3.1/x86_64/
```

```
~ # mkdir /apps/jtr/env
~ # cd /apps/jtr/env
~ # echo 'export PATH=${PATH}:/apps/jtr/latest/x86_64' > prepenv_latest
~ # echo 'source /apps/openmpi/env/prepenv_latest' >> prepenv_latest
~ # echo 'cd /apps/jtr/latest/x86_64' >> prepenv_latest
```

Running a John the Ripper Sanity Check:

```
~ # env-update && source /etc/profile
~ # source /apps/jtr/env/prepenv_latest
```

~ # mpirun -np 4 ./john -test

The last command above will self test John the Ripper MPI and run a benchmark to see how fast your computer can run it.

Utilizing CPU Cores on a Standalone PC

At this point you should have a functioning MPI environment and a functioning version of John the Ripper. So now that the foundation is laid, let us move on to the fun part: Cracking Passwords!

The first task we need to complete is to create a cleanup script for John the Ripper, because it creates a lot of files that don't get cleaned up after its runs. Only run this if you no longer need the john_log and john.pot files, because it removes them as well.

~ # cd /apps/jtr/env
~ # echo "#!/bin/bash" > cleanup_latest.sh
~ # echo "rm /apps/jtr/latest/x86_64/john_rec.rec*" >> cleanup_latest.sh
~ # echo "rm /apps/jtr/latest/x86_64/john_log" >> cleanup_latest.sh
~ # echo "rm /apps/jtr/latest/x86_64/john.pot" >> cleanup_latest.sh
~ # chmod + x cleanup_latest.sh

The entire platform is setup now. Next we will begin a test run, with a simple MD5 hash.

~ # env-update && source /etc/profile
~ # source /apps/jtr/env/prepenv_latest
~ # echo "root:e80b5017098950fc58aad83c8c14978e" >> md5s.txt
~ # mpirun -np 4 ./john -format=raw-md5 -incremental:alpha ./md5s.txt
Loaded 1 password hash (Raw MD5 [raw-md5])
Loaded 1 password hash (Raw MD5 [raw-md5])
Loaded 1 password hash (Raw MD5 [raw-md5])
Loaded 1 password hash (Raw MD5 [raw-md5])
abcdef (root)
Process 0 completed loop.
thread: 0 guesses: 1 time: 0:00:00:00 c/s: 1582K trying: abaflm - abcdch

The last command above initiated John the Ripper MPI utilizing all 4 cores and successfully broke the password in under a second. The password was "abcdef". If you have questions about the syntax for running John the Ripper, please read the related docs.

After each run, I usually copy the "john.pot" file to safe spot, and then cleanup:

~ # /apps/jtr/env/cleanup_latest.sh

Outro

As mentioned in at the beginning of this document, I will be amending updates to show users how to expand to a cluster, spanning many PCs and many CPU cores. If you have trouble, utilize Google as much as possible; every error I came across had an answer and a quick Google search brought it right to me.

Jason R. Davis
TMTO[dot]ORG