Date Published: 07/07/2022

Student Cluster Compeititon - Tutorial 4

Table of Contents

Overview

Tutorial 4 demonstrates environment module manipulation and the compilation and optimisation of HPC benchmark software. This introduces the reader to the concepts of environment management and workspace sanity, as well as compilation of software on Linux.

In this tutorial you will:

- [ ] Install and configure Lmod on your head node.
- [ ] Download and compile the High Performance LINPACK (HPL) benchmark.
- [ ] Create Slurm batch scripts to submit jobs for your benchmark runs.
- [ ] Optimise HPL.
- [ ] Download and compile the High Performance Computing Challenge (HPCC) benchmark.

Date Published: 07/07/2022

Part 1 - Enviroment Modules with Lmod

Environment Modules provide a convenient way to dynamically change a user's environment through *modulefiles* to simplify software and library use when there are multiple versions of a particular software package (e.g. Python2.7 and Python 3.x) installed on the system. Environment Module parameters typically involve, among other things, modifying the PATH environment variable for locating a particular package (such as dynamically changing the path to Python from `/usr/local/bin/python2.7` to `/usr/local/bin/python3`).

Lmod is a Lua-based environment module tool for users to easily manipulate their HPC software environment and is used on thousands of HPC systems around the world.

Installing Lmod

To install Lmod on **CentOS 8**, make sure that you have the **EPEL** repository enabled:

```
[...@headnode ~]$ sudo dnf install epel-release
```

This will enable additional software to be made available to the CentOS package manager. With this done, install the Lmod packages:

```
[...@headnode ~]$ sudo dnf --enablerepo=powertools install Lmod
```

Now log out and back into your session.

Using Lmod

With Lmod installed, you'll now have some new commands on the terminal. Namely, these are: `module <subcommand>`. The important ones for you to know and use are: `module avail`, `module list`, `module load` and `module unload`. These commands do the following:

Command	Operation
`module avail`	Lists all modules that are available to the user.
`module list`	Lists all modules that are loaded by the user.
`module load <module_name>`	Loads a module to the user's environment.
`module unload <module_name>`	Removes a loaded module from the user's environment.

Lmod also features a shortcut command `ml` which can perform all of the above commands:

Command	Operation
`ml`	Same as `module list`
`ml avail`	Same as `module avail`
`ml <module_name>`	Same as `module load <module_name>`
`ml -<module_name>`	Same as `module unload <module_name>`

Date Published: 07/07/2022

Command	Operation
`ml foo`	Same as `module load foo`
`ml foo -bar`	Same as `module load foo` and `module unload bar`

Adding modules to Lmod

Some installed packages will automatically add environment modules to the Lmod system, while others will not and will require you to manually add definitions for them. For example, the `openmpi` package that we will install with `dnf` later in this tutorial will automatically add a module file to the system for loading via Lmod.

We will detail how to create a module file for your own software later.

Date Published: 07/07/2022

Part 2 - High Performance LINPACK (HPL) Benchmark

The High Performance LINPACK (HPL) benchmark is used to measure a system's floating point number processing power. The resulting score (in Floating Point Operations Per Second, or FLOPS for short) is often used to roughly quantify the computational power of an HPC system. HPL requires math libraries to perform its floating point operations as it does not include these by itself and it also requires an MPI installation for communication in order to execute in parallel across multiple CPU cores (and hosts).

System Libraries

A library is a collection of pre-compiled code that provides functionality to other software. This allows the re-use of common code (such as math operations) and simplifies software development. You get two types of libraries on Linux: `static` and `dynamic` libraries.

Static Libraries

Static libraries are embedded into the binary that you create when you compile your software. In essence, it copies the library that exists on your computer into the executable that gets created at **compilation time**. This means that the resulting program binary is self-contained and can operate on multiple systems without them needing the libraries installed first. Static libraries are normally files that end with the `.a` extension, for "archive".

Advantages here are that the program can potentially be faster, as it has direct access to the required libraries without having to query the operating system first, but disadavanges include the file size being larger and updating the library requires recompiling (and linking the updated library) the software.

Dynamic Libraries

Dynamic libraries are loaded into a compiled program at **runtime**, meaning that the library that the program needs is **not** embedded into the executable program binary at compilation time. Dynamic libraries are files that normally end with the `.so` extension, for "shared object".

Advantages here are that the file size can be much smaller and the application doesn't need to be recompiled (linked) when using a different version of the library (as long as there weren't fundamental changes in the library). However, it requires the library to be installed and made available to the program on the operating system.

HPL uses dynamic libraries for its math and MPI communication, as mentioned above.

Message Passing Interface (MPI)

MPI is a message-passing standard used for parallel software communication. It allows for software to send messages between multiple processes. These processes could be on the local computer (think multiple cores of a CPU or multiple CPUs) as well as on networked computers. MPI is a cornerstone of HPC. There are many implementations of MPI in software such as OpenMPI, MPICH, MVAPICH2 and so forth. To find out more about MPI, please read the following: https://www.linuxtoday.com/blog/mpi-in-thirty-minutes.html

Installing HPL

We need to install the dynamic libraries that HPL expects to have, as well as the software for MPI. The MPI implementation we're going to use here is OpenMPI and we will use the Automatically Tuned Linear Algebra Software (ATLAS) math library.

Remember, since it's dynamically linked, we need to ensure that ALL of our nodes that we expect to run the software on have the expected libraries.

1. Install OpenMPI and ATLAS and their libraries on all of your nodes:

Date Published: 07/07/2022

```
[...@headnode ~]$ sudo dnf install openmpi atlas openmpi-devel atlas-devel
```

```
[...@computenode ~]$ sudo dnf install openmpi atlas openmpi-devel atlas-devel
```

2. On the head node, in your NFS-shared home directory, download and extract the latest HPL release from the following URL:

```
http://www.netlib.org/benchmark/hpl/
```

3. Extract the downloaded file using `tar`.

4. Enter the extracted directory. In this directory there is a directory called `setup`, which contains templates for settings for compiling HPL with various computer hardware configurations. Pick one that is closest to your CPU architecture and copy it to the root HPL extracted directory. The `Make.Linux_PII_CBLAS_gm` file will do for your VMs.

```
[...@headnode ~]$ cd hpl-<version>
[...@headnode hpl-<version>]$ cp setup/Make.Linux_PII_CBLAS_gm .
```

NOTE: The Makefile suffix (`Linux_PII_CBLAS_gm` in this case) is used as the architecture identifier and is required to be specified when building the software later. The period `.` indicates that we are working in the current directory.

5. Edit the `Make.Linux_PII_CBLAS_gm` file and change the following line to represent where your extracted `hpl-<version>` directory is (you can use `pwd` while inside of the directory to get the full path):

```
TOPdir = </path/to/extracted/hpl/directory>
```

6. In `Make.Linux_PII_CBLAS_gm`, edit the following lines to specify where your dynamic libraries are for ATLAS:

```
LAdir        = /usr/lib64/atlas
LAlib        = $(LAdir)/libtatlas.so $(LAdir)/libsatlas.so
```

7. In `Make.Linux_PII_CBLAS_gm`, edit the following lines to specify which binaries you want to use for your C compiler and linker. We want to use the compiler included with OpenMPI, which is called `mpicc` (MPI C Compiler):

```
CC           = mpicc
LINKER       = mpicc
```

> **! >>> By default, the OpenMPI you installed above is not available in your environment ($PATH & $LD_LIBRARY_PATH). Make OpenMPI available BEFORE you do any compilation steps below. To load it, do a `module avail` and look for the `mpi` module and then `module load` that module.**

8. Compile HPL

Important tip: if your compile fails, you should reset to a clean start point with `make clean`.

```
[...@headnode ~]$ make arch=<arch> #<arch> in this case is Linux_PII_CBLAS_gm
```

Date Published: 07/07/2022

To confirm that the compilation completed successfully, check that the `xhpl` executable was produced in `<hpl_extracted_dir>/bin/<arch>/xhpl`.

9. The `HPL.dat` file (in the same directory as `xhpl`) defines how the HPL benchmark solves a large dense linear array of **double precision floating point numbers**. Therefore, selecting the appropriate parameters in this file can have a massive effect on the FLOPS you obtain.

```
     The most important parameters are:
N:   defines the length of one of the sides of the 2D array to be solved.
     Therefore, problem size ∝ runtime ∝ memory usage ∝ <N>².
     For best performance N should be a multiple of NB.
NB:  defines the block (or chunk) size into which the array is divided.
     The optimal value is determined by the CPU architecture such that the block fits in cache (Google).
P&Q: define the domains (in two dimensions) for how the array is partitioned on a distributed memory system.
     Therefore P*Q = MPI ranks.
```

   You can find online calculators that will generate an `HPL.dat` file for you **as a starting point**, but you will still need to do some tuning if you want to squeeze out maximum performance.

10. Execute HPL on your head node

```
[...@headnode ~]$ mpirun -np <cores> ./xhpl
```

11. Create a SLURM submission script to run your benchmark on your compute node. The script should look as follows:

```
#!/bin/bash
#SBATCH --ntasks <MPI_RANKS>
#SBATCH -N <NODES>
#SBATCH -t 02:00:00
#SBATCH --export=ALL
#SBATCH --job-name=hpl_benchmark

mpirun /path/to/xhpl
```

    SLURM will populate the appropriate MPI parameters based on the resources you requested, so specifying ranks (-np) is not required.

12. Make sure your MPI environment is loaded, and then submit your job to SLURM:

```
[...@headnode ~]$ sbatch <script>
```

13. Check the state of your job with

```
[...@headnode ~]$ squeue
```

    By default, SLURM will store the job output to a log file `slurm-<job_id>.out`, on the compute node (Slurm writes the output file on the first node that the job has been allocated to by default). Since your home directory is shared via NFS, you should be able to check the output on any of your nodes.

Adding a Second Compute Node

Date Published: 07/07/2022

At this point you are ready to run HPL on your cluster with two compute nodes. It's time to deploy a second compute node in OpenStack.

!!! Have the output **log file** `slurm-<job_id>.out` AND `Make.<architecture>` **configuration file** ready for instructors to view on request.

Pay careful attention to the hostname, network and other configuration settings that may be specific to and may conflict with your initial node. Once your two compute nodes have been successfully deployed, are accessible from the head node and added to SLURM, you can continue with running HPL across multiple nodes.

As a sanity check repeat Steps 10-12 of the previous task: Message Passing Interface (MPI), but this time do it for **two** compute nodes and check the new results of your benchmark.

Optimising HPL

You now have a functioning HPL benchmark and a compute cluster. However, using math libraries (BLAS, LAPACK, **ATLAS**) from a repository (`dnf`) **will not yield optimum performance**, because these repositories **contain generic code compiled to work on all x86 hardware**.

Code compiled specifically for HPC hardware can use instruction sets like **AVX**, **AVX2** and **AVX512** (if available) to make better use of the CPU. A (much) higher HPL result is possible if you compile your math library (such as ATLAS, GOTOBLAS, OpenBLAS or Intel MKL) from source code on the hardware you intend to run the code on.

The VMs that make up your cluster are not necessarily the same architecture, since they run on a variety of hardware in the ACE Lab. In order to compile high performance codes for your compute nodes, you need to perform the following steps on your compute nodes:

1. Download a math library's source code and compile it.
2. Recompile HPL using this new library implementation (edit `LAdir` in the `Makefile`). This has to be done for the target machines that you intend to run HPL on (think: NOT just the head node, since that just schedules the jobs to be run).
3. Re-run HPL on your cluster.

Theoretical Peak Performance

It is useful to know what the theoretical FLOPS performance (RPeak) of your hardware is when trying to obtain the highest benchmark result (RMax). RPeak can be derived from the formula:

```
RPeak = CPU Frequency [GHz] * Num CPU Cores * OPS/cycle
```

Newer CPU architectures allow for 'wider' instruction sets which execute multiple instructions per CPU cycle. The table below shows the floating point operations per cycle of various instruction sets:

CPU Extension	Floating Point Operations per CPU Cycle
SSE4.2	4
AVX	8
AVX2	16
AVX512	32

You can determine your CPU model as well as the instruction extensions supported on your **compute node(s)** with the command:

Date Published: 07/07/2022

```
[...@computenode ~]$ cat /proc/cpuinfo | grep -Ei "processor|model name|flags"
```

For model name, you should see something like "… Intel Xeon E5-26…..". If instead you see "QEMU…", please notify the course Instructors to assist you.

You can determine the maximum and base frequency of your CPU model on the Intel Ark website. Because HPL is a demanding workload, assume the CPU is operating at its base frequency and **NOT** the boost/turbo frequency. You should have everything you need to calculate the RPeak of your cluster. Typically an efficiency of at least 75% is considered adequate for Intel CPUs (RMax / RPeak > 0.75).

Part 3 - HPC Challenge

HPC Challenge (or HPCC) is benchmark suite which contains 7 micro-benchmarks used to test various performance aspects of your cluster. HPCC includes HPL which it uses to access FLOPs performance. Having successfully compiled and executed HPL, the process is fairly straight forward to setup HPCC (it uses the same Makefile structure).

1. Download HPCC from https://icl.utk.edu/hpcc/software/index.html
2. Extract the file, then enter the `hpl/` sub-directory.
3. Copy and modify the Makefile as your did for the HPL benchmark, but in this case also add "-std=c99" to the "CCFLAGS" line in the Makefile.
4. Compile HPCC from the base directory using

   ```
   make arch=<arch>
   ```

5. Edit the `hpccinf.txt` file (same as `HPL.dat`)
6. HPCC replies on the input parameter file 'hpccinf.txt' (same as `HPL.dat`). Run HPCC as you did HPL.
7. Download this script which formats the output into a readable format https://tinyurl.com/y65p2vv5.
8. Install `perl` on your head node through `dnf`.
9. Run the script with

   ```
   ./format.pl -w -f hpccoutf.txt
   ```

   To see your benchmark result, your HPL score should be similar to your standalone HPL.

!!! Have the output `hpccoutf.txt` AND `Make.<architecture>` **configuration file** ready for instructors to view on request.