19th June 2017

Maximising HPC performance on AWS public cloud

How to profile application I/O patterns with Ellexus Mistral on an Alces Flight high performance computing cluster

With the move to cloud computing comes a flexibility in the compute environment that has never been experienced before. This flexibility is great for scaling on demand and optimising on compute spend, but it can introduce challenges when tuning and optimising applications.

It used to be possible to tune a high performance computing (HPC) application and let it run for years, but major changes in compute, memory and storage capabilities that can be reconfigured at the click of a button mean that a new approach is needed for applications to be able to run well in the cloud. When the infrastructure changes, bottlenecks shift and previously masked problems can become obvious. Whether doing financial modelling, cancer research or designing the next super car, it's never been a better time to take advantage of what cloud computing can offer and ready your applications for the change. One way of doing this is by profiling file I/O and eliminating wasteful I/O patterns.

In this white paper, we look at how easy it is to spin up an HPC cluster on Amazon Web Services (AWS) using Alces Flight and profile the performance of a typical HPC application in that environment using Mistral from Ellexus. The application we have chosen is a publicly available genome pipeline from a well-respected cancer research organisation. We were looking at I/O patterns and how to characterise the requirements of the pipeline. The application was freely available as a Docker container and we used a publicly available reference genome and mutant sample to test it.

For more information about the tools used in this document, visit:

www.ellexus.com

www.alces-flight.com

I/O profiling with Mistral

Mistral is an APM I/O profiling tool from Ellexus that protects against bad I/O patterns. It is possible for applications to overwhelm shared resources such as parallel file systems and to harm their own performance with bad I/O patterns even when other optimisation techniques have been used. Mistral protects against such applications by being lightweight enough to run in protection mode all the time on your applications and flexible enough to be able to run more detailed tests to validate applications before they are run at scale.

In this study we configured Mistral with an I/O contract that gathered a large amount of information on bandwidth, meta data and file system performance to gain a detailed picture of the requirements of the pipeline and the suitability of the compute environment.

Launching an Alces Flight cluster on AWS

Alces Flight Solo is a single-user HPC cluster environment, complete with job scheduler and scientific applications, which can be used to provide a familiar Linux cluster environment suitable for running performance tests. Clusters can be created quickly and easily, with a range of different methods available for the further customisation and automatic deployment of HPC resources. Flight is compatible with a range of public cloud platforms including AWS, Microsoft Azure and IBM Bluemix as well as bare-hardware and on-premises private cloud solutions.

A freely available community supported version of Alces Flight Solo is available for use from the AWS Marketplace and provides a simple method to quickly create an HPC cluster environment in a few minutes. A step-by-step guide on getting started is included at the end of this document.

Test set up

AWS set up
We chose to run the pipeline on an AWS m4.16xlarge instance as a login node. This has 64 cores and 256GiB of memory, and 10,000Mbps bandwidth to the storage. We used “gp2” AWS Elastic Block Store with default settings for the test. The gp2 EBS is a general purpose SSD that can handle up to 10,000 IOPS and 160 MB/s per volume. These volumes have a dedicated 10Gb Ethernet network link to the instance. This was more than sufficient to run the pipeline. We didn’t want the test set up to introduce any bottlenecks in the test that might affect the results.

Mistral I/O profiling set up
Mistral is configured via an I/O contract containing I/O rules. When a rule is broken, a log message is generated detailing how much I/O was performed. So that we could see what the pipeline was doing throughout the run, a rule was written for every type of I/O and set to 1B or 1 I/O operation so that it would be triggered whenever any I/O was performed. For more information on how to set Mistral I/O rules consult the Mistral documentation on www.ellexus.com/downloads

The Mistral contract was further set up to measure reads, writes, seeks, opens, access (also known as stat), create and delete operations. Reads, writes and seeks were split by the size of the operation with rules for the following ranges: 1B, 1B-32kB, 32kB-1MB, 1MB-10MB, 10MB+. We also monitored the performance of all these operations with rules for mean and max latency.

Profiling results

Profiling results: overview
The pipeline took 10h21m to run. Mistral added an overhead of about 20% as it had all possible profiling options enabled and was gathering a lot of analytics data. Although we have identified a number of areas of improvement, it’s clear that a lot of work has gone into optimising the pipeline. The possible optimisations that we’ve highlighted below demonstrate that there is no substitute for profiling the I/O in a production environment to really understand how an application will perform.

Profiling results: 1B reads and writes
One of the first checks that we do is to look at small reads and writes. It’s very common for older libraries to read data one byte at a time. While it is possible to use some of the buffering operations available in the glibc, most don’t and so every read or write results in a system call, an operating system context switch and perhaps a call to the storage. Even though the operating system and file system will try to cache the I/O, the cache is likely to fetch and send data in small chunks that are not good for file-system performance.

The pipeline read 857MB of data in 1B operations and wrote 3.26GB of data 1B at a time. The writes happened over a period of about 40 minutes and were sustained at about a million per second. The reads occurred in three sections lasting around 10 minutes at a similar rate. The resultant bandwidth of around 1MB/s falls a long way short of what the storage can deliver so depending on the level of computation at that point in the pipeline, optimising the I/O may well speed up the pipeline within that intense 70 minutes of very small

I/O. As 10% of the pipeline is spent during this time, it's worth investigating.

A quick look at the Mistral log files tells us which program and which files have most of the IB I/O activity. A sample program and file is logged with each measurement. The 1B reads come from a program called runASCAT.R and it only took a minute to locate the common R call read.table() as the source of the 1B reads.

Profiling results: small I/O performance
The performance impact of 1B operations cannot be recorded by Mistral because even at sub-optimal levels they are too fast to measure, but we can look at slightly larger operations to see the impact of small I/O operations.

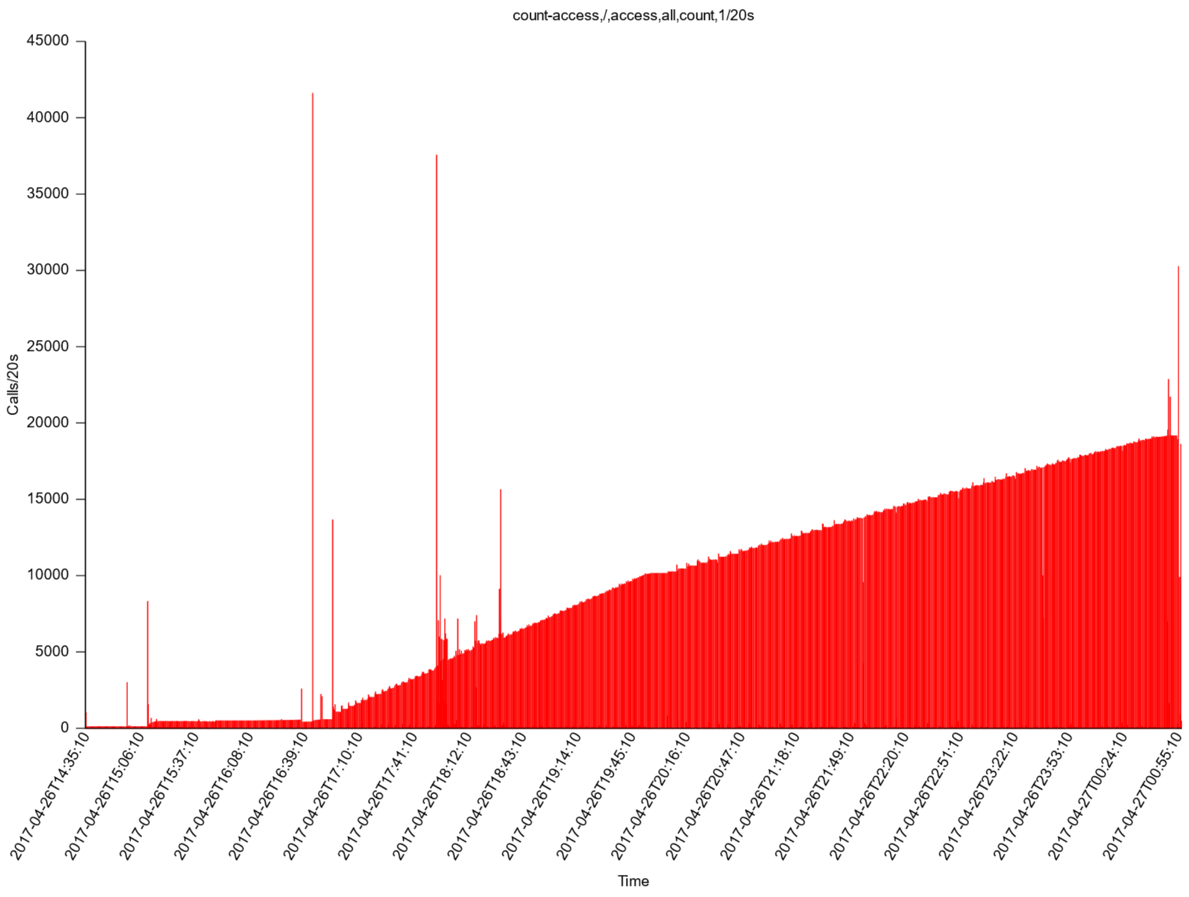
There were enough I/O operations in the 1B-32kB range and the 32kB-1MB range to provide a comparison of read performance. The performance was very fast because the system was over specified. The performance difference on a cost-effective system would be much more significant with slower I/O. Even so, you can see a big performance difference between the smaller and larger reads as the smaller I/O has larger context switch overhead and more cache misses.

Read size range	Number of reads	Average read size	Mean latency	
			per call	per byte
1B-32kB	1.4 Billion	707 B	1.5 us	2.1 x10^{-3} us
32kB-1MB	2.3 Million	62 KB	2.7 us	4.3 x10^{-5} us

Almost all the writes that were larger than 1B were in a single range: 121GB written out on average 218 bytes at a time. This means there is potentially an opportunity to make sure that the writes are buffered as much as possible in user space before being flushed.

Profiling results: Meta-data
Like most genome pipelines, this pipeline creates small files throughout and then deletes most of them right at the end. This can put a strain on the file system, particularly if many pipelines do this on shared storage at the same time.

In addition to the activity of creating and deleting files, the pipeline has also been fitted with a progress monitor, which is a sensible addition to a long-running application. This works by running stat() or access() on the directory containing the temporary files and progress is measured by how many it finds. While this is a simple and effective measurement, towards the end of the pipeline there are nearly 20,000 files that are accessed every 30 seconds. When performed in parallel this puts unnecessary load on the meta-data server which is often affected by bad I/O patterns elsewhere. A simple redesign of this mechanism would release the load and lower the risk of overloading the meta-data server.



Conclusion

In this pipeline, optimising the small reads and writes seems like an easy win to save the CPU as much as the storage. It would also be a good idea to put the temporary files on instant-disks that are directly attached to an instance and private. This would mitigate the effects of hitting shared storage with so many small files and small I/O operations.

The main conclusion of this study is that even well designed and well-maintained code can hide I/O issues if the code has not been profiled in a real environment. To run sub-optimal code in the cloud is an obvious waste of money. Alces Flight and Ellexus Mistral make it easy and quick to test the suitability and performance of applications in the cloud to facilitate sizing and optimisation processes.

How to launch Mistral on a Docker container

The pipeline chosen was available as a Docker container. If the application is available in another format then follow the instructions that come with Mistral on how to set up and launch the application.

Step 1: Docker may not in be installed by default. To install Docker use apt-get or the appropriate equivalent command.

Step 2: Download Mistral from www.ellexus.com/downloads, save the licence file in the same directory, set up the contract or use the default contract that comes with Mistral. Then set up the Mistral starter script starter.sh with the following parameters:

```
#!/bin/bash

export
MISTRAL_CONTRACT_MONITOR_GLOBAL=/ellexus/global_contract
export MISTRAL_LOG_MONITOR_GLOBAL=/ellexus/log
export MISTRAL_RLM_LICENSE=/ellexus/dev.lic

/ellexus/mistral <docker entrypoint>
```

Step 3: The last thing to do is to run the Docker image, adding the directory volumes to make Mistral available inside the container namespace, overriding the entrypoint to

```
docker run -v mistral:ellexus/mistral --entrypoint
/ellexus/starter.sh <path to docker image>
```

Step by step for launching a cluster on AWS

The following steps were taken to provide an environment for the testing outlined above:

1. Create a user account on AWS, or get permission to use an existing account. The account should be enabled to use EC2, Cloudformation and S3 services, and be authorized to create new VPCs. New accounts typically include limit the EC2 resources which can be launched – review the EC2 limits in place before choosing to launch compute clusters with many compute nodes. Users should create or import an SSH keypair registered in the AWS region they want to use.

2. In a web-browser, navigate to "AWS Marketplace" – the app-store for the AWS public cloud. Search for "Alces Flight" and select the Solo Community edition. Review the information on the product page and click on the "Subscribe" button when ready. There is no charge for subscribing to the community edition of Flight Solo.

3. On the product launch page, choose the "Manual Launch" tab; choose the AWS region where you want to run and choose the "Personal HPC compute cluster" deployment option. Click the "Launch in Cloudformation console" button when ready.

4. Click the "Next" button to continue in the Cloudformation console; make the following choices to launch your cluster:

   a. Enter a name for your stack (e.g. "cluster1")
   b. Choose an existing Cluster Administrator SSH keypair from the drop-box
   c. For the Access Network Address, use "0.0.0.0/0"
   d. Choose the login node instance type "m4.16xlarge"
   e. Choose the desired compute node type, number and spot price to bid. For example, for the lowest cost nodes, choose a quantity of one "c4.large" type and set the spot bid price to 0.01.
   f. Choose the size of the login node system volume; this filesystem will be mounted across all compute nodes in the cluster.

Click the "Next" button when ready; optionally, set any tags desired for the cluster instances and click the "Next" button again. Review the settings chosen; when ready, select the checkbox to allow creation of the cluster and click on the "Create" button.

Users are returned to the AWS CloudFormation console, which will report the status of the cluster as it is created. Once the cluster stack status shows "CREATE_COMPLETE", click on its entry in the table and select the "Outputs" tab below. The cluster login node IP address will be shown, along with the login user name. Use these with your SSH private key to log in to your HPC cluster.

For more details of launching and customising your own HPC cluster on AWS, see the Alces Flight online documentation:

http://docs.alces-flight.com/en/stable/launch-aws/launching_on_aws.html