12th July 2018

Profiling OpenFOAM on Oracle HPC cloud with Ellexus' analytics tools

The software tool OpenFOAM is used extensively in high-performance computing (HPC) to create simulations, but is known to have challenging I/O patterns. To uncover the reasons why, we profiled OpenFOAM with the Ellexus I/O profiling tools on the Oracle bare metal cloud.

The results detailed in this whitepaper provide a clear picture about why the tool performs slowly at times and highlight key areas for improvement.

This whitepaper has been produced by Ellexus, the I/O profiling company.

For more information about Ellexus' tools, including case studies, videos and blogs, visit

www.ellexus.com

OpenFOAM

OpenFOAM is free, open source software for computational fluid dynamics (CFD) from the OpenFOAM Foundation. The tool includes hundreds of applications that can be extended and customised, which means it is widely used to create simulations in many industries and academic, research and scientific institutions – including HPC. OpenFOAM is packaged for installation on Ubuntu Linux, which can be directly installed on Windows 10 and is available as a Docker image for other Linux and macOS.

OpenFOAM is an MPI application and Breeze automatically detects when the master process is spawning addition MPI ranks. In this set up, OpenFOAM runs on a master machine and two slaves so Breeze made three traces, one for each machine, with around 50 MPI ranks on each.

Oracle Cloud

Oracle is one of the leading cloud platform providers on the market. It has various offerings, including software-as-a-service, platform-as-a-service and infrastructure-as-a-service.

In this case we selected Oracle Cloud Infrastructure, Oracle's bare metal cloud which is designed to compete with on-prem offerings. One advantage to using a bare metal cloud is that the virtualisation needed on other cloud platforms can be a problem for I/O intensive workloads, such as those found in the high-performance computing (HPC) sector.

Ellexus' I/O profiling tools

Ellexus has two I/O profiling tool suites, Breeze and Mistral. The tools are used widely in the HPC and scientific computing sectors to identify ways to improve storage architecture and plan the migration of applications and data between different storage architectures – either on-premise or in the cloud. As well as detailed information about how each program has accessed each file, Breeze and Mistral also gather CPU and memory stats.

The Breeze product range is design to inspect an application in detail, recording information about every program and file accessed with I/O patterns in detail. It is used by IT managers and users to trace application file and network dependencies, to work out why an application runs in one environment and not another and to profile application I/O.

Mistral also captures application I/O, but in less detail so it is lightweight enough to be run in production. Mistral will report an aggregate of how much data is read or written by an application or how many meta data operations are carried out.

The information gathered by both tools can be used in multiple ways; to optimise storage, to gain a picture of application dependencies, to develop a

go-to-cloud strategy, to improve security and accountability or to improve quality of service across a cluster.

The trace

The following commands were used to generate the trace of OpenFOAM:

> *./trace-program.sh --variant=mpich -f tests/trace_out_mpi_foam mpirun -np 150 -ppn 50 -f=/home/opc/hostfile simpleFoam -parallel*
>
> *./mistral --variant=mpich --traffic-light-log=/path/to/tlr.log mpirun -np 150 -ppn 50 -f=/mnt/blk/share/data/OpenFOAM/motorBike/hostfile simpleFoam -parallel*

Mistral was configured using a contract file that measured the bandwidth, maximum latency, mean latency and counts for all types of file access over a variety of size ranges.

Results of the Breeze trace

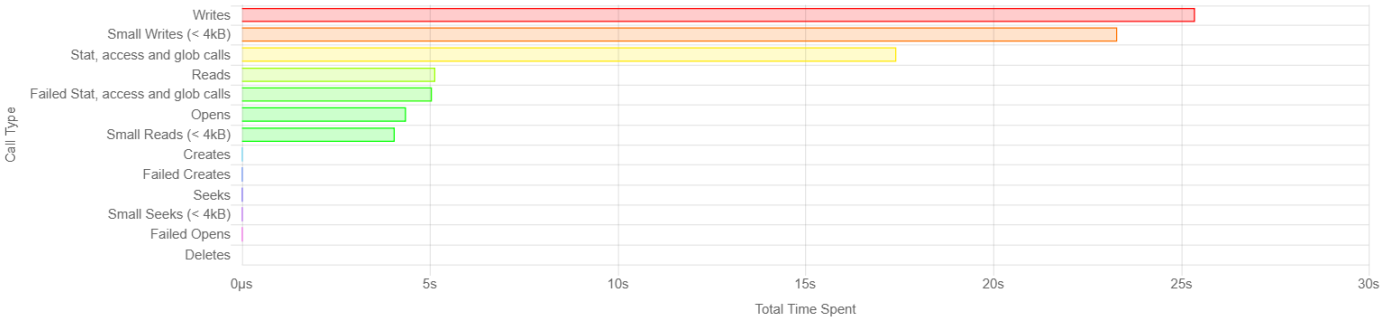
Most of the information in this section was pulled directly from the Healthcheck report, which is part of the Breeze tool suite. To obtain additional information we fired up the Breeze GUI and drilled down into the data to reveal individual ranks or specific measurements.

The following command was traced and the application took 4m 39s to run under Breeze:

```
/opt/intel/compilers_and_libraries_2018.1.163/linux/mpi/bin64/pmi_proxy --control-port hpc-master-6b686.sub08032221500.main.oraclevcn.com:53771 --pmi-connect alltoall --pmi-aggregate -s 0 --rmk user --launcher ssh --demux poll --pgid 0 --enable-stdin 1 --retries 10 --control-code 544255790 --usize -2 --proxy-id 2
```

Overview of time spent doing file I/O for the master compute node

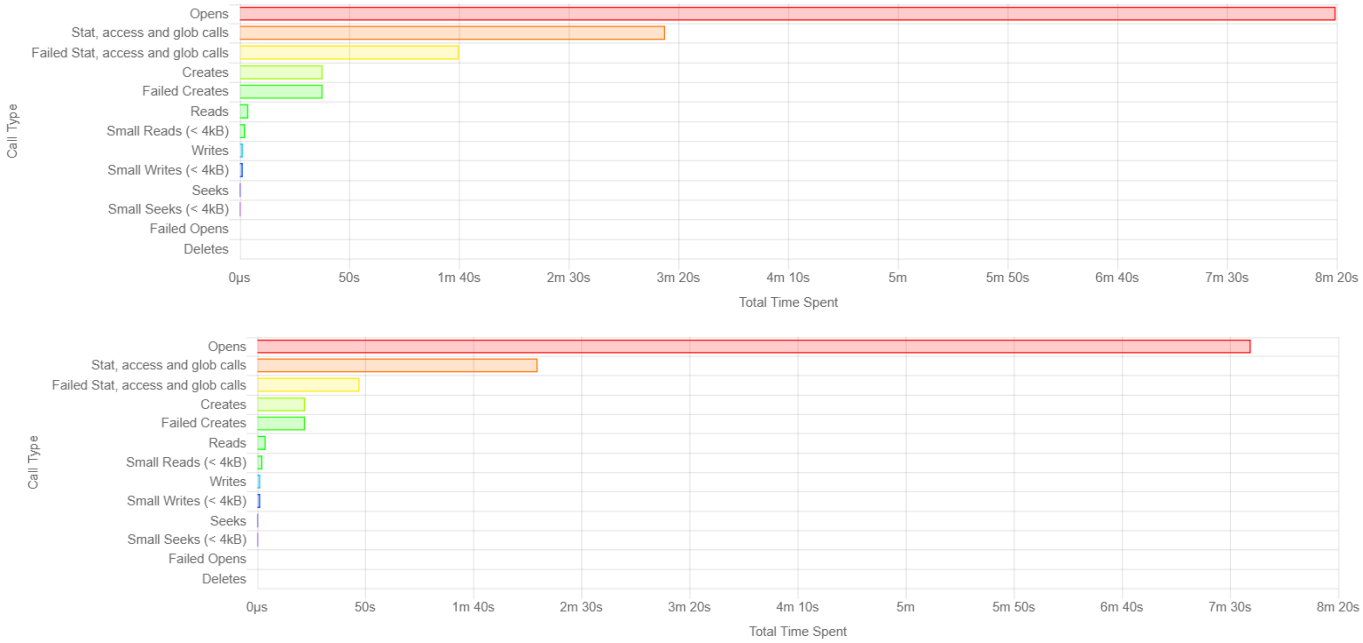
There are a few points to note as soon as we look at the I/O for the master node. Most of the time doing file system I/O is spent doing writes and most of that is small writes. Small writes should be optimised to take advantage of the high-speed streaming I/O capabilities of shared storage.



The next point of interest is the amount of time spent doing stat() or access() calls. These are meta-data reads that look up information about a file such as existence, permissions or size. The application spends almost as much time reading meta data as it does writing and it far exceeds the amount of time spent reading file data. Given that meta data accesses are usually faster than data reads, the application must be reading far more meta data than data so there is scope for optimisation there.

Overview of time spent doing file I/O for the slave compute nodes

The two slave nodes have very different I/O profiles from the master node. The amount of time spent waiting on open() calls across all processes exceeds the total run time of the application! At any given time there are at least two processes waiting on open() calls throughout the program execution. That is extraordinary and puts a huge load on the file system. Even quite I/O intensive applications usually spend a small proportion of their time waiting on I/O. To spend more than the run time waiting on meta data especially is unusual and highlights why this application is known for having such a heavy load on the I/O system.

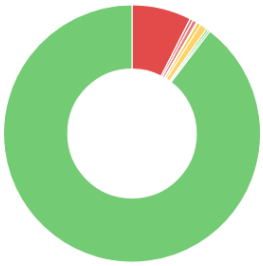


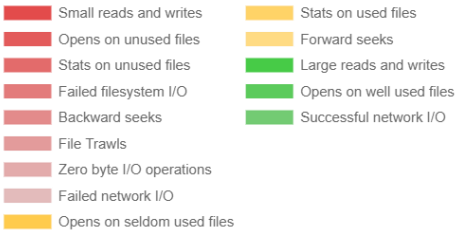
In addition to the large amount of time spent on open calls, there is also much longer spent on stat() or access() calls and half that time again on failed access() calls. This is usually caused by a program looking for a file that doesn't exist. Given the number of failed I/O calls, that is another area that should be optimised by a redesign.

Good vs Bad I/O

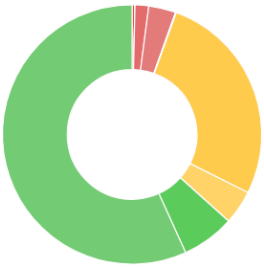
The following reports break down the I/O into good, bad and medium I/O. In this case we have included network I/O. The large amount of network I/O is due to the MPI communication between ranks.

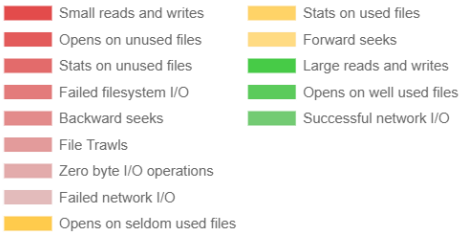
Master traffic light report from Breeze

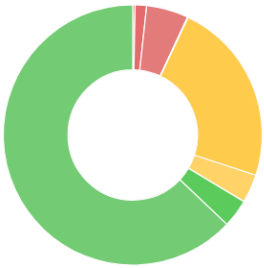


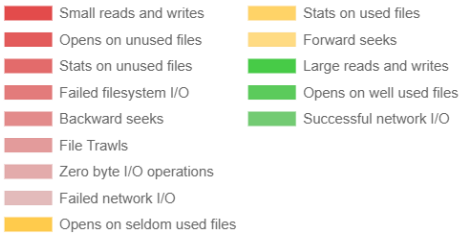


Slave traffic light reports from Breeze









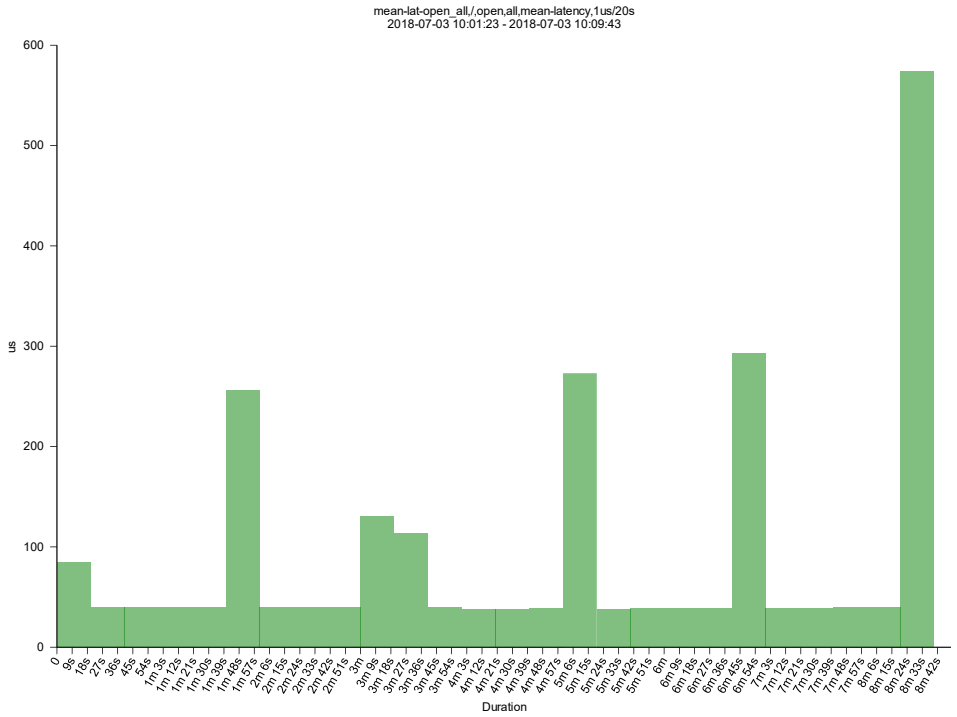
All three nodes spend a lot of time doing network I/O, but again you can see the impact of open() calls on the slave nodes. Breeze has listed a lot of the time spent opening files as being on files that are seldom used, ie they have few read() or write() operations once they are opened.

All three nodes spend some time doing small reads and writes and given the amount of I/O performed there would be some benefit to fixing that even though the proportion of time spent is relatively low. Small reads and writes harm the performance of the shared file system and will get worse as the application scales. If this was run on more machines in parallel then the wait time for those small I/O operations would increase and the impact on the application could be much more significant.

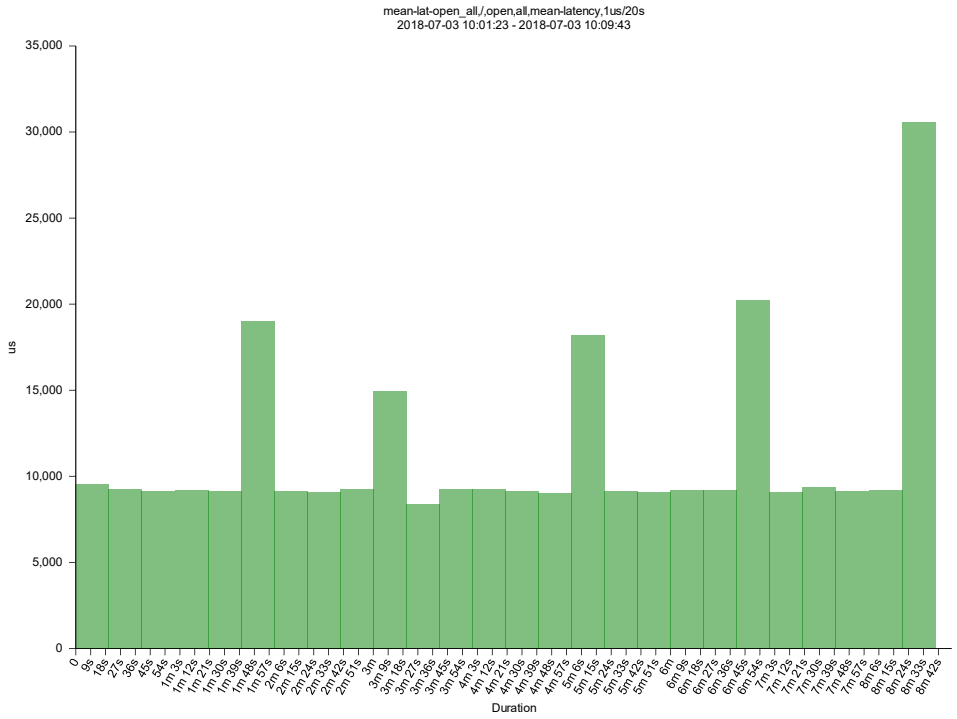
Open call performance

Assuming the ranks on the master node vs the slave nodes do a similar number of I/O operations, the time taken to the perform the I/O operations must be very different. This can be deduced from the measurements taken using Mistral.

Latency of open calls on the master compute node



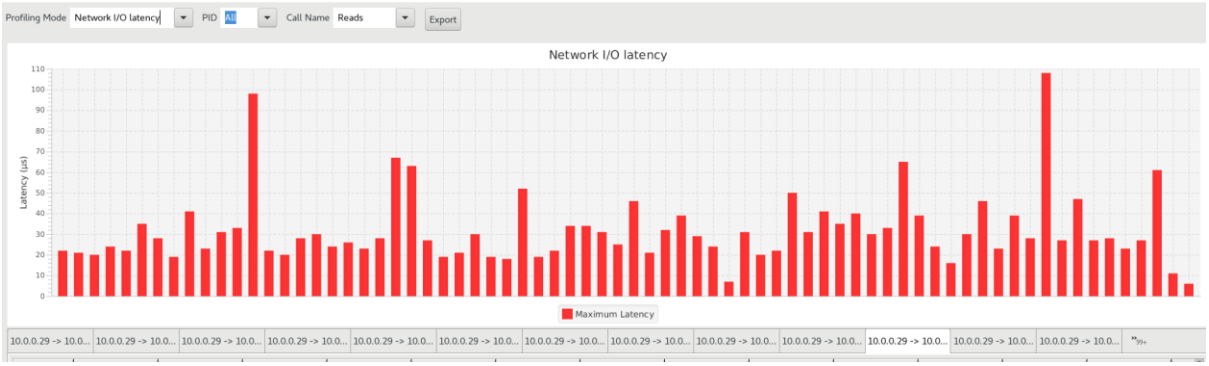
Latency of open calls from one of the slave nodes



Most of the time, the latency of the open calls on the master node is about 50us whereas it is consistently above 9ms on the slave nodes. Therefore, it takes about 200 times as long to open a file on the slave nodes as it does on the master nodes. This highlights the importance of making the right storage choices for these I/O intensive applications.

Network performance

The performance of the network cannot be ignored. Not only do all the I/O operations for remote storage have to go over the network, but all the direct communication between MPI ranks has to go over the network as well. Breeze lets you look at the network performance over time for individual connections. In this case the performance was fairly consistent throughout, but this should be monitored as the application is scaled to more nodes.



Conclusion

There are many areas in which this application could be improved, but even with those improvements the I/O performance of the system will remain critical to the performance of the application, particularly as it scales.