First things first…
For reproducibility, please find all our launch, provisioning, building and benchmarking scripts at https://github.com/DUSCC/ClusDur-CIUK22

Challenge 1: alcesflight

Team ClusDur

1. Challenge 1: Virtual or bare-metal?

1.1 Method

1.1.1 Hardware

The main objective of this challenge is to determine the impact of virtualization on benchmarking results, and not necessarily to achieve the highest scores in comparison to other hardware architectures.
Therefore, we originally planned to run all single-node benchmarks on the cheapest nodes in terms of occupancy cost. Since these were continuously allocated, we resorted to using baremetal.amd.128.100gb and vm.amd.128 instances, which were the cheapest ones available during our tests.

We used `lscpu` to detect CPU characteristics, as shown in Figures 1 and 2.

```
[centos@clusdur-amd128 ~]$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              128
On-line CPU(s) list: 0-127
Thread(s) per core:  1
Core(s) per socket:  1
Socket(s):           128
NUMA node(s):        1
Vendor ID:           AuthenticAMD
CPU family:          25
Model:               1
Model name:          AMD EPYC-Milan Processor
Stepping:            1
CPU MHz:             2445.406
BogoMIPS:            4890.81
Hypervisor vendor:   KVM
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            512K
L3 cache:            32768K
NUMA node0 CPU(s):   0-127
```

Figure 1: CPU properties on virtual instance instance

```
[centos@clusdur-amd128-baremetal ~]$ lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              128
On-line CPU(s) list: 0-127
Thread(s) per core:  1
Core(s) per socket:  64
Socket(s):           2
NUMA node(s):        2
Vendor ID:           AuthenticAMD
CPU family:          25
Model:               1
Model name:          AMD EPYC 7763 64-Core Processor
Stepping:            1
CPU MHz:             2445.350
BogoMIPS:            4890.70
Virtualization:      AMD-V
L1d cache:           32K
L1i cache:           32K
L2 cache:            512K
L3 cache:            32768K
NUMA node0 CPU(s):   0-63
NUMA node1 CPU(s):   64-127
```

Figure 2: CPU properties on baremetal

And, retrieved memory characteristics with `sudo lshw -C memory` to detect memory type and size, as shown in Figure 3.

```
  *-bank:0
       description: DIMM DDR4 Synchronous Registered (Buffered) 3200 MHz (0.3 ns)
       product: 36ASF8G72PZ-3G2E1
       vendor: Micron Technology
       physical id: 0
       serial: 32F3B249
       slot: A1
       size: 64GiB
       width: 64 bits
       clock: 3200MHz (0.3ns)
```

Figure 3: Memory properties on baremetal instance

The compute nodes are connected via 1 Gb Ethernet:

* We resorted to the AMD nodes since the actual cheapest Intel nodes were continuously allocated.

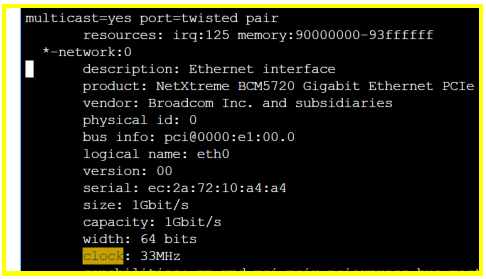


Figure 4: Network interface

1.1.2 Software Stack

Our software stack is based on the CentOS 8 image provided by you (Thanks!), the benchmarks stated in Section 1.2, their dependencies such as OpenMPI and OpenBLAS, as well as packages required to install them such as git, wget, make and cmake.

1.2 Benchmarking

A. Single-core integer performance

To determine single-core integer performance, we use the Fhourstones benchmark. It computes the speed that a CPU can search for connect four game positions that will result in a victory per second and the performance metric is Kpos/sec (kilo positions per second) with higher being better.

The Kpos/sec result is a relative performance metric and can only be compared to Fhourstone runs with the same set up. This allows virtual and barmetal results to be easily compared to each other and compared to other runs.

This benchmark was used as it is open source and has been widely used as opposed to the spec benchmark int_base which is typically used in industry and neither open source nor freely available.

This benchmark was installed from https://github.com/qu1j0t3/fhourstones and compiled using the -O3 optimisation flag. We ran this using the default input parameters and saved the output file.

On the virtual instance, the benchmark yielded 10319.2, and on the bare-metal instance the benchmark yielded 14551.0. Sadly, we weren't able to compute a theoretical maximum for this due to the unobvious numbers of instructions and instruction types used by the benchmark. While we were not able to estimate a theoretical value, we at least compared our result against results published by openbenchmarking for a similar AMD architecture that led to 13755 Kpos/s.

B. Single-core floating point performance

To determine single-core floating point performance, we use High Performance LINPACK. It solves a linear system to measure GFLOPS where higher is better. As a team we have experience using this benchmark and it is used frequently to test the top 500 supercomputers.

We installed OpenBLAS and OpenMPI and compiled HPL using gcc. We installed OpenMPI via yum but had to install OpenBLAS manually. While different BLAS and MPI libraries can be used, we have generally had a good past experience with these ones.

The benchmark was run with the following parameters:
N = 10,000 NB = 191 P = 1 Q = 1

This allowed for a small problem size which minimises compute time and therefore cost, while balancing a high performance run. Our HPL.dat file is based on best practice guidelines written by AMD for HPL on the zen architecture as outlined in
http://developer.amd.com/wp-content/resources/56420.pdf.
The process grid should be 1x1 as we're only working on a single core.

On the virtual instance, the benchmark yielded X (unknown since we could not reserve an AMD 128 core machine once more), and on the bare-metal instance the benchmark yielded 47.04 GFLOPS. This was surprisingly good given we initially calculated our expected value at 39.2 GFLOPS, but

quickly realised that running this on a single core meant boost clock was enabled, which using this frequency yields a theoretical best performance of 56 GFLOPS.

C. Multi-core single-node performance
To determine multi-core single-node floating point performance, we used the same HPL benchmark compiled exactly the same way. We simply changed a couple of parameters to accommodate for the 128 cores available.

The parameters that we used were:
N = 60,000 NB = 191 P = 8 Q = 16

We used a larger problem size to accommodate for majorly increased compute power while maintaining a short (sub 2-minute) run that should yield relatively good performance. We found from experimentation that a process grid where P < Q by a small factor provides the best performance.

On the virtual instance, the benchmark yielded 1824 GFLOPS, and on the bare-metal instance the benchmark yielded 2913.1 GFLOPS. This aligns with our expectations since the bare-metal result is about 60% of peak performance for these CPUs, calculated via:
Theoretical max GFLOPS = Num cores * Num sockets * Base Frequency * Double Instructions per Cycle
which is typically what you'd expect for a HPL run where you use a smaller problem size than ~90% of available memory. This helps balance cost and time to performance, achieving the best of both worlds.

We must admit we were quite surprised at the difference between bare-metal and virtual instances on this benchmark, with bare-metal achieving much higher results. We suppose this is due to locality information in terms of NUMA/Sockets not being in alignment with the actual hardware (compare Figures 1 and 2), and hence inhibiting continuous core binding via –bind-to core.

D. Memory bandwidth
To determine memory bandwidth, we use the STREAM benchmark. It measures the memory bandwidth by reading and writing large arrays and the performance metric is MB/s (higher is better).

The benchmark was compiled by simply installing the benchmark and compiling (using GCC) with the -fopenmp flag and -O3 flag.

We increased the problem size such that the array was of a size a little over 4x the amount of CPU cache available, as to ensure that we were truly testing memory bandwidth. We also made sure to set OMP_NUM_THREADS=64. We actually found that setting this to 128 nearly halved the performance. We speculate that this could be due to two sockets halving separate caches for cache references, causing an overhead of each CPU accessing the others cache.

On the virtual instance, the benchmark yielded 92GB/s (although this was Intel node due to time constraints and other nodes being busy), and on the bare-metal instance the benchmark yielded 933796.8 MB/s (934 GB/s) (this one was a smaller problem size so all fits into cache).This does not align with our expectation for the DDR4 RAM @ 3200 MHz, which should provide maximally 25.6 GB/s according to
https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/modules/rdimm/ddr4/asf36c8gx72pz.pdf?rev=52efd03dbf8648a5b59fdc7d1cd4bb31. This results from the large caches (32MB L3!!) which allow for a large number of cache hits and hence lead to higher bandwidth and lower latency as most data will be cached (or, in other words, we should have chosen a higher array size to measure memory bandwidth instead of cache bandwidth). For the virtual instances, its more in line with what we expected but still quite a lot higher! I'm suspecting potentially accessing memory in parallel (multiple RAM sticks) and therefore we can multiply this max bandwidth, perhaps by 4 sticks would seem reasonable?

E. Interconnect latency
To determine interconnect latency, we use the Intel IMB benchmark. It computes interconnect latency for fixed message sizes and the performance metric is s (lower is better)

The benchmark was compiled using the makefiles provided after installing and configuring MPI. We ran using the IMB-example executable provided as we struggled to get the IMB-MPI1 compiled/working (did not work using instructions suggested in documentation), which we believe affected our results somewhat. We ran this with the command:

```
/usr/lib64/openmpi/bin/mpirun -n 2 --host 10.151.15.73 --host localhost ./IMB-example
```

Due to time and unavailability of other nodes, we only managed to test this on the virtual instances of the intel nodes. For a small packet (4KB) the time reported was 0.247 ms, and we'd expect this to be slightly faster for bare metal instances, and even faster if we had time to configure Infiniband. This is slower than we'd expect, but given it being virtual instances, using only an example test case and not having infiniband, this result is understandable.

F. Interconnect bandwidth

To determine interconnect bandwidth, we still used the IMB benchmark. The metric for this is in MB/s (higher is better).

We followed the same steps as latency but instead looked at the largest packet size available (4MB). We also calculated bandwidth by simply calculating (1/latency)*packet size (as we can be sending some packet size x, n times per second). While this is a rather crude manner to calculate bandwidth, in theory it should work. We had hoped to both perform this across multiple different types of nodes and instances, but we were simply constrained by time and resource availability. We also hoped to perform this on larger message sizes where we'd expect improved results.

On the virtual intel instance, the benchmark yielded 560.37 MB/s which is slightly lower than what we expected, but for the reasons detailed above this is understandable. The theoretical for this could even be up to 50 GB/s which we are quite far off (if infiniband was installed), nevertheless these aren't bad results.

G. I/O performance to disk

To determine I/O performance to disk, we use the iozone benchmark. It computes read and write times to disk and the performance metric is kB/sec (higher is better).

The benchmark was compiled by downloading and extracting the zip, and simply running the make file with "make linux".

We ran the benchmarks using the following commands:

1 Core:

```
OMP_NUM_THREADS=1 ./iozone -t 1 -i 0 -s 10G -r 1M -w -e > ~/results/iozone_w_1.txt
OMP_NUM_THREADS=1 ./iozone -t 1 -i 1 -s 10G -r 1M -w -e > ~/results/iozone_r_1.txt
```

64 Core:

```
OMP_NUM_THREADS=64 ./iozone -t 64 -i 0 -s 156M -r 1M -w -e > ~/results/iozone_w_64.txt
OMP_NUM_THREADS=64 ./iozone -t 64 -i 1 -s 156M -r 1M -w -e > ~/results/iozone_r_64.txt
```

The -t flag indicates how many threads to use, and -i specifies write (0) or read (1). -s specifies the size per core and -r specifies the size we read at a time.

Due to the number of results, we shall simply direct you to look at the table in the next section. The results mostly aligned with our expectations in that reading is much faster than writing (and more cores massively sped up reading!), but were surprised to see that more cores pretty much didn't speed up writing at all, and reading with 64 cores was double the speed on bare metal vs virtual.

1.3 Conclusion

The following table reports the **best** results we achieved for each characteristic, with the benchmarks and metrics as outlined in Section 1.2. (~~Please imagine this to be a marvellous bar plot that shows how much better/worse one instance is relative to the other. Unfortunately, we were busy playing with your exciting HPC hardware which was simply more fun than beautifying data 😀. Sorry!~~) Ignore that! We found a little extra time to provide you with some wonderful plots on the next page!

Characteristic	**Metric**	**Theoretical peak**	**Virtual**	**Bare-metal**
A. Single-core integer performance	Kpos/s	13755*	10319.2	**14551.0**
B. Single-core floating point performance	GFLOP/s	56**		**47.04**
C. Multi-core single-node performance	GFLOP/s	5017.6	1824	**2913.1**
D. Memory bandwidth	MB/s	25600***	92329.4	**933796****
E. Interconnect latency (len=4096B)	ms	0.044336	0.247251	
F. Interconnect bandwidth	MB/s	3125*****	560.37	
G. I/O performance to disk:				
Write - 1 Core	kB/sec		371330.56	**376423.19**
Read - 1 Core	kB/sec		14449725.00	**15156862.00**
Write - 64 Core	kB/sec		394917.51	**399835.02**
Read - 64 Core	kB/sec		46897264.33	**108879387.38**

***** estimation of minimum value to be expected based on results published by openbenchmarking for a comparable architecture

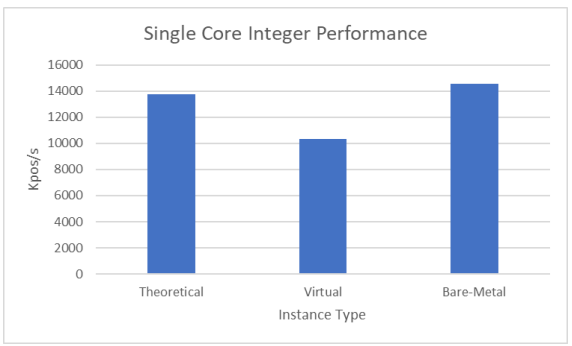
** Using boost clock freq as 1 core!

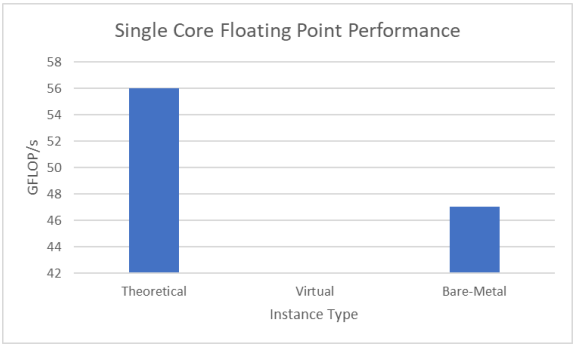
*** Per stick of RAM we think??

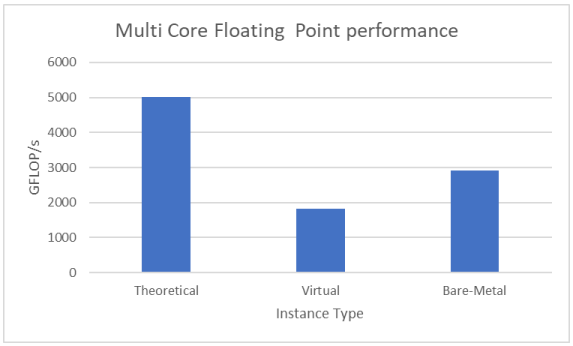
**** This was reading from cache due to small problem size but we ran out of time to change!

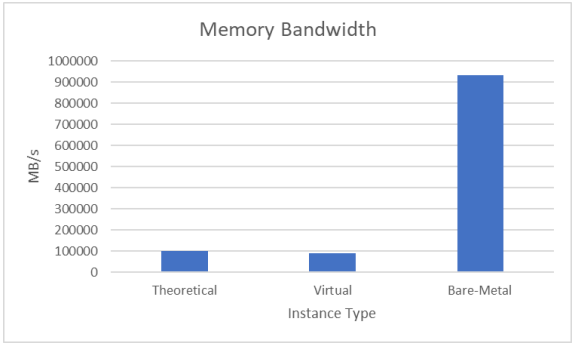
***** Due to 25 Gb Ethernet on the intel nodes

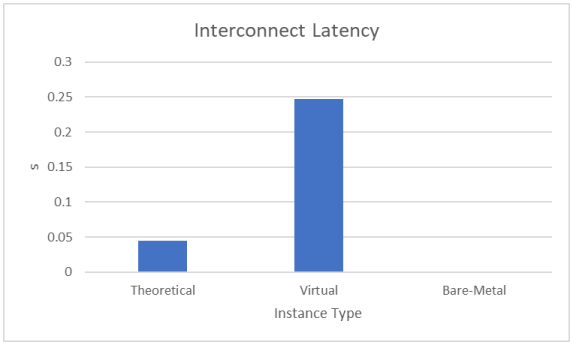
Answering the introductory question: For this hardware and KVM configuration, performance on the baremetal instance is better in every respect. While the difference between both is surprisingly high for the compute-bound HPL (probably due to thread pinning), it is rather low for the cache bandwidth and I/O tests (except for Read - 64 Core).

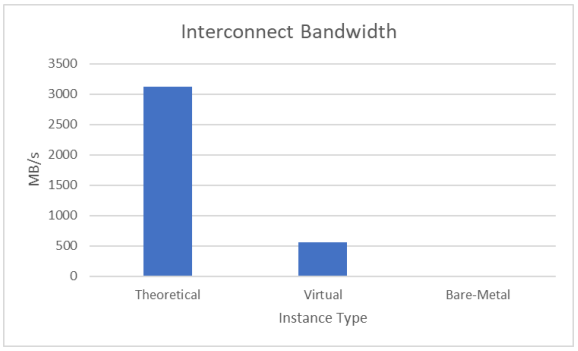

Single Core Integer Performance
Kpos/s
16000
14000
12000
10000
8000
6000
4000
2000
0
Theoretical
Virtual
Bare-Metal
Instance Type

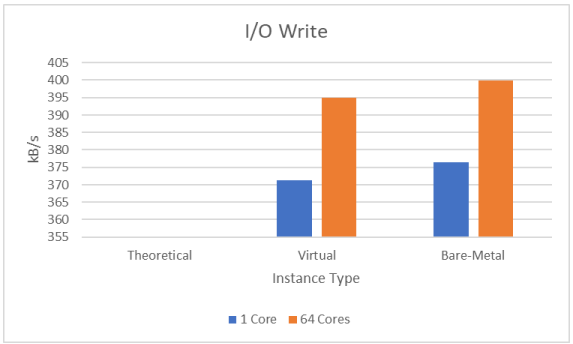

Single Core Floating Point Performance
GFLOP/s
58
56
54
52
50
48
46
44
42
Theoretical
Virtual
Bare-Metal
Instance Type

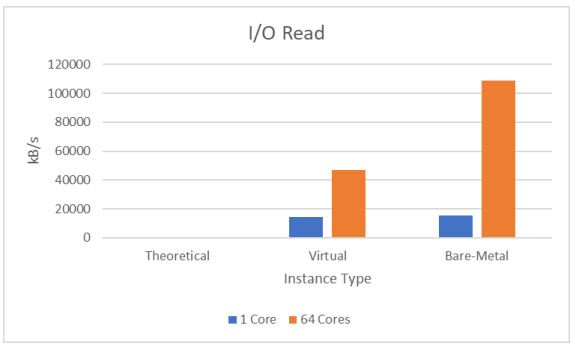

Multi Core Floating Point performance
GFLOP/s
6000
5000
4000
3000
2000
1000
0
Theoretical
Virtual
Bare-Metal
Instance Type


Memory Bandwidth
MB/s
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
Theoretical
Virtual
Bare-Metal
Instance Type


Interconnect Latency
s
0.3
0.25
0.2
0.15
0.1
0.05
0
Theoretical
Virtual
Bare-Metal
Instance Type


Interconnect Bandwidth
MB/s
3500
3000
2500
2000
1500
1000
500
0
Theoretical
Virtual
Bare-Metal
Instance Type


I/O Write
kB/s
405
400
395
390
385
380
375
370
365
360
355
Theoretical
Virtual
Bare-Metal
Instance Type
1 Core
64 Cores


I/O Read
kB/s
120000
100000
80000
60000
40000
20000
0
Theoretical
Virtual
Bare-Metal
Instance Type
1 Core
64 Cores

2. Challenge 2: Running a real application

2.1 Method

We initially started off using phoronix test suite to run this test but found that we could not easily automate the test process and that phoronix would install and compile the GPU benchmark but not give an option to run it. We also realised that running gromacs through phoronix would not offer the ability to properly experiment with and tune the number of threads or allow us to enable newer options within gromacs that offload additional work to the GPU. Because of this we decided to write custom scripts to automate installing, building and running both CPU and GPU benchmarks using gromacs. This gave us the flexibility to experiment with the program parameters and also had the added benefit of significantly reducing compile times as we no longer had to build both CPU and GPU benchmarks on install and could specify the exact cuda architecture to build for.

During research we learned from https://docs.bioexcel.eu/gromacs_bpg/en/master/cookbook/cookbook.html that Gromacs can very efficiently use A100 GPUs however it doesn't necessarily as scale well across multiple GPUs. Additionally the nvidia research article https://developer.nvidia.com/blog/creating-faster-molecular-dynamics-simulations-with-gromacs-2020/ was helpful in understanding how nvidia enabled their GPU optimisations and the relevant settings we could enable for optimal performance.

Our testing process included:

- Install prerequisites like wget, tar, gcc, gcc-c++, make, cmake, cuda and nvidia drivers
- It took us a while to sort out Cuda installation since instructions generated by Nvidia https://developer.nvidia.com/cuda-11-7-0-download-archive?target_os=Linux&target_arch=x86_64&Distribution=CentOS&target_version=7&target_type=rpm_local) did not work for us due to missing dkms and the nvidia driver magically disappearing temporarily. Finally sorted it out: https://github.com/DUSCC/ClusDur-CIUK22/blob/main/scripts/cuda-install.sh
- Use wget and chmod to download the relevant script from our github repository and make it executable. Scripts can be found here as gromacs-cpu and gromacs-gpu: https://github.com/DUSCC/ClusDur-CIUK22/tree/main/scripts
- Execute the script
- Results are stored in ~/gromacs-challenge-gpu/gromacs-gpu-results.txt for GPU runs or ~/gromacs-challenge-cpu/gromacs-cpu-results.txt for CPU runs
- estimated costs and power for servers to decide on which machines to run
- planned to observe power consumption with nvidia-smi, but nvidia-smi needs longer to start-up then the job to run

Because of the results from challenge 1 we decided to execute all runs on the baremetal machines. Below there are tables comparing the performance, cost and power usage of the best machines we ran the benchmark on. In general our tactic was to use the Nvidia A100

server for the fastest run time category and use cheaper servers like the intel.25gb for the lowest cost and lowest carbon categories.

2.2 Cost and Power Estimation

2.2.1 intel.25gb

Component	Quantity	Performance [GFLOPS]	Cost [£]	Power [W]
GPU	N/A	N/A	N/A	N/A
CPU	2	1228.8	680	135
Total Server	**1**	**1228.8**	**12000**	**380**

2.2.3 nvidia.a100 (Using only 1 GPU)

Component	Quantity	Performance [GFLOPS]	Cost [£]	Power [W]
GPU	1 (4 in server but only 1 in use)	19500 (No tensor cores)	38242.79	250
CPU	1	N/A*	N/A**	280
Total Server	**1**	**19500**	**125000**	**2800**

* basically everything offloaded to GPU
** Since overall price is known

2.2.4 Conclusion

The most GFLOPs are provided by the nvidia.a100. Therefore, we execute our fastest Gromacs run on it.
The most cost-efficient server is baremetal.intel.25gb. Therefore, we execute our cheapest Gromacs run on it.
The most energy-efficient server is baremetal.intel.25gb. Hence, we execute our lowest carbon Gromacs run on it.

2.2 Results

The following Table provides an overview of our best Gromacs runs.
Run cost is calculated by assuming the hardware will last for 3 years and be running continuously in that time. Energy is calculated from the maximum server power draw as specified by Alces, it may be that in fact we use less power in some cases (Especially on the baremetal Nvidia A100 as we are only using 1 GPU).

Run	Instance	Processor	Performance	Execution	Run Cost	Energy

			[ns/day]	Time [s]	[£]	[J]
Fastest	baremetal.nvidia.a100	GPU	8.478	10.212	0.01321	28593.6
Lowest Cost	baremetal.intel.25gb	CPU	3.873	44.634	0.005661	16959
Lowest Carbon	baremetal.intel.25gb	CPU	3.873	44.634	0.005661	16959

3. Challenge 3: Putting it all together

We provisioned our software stack using Python, Ansible and Bash….
…but a script is worth a 1000 words:
https://github.com/DUSCC/ClusDur-CIUK22/blob/main/workflow.py

…and a video even more 😬
https://youtube.com/shorts/unm8PH25gGE?feature=share

The script has been made into a CLI tool for ease of use, moving through all of our stages: Launch, Provision, Benchmark, Report, Delete

In order to start this workflow, you simply type “workflow (server_name) (flavor)” in our team’s environment (this has been aliased into .bashrc to execute the python script) with optional flags to disable automatic provisioning or deletion (--no-provision or --no-delete). In the Launch phase of this script, this will create a server, assign an IP and wait for an ACTIVE state. The script will then send ssh connection requests to the server, and will wait until the server is reachable.

If the server is not reachable, after 24 retries (roughly 2-3 minutes) it will automatically delete the instance.

```
[team2@ciuk22login ~]$ workflow ClusDur-baremetal-intel baremetal.intel.25gb
[0:00:00] Creating server ClusDur-baremetal-intel with flavor baremetal.intel.25gb
[0:00:04] Attaching floating ip 10.151.15.210 to ClusDur-baremetal-intel
[0:09:17] Server in ACTIVE status
[0:09:22] Attempting connection to host.
[0:12:46] Cannot connect to host: [Errno 113] No route to host. Retrying 0 more times.
[0:12:51] Failed to connect to host.
[0:13:20] Deleting server ClusDur-baremetal-intel No server with a name or ID of '6e935f73-1c95-4442-8492-a15352f60b20' exists.
[0:13:26] Deleting server ClusDur-baremetal-intel
[0:13:31] Server ClusDur-baremetal-intel successfully deleted!
```

For the rest of the workflow, this will be handled via an Ansible playbook that will install all prerequisite libraries and packages, then use script tags to upload and execute bash scripts we have written for each individual benchmark. We did not have time to make this more modular, however ideally there would have been variables for different instance types to have finer control over which benchmarks are run.

Every benchmark script will write an output to the /home/centos/results folder, which is copied to the login node prior to instance deletion.

After the ansible playbook has executed, the script will finally delete the instance. A total time taken is output, with vm runs taking roughly 7.5 minutes and baremetal runs taking 15 minutes on both intel and AMD nodes, from instance reservation to complete deletion. Due to the over-reservation of instances, we were unable to test this workflow on a40 or a100 instances.

This workflow balances both performance of the node, as well as cost and carbon. With our VM runs on Intel, these only cost about 2.3p and only use about 7.5 minutes of compute on

some of the most green nodes (and about 4.6p on bare metal!). Therefore, we are still achieving high performance without destroying the environment or bankrupting alcesflight.

We also observed that baremetal provides improved performance but comes at nearly double the cost and carbon output, but its negligible given how short our workflow takes to execute. Therefore, we'd still go for baremetal to get the slightly improved performance! Baremetal only has a slightly higher initial time cost to create, and will still take the same amount or less time as a virtual machine for script execution. Over time, and for larger benchmarks, this further narrows the gap between baremetal and virtual machine.

Keyboard interrupts during script execution will be delayed, with the build cancelling and the instance deleted.