First Experiences with HPC as a Service

ABSTRACT

HPC in the Cloud or HPC as a Service (HPCaaS) is assuming increasing importance. However, it does have its own set of challenges. These challenges include choosing the right service provider from the many options available and then choosing the right hardware and software combination for the target application with a particular provider. We propose to tackle this problem from two angles. We first identify a set of representative HPCaaS applications to form a benchmark suite, HPCaas-Bench, to enable comparison among service providers and hardware/software combinations. We then use HPCaaS-Bench to show the impact of various hardware/software combinations on application performance and thus end-cost to user. We believe this study will serve as a useful guide for future users as they traverse the available options in HPCaaS space for their respective applications and also aid cloud providers to better equip or configure their clouds.

ACM Reference Format:

. 2023. First Experiences with HPC as a Service. In *Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '23).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

HPC in the Cloud or HPC as a Service (HPCaaS) is assuming increasing importance with its market size expected to reach 17 billion dollars in 2026 up from 6.28 billion dollars in 2018[1]. HPCaaS offers the opportunity for application developers and their users to take advantage of the latest hardware and software on state-of-the-art platforms, without the need to install or compete for access to large-scale on-premises supercomputing systems. Thus HPCaaS greatly improves both the cost and ease of use for HPC users while ensuring the desired great performance. These significant advantages are driving a shift in various important commercial HPC verticals (and their users) towards HPCaaS. These verticals include key industry segments such as finite element analysis (FEA), computational fluid dynamics (CFD), commercial weather, life sciences, financial services among others.

This shift from traditional on-premises HPC systems to HPCaaS, while promising to bring more users to HPC, has its own set of challenges. A major driver behind the rise of HPCaaS is the emergence of a number of Software as a Service (SaaS) solutions where a custom web-based interface allows users to easily and quickly submit their job and only pay for the resources (cores x time) used. For instance, for a job, the users may only have to specify the dataset for the software and size (i.e. number of cores/nodes) of job, and the SaaS provider may then run it on an Amazon or Microsoft or Google cloud and use any particular combination of core/node (any SKU from a list of AMD, Intel, ARM processors in the Cloud), compiler (Intel, Cray, gnu, amd, nvidia, arm), interconnect (Infiniband, Slingshot), MPI implementation (OpenMPI, cray-mpich, intelMPI, mvapich, anl-mpich) or even distribution of MPI ranks on nodes. While this approach is easy and opaque to user, the user may have to pay quite a bit extra in terms of core hours for any sub-optimal choice made by the service provider.

Even when the user uses the less opaque Platform as a Service (PaaS) solution, where the user can potentially build the application with specific software (compiler, MPI, etc.) and run on specific hardware with one service provider, the users have little help in determining the best (fastest, cheapest) PaaS provider among the many options for their specific application since different providers offer different hardware and software. For instance, Amazon offers its own cheaper ARM-based processor instances that are usually slower than traditional x86 instances from Intel and AMD that are standard with some other providers, but those may provide the best performance per dollar for a certain class of applications. From our own experiments discussed later in this work, we find that a change in just the MPI implementation used with the application can cause a 1.7x difference in application performance. In summary, there is no standard way for the users to pick the right service provider for their respective needs.

We propose to tackle the above challenge through two complementary solutions.

- In this many-provider scenario, the users have to be able to compare performance across providers and then come to the best choice to optimize performance per dollar (or other relevant criteria). For this purpose, we first identify five representative (and fastest growing) HPCaaS verticals and choose one open-source application/software in each of those verticals to comprise a benchmark suite, called HPCaaS-Bench. The representation across multiple verticals tries to capture the variety that is seen in real user workloads and could help users to choose the provider with the desired important features in their cloud based on observed performance on HPCaaS-Bench.
- We use this HPCaaS-Bench to further study the various choices that a cloud provider or user could make and how it impacts performance of workloads. These choices include the processor type, the compiler and related tools such as scientific libraries used, the particular combination of MPI implementation and interconnect technology, the file system used and finally the configuration of the runtime. We believe that this study could serve as a useful guide for users both in these verticals and otherwise to understand the impact of these choices and thus choose the best options and/or service provider for their respective workloads. This could also be used by cloud providers to better equip or configure their clouds for specific verticals.

SC '23, November 12–17, 2023, Denver, CO

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

HPC Vertical	Application	Dataset/Model
Finance Services Industry	Monte Carlo Options Pricing	Samples = 10000000
Life Sciences	NAMD	STMV (virus) benchmark; 1,066,628 atoms
Finite Element Analysis (FEA)	OpenRadioss	1996 Chrysler frontal crash model; 1M elements
Computational Fluid Dynamics (CFD)	OpenFOAM	motorBike 10M cells
Weather Forecasting Services	MPAS-A (NCAR)	120km mesh resolution; 40962 horizontal grid cells; 20-day forecast

Table 1: Applications in HPCaaS-Bench

We thus make the following contributions.

- We identify applications/software from fast-growing HPC verticals to form HPCaaS-Bench.
- We use applications from this suite to study the impact of specific hardware and software choices that an HPCaaS user could make on application performance. These choices specifically include various compilers, compiler options, compiler targets, node configurations, price vs performance, MPI implementations, interconnects and communication frameworks.
- We provide a list of best practices from this study that could both educate a user about making the best choices on a given platform and also in choosing the right platform for their application. Finally, we believe this study could also be useful for service providers to provide the right software and hardware combinations (and their respective configurations) to their users.

The rest of the paper is organized as follows. Section 2 lists the applications in HPCaaS and the describes the test environment for our study. Sections 3 through 7 take each of our 5 chosen applications and discuss (1) their relevance to HPCaaS, (2) their specific characteristics that influence their performance on the node or on the system, and (3) their sensitivity to specific combinations of hardware and software. Through this, we also highlight the importance of that aspect of hardware and software to overall system performance. Section 8 summarizes the lessons learnt and lists best practices for HPCaaS. The related work is discussed in Section 9 and Section 10 concludes this work.

2 APPLICATIONS AND ENVIRONMENT

Table 1 lists the five applications, one in each identified HPC vertical, chosen to comprise HPCaaS-Bench. The applications chosen are all open-source and also among the most popular options in the respective verticals.

For our experiments with these applications, we use HPE's development cloud (or devCloud) platform, which is a precursor to their first production cloud and is similar to HPE Cray EX "Everest" platform. We chose HPE's devCloud since it gives us the needed variety both in hardware and system software for our exploration. The devCloud has nodes of AMD Genoa (96 cores per socket, 2 sockets per node), Intel Sapphire Rapids (56 cores per socket, 2 sockets per node) and AMD Milan (64 cores per socket, 2 sockets per node). For network interconnect, we again evaluate two options - Infiniband HDR200 and Cray Slingshot. The devCloud only has the Cray Slingshot. For evaluation with Infiniband HDR200, we use another cluster with HPE Apollo 2000 Gen10 Plus hardware that has AMD Milan nodes (56 cores per socket, 2 sockets per node). Both the devCloud and HPE Apollo systems have the same variety of system software. In terms of system software, both the devCloud and HPE Apollo provide the Cray compiler (cce - version 15.0.1), the GNU compiler (gnu - version 12.2.0), the Intel compiler (intel - version 2021.4.0), the AMD compiler and PGI compiler although we restrict our evaluation to cce, gnu and intel compilers. For each of these compilers, we can further target specific processor targets such as rome, milan and genoa for AMD, and broadwell, skylake, cascadelake and sapphirerapids for Intel - each of these targets enforces different code schedulers, and more importantly, different target vector widths and vectorization cost models. For MPI implementations, we evaluate cray-mpich and OpenMPI, which is the most popular choice among users in the commercial HPCaaS space. Cray-mpich is the default MPI implementation of both the systems used; we use our own build of OpenMPI (version 4.1.4) to compare the two implementations. While Cray Slingshot is only compatible with ofi (OpenFabrics Interfaces) communication framework and thus the only choice on devCloud, we compare ofi with ucx (Unified Communication X) on the HPE Apollo system. Finally, we evaluate the impact of different distribution models in parallel jobs and the impact of using a parallel file system like lustre versus the traditional Network File System (NFS). We next discuss each of the applications one at a time and also share results from our experiments with each.

3 FINANCIAL SERVICES INDUSTRY - MONTE CARLO

Market risk management is critical to trading firms, i.e. how various combinations of market price movement affect their investments in various financial instruments[4]. Understanding this risk helps these firms to decide when to invest or divest in various instruments, which is crucial to the financial integrity of these important institutions and thus the capital markets. This task of risk analysis requires a firm to analyze various financial instruments that rely on other underlying instruments and changes in the overall market, and makes use of computationally intensive numerical methods that must run on datacenters or supercomputers. While some of the bigger firms usually have an on-prem datacenter or supercomputer, their enhanced compute and shorter turnaround time requirements during taxing times such as increased market volatility and more complex portfolios require them to use the latest and fastest processors that are readily and plentily available in the cloud. This industry is thus a key customer of the cloud.

Many financial instruments such as equities, bonds, interest rates, currencies, etc. incorporate options, where the number of

options contracts every year are tens of billions[4]. As a result, understanding options pricing is crucial to trading firms. The first famous model for options pricing was Black Scholes formula [12] that assumes that an asset's price is stochastic. While later models such as Heston[17] even assume that the volatility of options prices is also stochastic, it (like many models in finance) leads to a partial differential equation that only yields an exact solution in special cases. The Monte Carlo method[13] is a very popular alternative that runs a large number of random simulations, where each simulation is a potential path that an associated asset's price and volatility could take; an analysis of the outcome of all the paths then provides the overall most likely option price.

The Monte Carlo implementation we use for our evaluation comes from FinanceBench[16] and models pricing of a single option using QMB (Sobol) Monte-Carlo method. This application like other applications/software in this space relies on shared-memory parallelism and not distributed memory parallelism like most other applications in HPCaaS, and is unique in this respect. For shared-memory parallelism, we use OpenMP programming model in this case. The original code from FinanceBench uses rand() function for random value generation. However, since rand() does not accept a seed as an argument and instead uses a hidden seed value, each call to rand() modifies the seed. Thus, since rand() depends on state, the use of this function often enforces serial execution. We thus found the original version to not scale with threads. We thus use rand_r() that is thread-safe and allows excellent scaling with threads.

genoa	target=genoa 1-thread	target=genoa 192 threads	target=genoa 384 threads
	30.1s	172ms (175x)	182ms (165x)
spr	target=spr 1-thread	target=spr 112 threads	target=spr 224 threads
	28.3s	383ms (74x)	346ms (82x)

Table 2: Execution time of Monte-Carlo with different configurations

Table 2 shows the execution time observed when running Monte-Carlo on AMD genoa and Intel sapphire-rapids under different configurations. For compilation, we use the same Cray compiler with 'craype-x86-genoa' and 'craype-x86-spr' targets, respectively, to generate binaries for genoa and sapphire-rapids. The configurations tested include running with a single thread, as many threads as the number of cores in the node and as many threads as the number of threads (two threads per core) in the node. We make the following observations from the table.

- We first observe that Intel sapphire-rapids is 1.06x better than AMD genoa in single-thread performance. This is because sapphire-rapids uses the Intel golden-cove core that is bigger (has more out-of-order execution potential) than genoa's zen4 core.
- For multi-threaded runs, we find that while scaling is good for both genoa and sapphire-rapids and performance is not limited by memory bandwidth (since Monte-Carlo is largely compute bound), the performance does not scale perfectly in either case due to reduced clock speed observed under load. Genoa scales better than sapphire-rapids though since the clock rate drops more significantly for sapphire-rapids under load. This is because of multiple factors - sapphire-rapids uses bigger and thus more power hungry cores, has a lower TDP (350W vs 400W) than genoa and the use of an arguably less advanced and thus less power-efficient process (Intel 7 in sapphire-rapids vs TSMC 5nm in genoa).
- We find that for genoa, the best performance is with using 1 thread per core and using all the cores on the node. On the other hand, for sapphire-rapids, the best performance is seen when using both hyperthreads on all cores. This is again because each core on sapphire-rapids has bigger queues than genoa to support out-of-order execution. For instance, the reorder buffer on sapphire-rapids has 512 entries[14] versus genoa's 320[7] allowing for more instructions to execute out of order. The bigger queues result in fewer backend stalls, giving more cycles/slots to hyperthreads to do useful work.

While the above table compares performance across two most popular state-of-the-art processors, it is ultimately more important for the user to compare performance per dollar among available options in the cloud. As an example, we thus looked at the pricing of Oracle Compute Instances to find the better of these processors from a performance per dollar perspective. Table 3 uses the monthly price for a 'BM.Standard3.64' and a 'BM.Standard.E4.128' instance, which are, respectively, Intel Xeon Platinum 8358 (i.e. Icelake - immediate predecessor of sapphire-rapids with 76 cores) and AMD EPYC 7J13 (i.e. Milan - immediate predecessor of genoa with 128 cores). We further compute the price per core for each of these instances and then use that to calculate estimate price for sapphire-rapids and genoa instances with 112 and 192 cores, respectively. In this calculation, we make the assumption that relative price per core between AMD and Intel will stay the same across the latest two successive generations of server processors. Table 3 then combines the relative performance (we take the best performance for each processor, which is with 192 threads for genoa and 224 threads for sapphire-rapids) and the estimated relative price for both processors. We thus find that genoa scores 1 versus sapphire-rapids' 0.78, thus becoming the preferred choice for a user targeting better performance for a given price even though sapphire-rapids beats genoa in single-thread performance.

	Relative Performance	Relative Price	Performance per dollar
genoa	1	$\frac{\$BM.Standard.E4.128}{128}$ x 192 = $\frac{\$4666.37}{128}$ x 192 = $7000	1
spr	$\frac{162}{327}$ = 0.5	$\frac{\$BM.Standard3.64}{76}$ x 112 = $\frac{\$3047.42}{76}$ x 112 = $4491	$\frac{\$7000}{\$4491}$ x 0.5 = 0.78

Table 3: Performance per dollar comparison between sapphire-rapids and genoa on Monte-Carlo

4 LIFE SCIENCES INDUSTRY - NAMD

The life sciences industry comprises companies that perform research, development and manufacturing of pharmaceuticals, biotechnology-based food and medicines, medical devices, biomedical technologies,

food processing, and other products with the aim of improving the lives of organisms[2]. With ever emerging newer diseases and rising health challenges among population at large, this industry is experiencing steep growth. For instance, the global biotechnology market size was estimated at $449 billion in the year 2019 and is expected to be $727.1 billion by 2025, and the global pharmaceutical industry made an aggregate estimated business worth $1.3 trillion in the pre-covid year 2019[2].

Molecular biology is increasingly relevant to pharmaceutical sciences as it helps in the understanding of structures, functions, and internal controls within individual cells, which can then be used to efficiently target new drugs, diagnose disease, and better understand cell physiology. We thus chose namd as a representative in this space for our investigation since it is a very popular molecular dynamics simulation software and plays an important role in modern molecular biology[18]. namd is particularly designed for high performance simulations of large biomolecular systems on parallel computers. NAMD is written using the Charm++ parallel programming model for parallel efficiency.

For our test case with namd, we used the STMV (virus) benchmark with 1,066,628 atoms and is known to scale well even up to thousands of processors. Since it incurs low communication overhead, we use namd to study single node performance in multiple scenarios.

Processor = **genoa** 8 ranks x 12 threads	target=**genoa**		target=**rome**	
	with ivdep	**w/o** ivdep	**with** ivdep	**w/o** ivdep
	39.1s	39.5s	33.3s	26.9s
Processor = **spr** 8 ranks x 7 threads	target=**spr**		target=**broadwell**	
	with ivdep	**w/o** ivdep	**with** ivdep	**w/o** ivdep
	62.2s	62.9s	68.9s	55.9s

Table 4: Execution time of NAMD with different configurations

Table 4 shows the execution time observed when running NAMD on AMD genoa and Intel sapphire-rapids under different configurations. These configurations include using two different versions of the application - one which uses ivdep (i.e. a pragma that instructs the compiler to ignore vector dependences) to vectorize some of the important routines in NAMD and the other one which does not use ivdep. Once the vector dependences are ignored, the routines in question can be vectorized using gather/scatter operations, and both genoa and sapphire-rapids support the AVX-512 instruction set that has gather/scatter instructions. It is important to note that we have previously observed benefit from using ivdep on Intel Skylake and Cascadelake processors (not shown here in evaluation) which also support AVX-512 instructions. In addition to using these two different code versions, we also use two different targets for each processor - 'cray-x86-rome' and 'craype-x86-genoa' for genoa and 'craype-broadwell' and 'craype-x86-spr' for sapphire-rapids, where AMD rome and Intel broadwell are both precursors to AMD genoa and Intel sapphire-rapids, respectively, and both support AVX2 instruction set. The idea behind using older targets is that since they use AVX2 instructions (256-bit vector operations) instead of AVX512, they are more conservative when it comes to vectorization especially when gather/scatter operations are involved. This is because vectorization cost model in the compiler assumes less benefit from vectorization and overhead may outweigh benefit given slower gather/scatter implementation in respective older hardware. We make the following observations from the table.

- When using 'target=craype-x86-genoa' with the Cray compiler and running on genoa, we observe similar performance for both the versions of the code. We verify using optimization report from the compiler that the compiler does indeed vectorize the intended routine after the use of ivdep. In this case, however, the overhead from vectorizing with gather/scatter operations does not outweigh benefits from vectorization. One of the overheads associated with gather/scatter is that these operations introduce multiple contiguous loads/stores into the pipeline that can cause more stalls in the load/store queue and also be sometimes less optimal than having various loads and other operations interspersed.
- When using 'target=craype-x86-spr' with the Cray compiler and running on sapphire-rapids, we make similar observation as above - both code versions perform similar. Overall, genoa performs better than sapphire-rapids given a higher core count per socket (96 vs 56).
- When using 'target=craype-x86-rome' and running on genoa, we here find that the code version without ivdep performs much better (1.24x) than with ivdep. This is because 'target=craype-x86-rome' uses 256-bit vectors and thus the benefits of vectorization are less as compared to AVX-512 and thus overshadowed by the overhead. More importantly, however, targeting 'rome' and running on genoa is much more performant as compared to targeting 'genoa' on genoa both with and without ivdep on certain important routines. This is because NAMD in general makes a lot of irregular memory references and vectorization is often possible using gather/scatter operations that are not always performant. Using 'target=rome' restricts such non-beneficial vectorization deeming it expensive in its cost model, resulting in better performance. It is important to note that in this case 'target=craype-x86-rome' is up to 1.5x better than 'target=craype-x86-genoa'.
- When using 'target=craype-broadwell' and running on sapphire-rapids, we make a similar observation as above where 'target=craype-broadwell' and not using ivdep is the most performance configuration and beats 'target=craype-x86-spr' by as much as 1.13x. However, we find that the code version with ivdep is slower with 'target=craype-broadwell' than with 'target=craype-x86-spr' although 'target=craype-x86-rome' was better than 'target=craype-x86-genoa' with ivdep - this could be due to genoa actually using two 256-bit vector units in hardware versus sapphire-rapids using two 512-bit vector units making the former more suitable for AVX2 instructions.
- Overall, we find that the best performing version on genoa ('target=craype-x86-rome' without ivdep) is 2.1x better than the best performing version on sapphire-rapids ('target=craype-x86-broadwell' without ivdep). It is important to note that while genoa has more cores (96 vs 56) than sapphire-rapids, the latter does have a bigger core. However, we observe that in this case, genoa runs at a clock rate of 3.19GHz while sapphire-rapids runs at a clock rate of 2.47GHz and since

NAMD is compute-bound, the performance is directly proportional to clock rate. The higher clock rate combined with higher core count together lead to the 2.1x speedup for genoa.

In the above discussion, we compared a particular combination of MPI ranks and OpenMP threads (or pthreads) for both genoa and sapphire-rapids. NAMD, however, like many other applications can be run with various combinations of ranks and threads such that all or most of the cores available on the node are utilized. Often, it is observed in HPC that running an application with just MPI (and no OpenMP threading) is more performant - this is because MPI tends to parallelize the entirety of an application's code with ranks splitting work right at the start whereas if certain loops cannot be parallelized (across threads) because of dependencies, then those loops have to execute serially. For instance, HPCG scales effectively with MPI but not with OpenMP since one of the two most time consuming loops has a loop-carried dependence and cannot be parallelized with OpenMP resulting in poor scaling performance. However, when all important loops/routines can be parallelized with OpenMP, the best configuration (ranks vs threads) usually depends on the overhead incurred from each of the parallelization strategies. NAMD is still more unique as it uses 1 thread per MPI rank dedicated to just communication which is unlike the usual case where the thread involved in communication also does other work. As a result, we find that while adding ranks is helpful to NAMD, adding ranks greater than 8 begins to degrade aggregate performance as shown in Table 5. We observe that the best performing configuration (4 x 24) is as much as 1.22x more performant than the worst performing configuration (24 x 4), and clearly, the best performance does not come from using the maximum possible number of ranks or threads per node.

Ranks x Threads	# Worker Threads	# Communication Threads	Execution Time
2 x 48	94	2	26.3s
4 x 24	92	4	26.0s
8 x 12	88	8	26.7s
16 x 6	80	16	28.8s
24 x 4	72	24	31.7s

Table 5: Execution time of NAMD with different number of ranks and threads

5 FINITE ELEMENT ANALYSIS - OPENRADIOSS

Altair Radioss is a multidisciplinary finite element solver that helps users evaluate and optimize product performance for linear as well as highly nonlinear problems under dynamic loadings. Various organizations have used Altair Radioss to replace costly tests related to crash, safety, manufacturability, durability, fluid-structure interaction, etc. with quick and efficient simulation. This not only reduces cost, but also helps organizations bring their products to market faster.

OpenRadioss is the publicly available open-source version of Radioss that was very recently released (late 2022). It is now increasingly being used and enhanced by developers and researchers in industry and academia. This is empowering users to tackle challenges associated with rapidly evolving technologies such as autonomous driving and battery development while also improving traditional use cases such as automotive crash and safety, shock and impact analysis, electronic and consumer goods drop testing, and fluid structure interactions[5]. To enable fast multiphysics simulations, OpenRadioss makes use of both OpenMP and MPI parallel structure, leading to industry-leading scalability regarding large, highly nonlinear structural and multiphysics simulations.

Regarding the model itself, we use the NEON FE (1M11) benchmark model that simulates a frontal crash of a 1996 Chrysler Neon traveling at 50km/h colliding with a rigid wall. This analysis simulates the performance of the Neon (with refined mesh) through the first 80ms of the crash event. The complete model includes approximately 1 million elements and is considered a medium-large size model by 2008 standards but is considered small today.

OpenRadioss is communication intensive and we thus use it as the testbed to compare multiple aspects of communication that affect performance such as the interconnect (Slingshot[SS] vs Infiniband[IB]), MPI implementation (Cray-mpich vs OpenMPI) and communication framework (ofi vs ucx). OpenRadioss is also IO intensive with the application reading from and writing to as many files as the number of MPI ranks. For IO performance, we compare NFS4 and Lustre file systems.

Table 6 shows the execution time of all the configurations that we tested. As mentioned earlier, while we have Slingshot on devCloud, we use Infiniband on the HPE Apollo system. For parity, we use 112 out of the available 128 cores of Milan on each node in devCloud while we use all all 112 cores of Milan on each node in HPE Apollo. Also again for consistency, we scale up to 392 x 4 (i.e. 14 nodes) on both systems since we only have 14 nodes of Milan on the HPE Apollo system. We make the following observations from the table.

- First of all, we make the observation across all columns that performance does not scale perfectly with the number of ranks/nodes. This indicates that OpenRadioss is indeed communication heavy. For most configurations, the performance begins to dwindle or even degrade as we increase the number of nodes beyond 8.
- We also similarly find that performance scaling is better with lustre file system as opposed to nfs4 file system. This is because OpenRadioss is IO intensive and thus benefits from the lustre parallel file system that provides considerable better IO bandwidth/throughput as compared to the NFS4 file system used otherwise.
- Cray-mpich vs OpenMPI on Slingshot: For this purpose, we compare column 1 vs 2, and column 3 vs 4. On the NFS4 file system, OpenMPI starts witnessing performance degradation beyond 4 nodes, whereas Cray-mpich continues to show some scaling until 8 nodes. Alongside lustre file system, we find that Cray-mpich experiences even better scaling with decent scaling upto 8 nodes whereas OpenMPI still almost stops scaling at 4 nodes with Cray-mpich outperforming OpenMPI by 1.6x and 1.7x at 8 and 14 nodes, respectively. This is primarily because of extensive tuning in Cray-mpich for Slingshot given that Slingshot is Cray/HPE's own proprietary interconnect.

Ranks x Threads	File system = nfs4		File system = lustre				
	SS+CrayMpich [Column 1]	SS+openMPI [Column 2]	SS+CrayMpich [Column 3]	SS+openMPI [Column 4]	IB+CrayMpich+ucx [Column 5]	IB+openMPI+ucx [Column 6]	IB+CrayMpich+ofi [Column 7]
28 x 4 (1 node)	1403s	1438s	1417s	1437s	1644s	2014s	1728s
56 x 4 (2 nodes)	618s	669s	606s	669s	777s	984s	778s
112 x 4 (4 nodes)	367s	389s	328s	351s	445s	623s	446s
224 x 4 (8 nodes)	335s	435s	232s	337s	363s	446s	380s
392 x 4 (14 nodes)	345s	543s	212s	357s	351s	381s	395s

Table 6: Execution time of OpenRadioss with different configurations

- Cray-mpich vs OpenMPI on Infiniband: We compare column 5 vs 6 for this purpose as the communication framework is ucx in both of these cases. We first observe that absolute performance of OpenMPI is lower especially at smaller node counts. This is because we had to use OpenMPI's mpirun to run parallel jobs on the HPE Apollo system given the absence of pmix library that is otherwise needed to run with Slurm's srun runtime. As a result, we could not benefit from srun's effective binding of ranks and threads to cores. With mpirun, we use '–rankfile' option to emulate srun's binding strategy, which was the most performant, but the ranks still could not be precisely pinned to a single core and instead rotated across 4 cores, which perhaps leads to some degradation on the HPE Apollo system. However, we find that Cray-mpich and OpenMPI otherwise scale similarly on Infiniband.
- Slingshot vs Infiniband: We compare column 3 vs 5, and columns 4 vs 6 for this purpose. We find that Slingshot and Infiniband are similar in performance with OpenMPI, which is significant given that OpenMPI is not optimized with Slingshot. On the other hand, Slingshot significantly outperforms Infiniband with Cray-mpich given their mutually beneficial designs.
- Finally, we compare the ucx and ofi communication frameworks on Infiniband using columns 5 and 7. Note that Slingshot is currently only compatible with ofi. We find that Cray-Mpich scales better with ucx (which is recommended for Infiniband by Mellanox) as ofi begins to loose performance when going from 8 to 14 nodes.

6 COMPUTATIONAL FLUID DYNAMICS - OPENFOAM

Computational fluid dynamics is a branch of fluid mechanics that uses numerical analysis and data structures to visualize the effect of a gas or liquid on the object it flows past. CFD is used by a variety of other industries such as automobile, aerospace and even sports equipment manufacturers to reduce the drag and friction from air and thus improve speed and efficiency of their products. The CFD market is expected to have a considerable growth rate of 8.1% during 2023-2028[8] and reach $3.7 billion mainly owing to rapid advancements in the aerospace and aeronautics industries.

In this space, we chose OpenFOAM to be a representative CFD software since,

- It is one of the top three CFD packages world wide given its significant user base.
- It comprehensively covers a wide range of engineering and scientific problems involving fluid flows, heat exchange and chemical reactions.
- OpenFOAM is a C++ open source toolbox for the development of customized numerical solvers, and pre-/post-processing utilities for the solution of continuum mechanics problems, most prominently including computational fluid dynamics.

The motorBike 10M cell model that we study is an OpenFOAM tutorial case. It is an incompressible steady state flow around a motorbike and rider. The model is constructed by creating a blockMesh which passed to the OpenFOAM snappyHexMesh to further refine the mesh resulting in approximately 10M cells. The model is then decomposed into a number of sub-domains. Each domain is then allocated to a parallel process to solve using the OpenFOAM potentialFoam and simpleFoam solvers executable models. Results are then post processed and analysed using visualisation tools such as paraview.

In our experiments, we ran OpenFOAM with up to 16 nodes of Milan in devCloud. While OpenFOAM is IO intensive, we find that it does scale very well up to the 16 nodes that we used. We thus use OpenFOAM to test various binding and distribution policies. A binding policy determines how MPI ranks are bound to resources within the node - these resources include sockets, cores and threads within a node. On the other hand, a distribution policy determines how the MPI tasks are distributed among nodes, and then among sockets within a node. Table 7 shows the execution time of OpenFOAM with various binding and distribution policies. These include 'none' (default), 'threads', 'cores', 'sockets' and 'auto' binding policies and 'block:block', 'block:cyclic' (default), 'cyclic:block' and 'cyclic:cyclic' distribution policies, where the first distribution method (before the :) controls the distribution of tasks to nodes and the second distribution method controls the distribution of tasks across sockets in the node. We first test with different binding policies while using the default distribution policy, and then test the different distribution strategies with the 'auto' binding policy, which is also the best performing binding policy among those tested. It is important to note that while there is no binding policy called 'auto', we refer to the policy that is chosen by default if we set cpu-bind to 'verbose' as 'auto'. We make the following observations from the table.

- We first observe that the default binding policy is clearly sub-optimal as it leads to considerably worse performance compared to other binding policies such as 'cores' and 'auto'. This is because the workload manager in this case does not bind ranks to any resources, and it may thus happen during execution that certain tasks are bound to the same core

Binding	Distribution	Ranks = 1 x 128	Ranks = 2 x 128	Ranks = 4 x 128	Ranks = 8 x 128	Ranks = 16 x 128
default (none)	default (block:cyclic)	607s	280s	118s	52s	28s
threads	default (block:cyclic)	1211s	528s	222s	99s	46s
cores	default (block:cyclic)	516s	216s	91s	46s	25s
sockets	default (block:cyclic)	575s	252s	111s	55s	29s
auto	default (block:cyclic)	510s	210s	85s	41s	23.4s
auto	block:block	511s	209s	85s	41s	22.7s
auto	block:cyclic	510s	210s	85s	41s	23.4s
auto	cyclic:block	510s	211s	87s	42s	25s
auto	cyclic:cyclic	509s	211s	87s	43s	25s

Table 7: Execution time of OpenFOAM with different bindings and distributions

(overloading the core), while some cores are underutilized, leading to sub-optimal performance.

- The 'core' and 'auto' binding policies perform considerably better than others. This is because the former binds tasks to individual cores (one thread) per core and the latter binds a task to the threads (two in this case) of a core - in both cases, since the number of tasks per node are equal to the number of cores on the node, this results in optimal binding and optimal performance. These policies are especially performant when the number of tasks per node equal the available cores on the node. Also, in this case, these are equivalent to the more sophisticated task to cpu binding made possible by the 'map_cpu' binding where the user can specify specific cpu ids to which tasks must be bound.
- The 'threads' and 'sockets' policies aim to bind tasks to individual threads and sockets, respectively. In the former case, we find 2 tasks bound to the same core, one per thread. This leaves half of the cores over-provisioned and half under-provisioned since the number of tasks per node are less than the number of threads, leading to sub-optimal performance. In the latter scenario, the tasks are bound to either of the 2 sockets per node, and again may lead to some tasks bound to the same core during execution leading to worse performance compared to other strategies.
- Among the different distribution policies, we do not find the same fluctuation in performance as with binding policies. Overall, 'block:block' performs the best while 'cyclic:cyclic' performs the worst. This is because the former policy distributes tasks such that consecutive tasks share a node; at the node, consecutive tasks are distributed on the same socket before using the next consecutive socket. This leads to consecutive tasks be as proximal as possible and since tasks usually communicate with immediate neighbors, 'block:block' usually performs best. The 'cyclic:cyclic' policy on the other hand, distribute consecutive tasks to consecutive nodes in a round-robin fashion; within the node, tasks are then distributed in round-robin fashion between the sockets. This usually increases communication overhead since neighboring tasks reside on different sockets and nodes.

7 WEATHER FORECASTING SERVICES - MPAS

Weather forecasting services is another increasingly important industry given the increasing need for weather solutions across industries. There is increasing demand to deploy sophisticated models and generate highly detailed forecasts that are critical for the decision-making process in a wide range of industries such as energy, transportation, agriculture, and emergency management. Weather forecasting applications are very complex mathematical and physics-based models that require a large amount of processing power and memory that has been traditionally provided by on-prem supercomputers. However, more recently, some of the commercial weather forecasting services are increasingly migrating to the cloud to use the latest hardware on demand.

In this space, we use the Model for Prediction Across Scales (MPAS), a cutting-edge weather model that uses a unique approach to simulate atmospheric processes at different scales. MPAS-Atmosphere (MPAS-A) is a fully compressible and non-hydrostatic dynamics model with the following features: split-explicit Runge-Kutta time integration, exact conservation of dry-air mass and scalar mass, positive-definite and monotonic transport options, generalized terrain-following height coordinate, and a support for unstructured variable-resolution mesh integrations for the sphere and Cartesian planes.

We ran the mode at 120-km mesh resolution (40962 horizontal grid cells) for 20-day forecast with the mesoscale reference suit physics. The "mesoscale_reference" suite includes several physics parameterizations, including:

- Morrison-Gettelman microphysics scheme
- Kain-Fritsch cumulus parameterization scheme
- RRTMG shortwave and longwave radiation schemes

These parameterizations are designed to capture the complex interactions between atmospheric processes like radiation, convection, and cloud formation that are critical for accurate mesoscale weather prediction.

We compiled MPAS with different compilers using the makefile provided with the application. The makefile contains build instructions with cray, gnu, intel, pgi and other compilers. We built and tested with cray, gnu and intel compilers. First, while the build instructions (i.e. compiler options) with the intel compiler worked as is, we had to add additional compiler options with the cray and gnu compilers. With the intel compiler, we used '-convert big_endian

Compilers	-O3	-Ofast
gnu	452s	-
intel	446s	424s
cray	441s	412s

Table 8: Execution time of MPAS with different compilers and compiler options

-FR -real-size 64' with -O3 or -Ofast. The 'convert' option allows the conversion of unformatted data between little- and big-endian representation to facilitate moving of data between different systems. With the cray compiler, we used '-default64 -f free -h byteswapio' with -O3 or -Ofast where we had to manually insert '-h byteswapio' option that performs the same functionality as 'convert' for the intel compiler. With gnu compiler, we used '-fdefault-real-8 -fdefault-double-8 -m64 -ffree-line-length-none -fconvert=big-endian -ffree-form -fallow-argument-mismatch -fallow-invalid-boz' where we had to add the last two options to convert certain warnings turned errors in the latest gnu compiler back to being warnings to build successfully. With gnu, when building with -Ofast, the application fails since -Ofast enables -ffast-math, and the attempted aggressive optimization causes a segmentation fault. As a result, -O3 is the best option with gnu compiler.

Table 8 shows the execution time with different compilers and different compiler options. We observe that while all compilers perform similarly with -O3, the cray compiler outperforms the intel compiler by 3% with -Ofast and gnu by 10%. The better performance with Ofast usually stems from reduced accuracy in floating point calculations, which is often acceptable especially with the accompanying performance boost. In this case, using -Ofast with gnu leads to an application crash whereas that is not the case with cray and intel compilers given their relative maturity with HPC applications and fast math.

8 SUMMARY OF LESSONS LEARNT AND IMPLICATIONS

We provide a summary of lessons learned from all the above explorations. We also discuss other implications in various cases so that a user can apply these lessons in various practical situations.

(1) In the HPC world, a processor cannot be judged by single-thread performance. It is important to evaluate a processor by its aggregate throughput, i.e. with all its cores/threads fully utilized. While this has always been important to HPC, it is particularly relevant at this given time as processor's aggregate throughput (especially for compute bound applications) is largely governed by its power budget (or TDP) and is no longer a function of single-thread performance and number of threads. The limited power budget throttles core's clock frequencies in ways that are different across processor vendors (especially Intel vs AMD since they use considerably different process technologies for manufacturing processors).
(2) In the HPCaaS world, a processor cannot even be judged by its aggregate throughput like in the HPC world. This is because cost of service as a function of usage/time is introduced. As a result, a processor with higher aggregate throughput may do so with a higher core/thread count, which may drive up its cost. It may also have a high TDP, requiring better cooling solutions that amounts to both a higher upfront cost and a higher cost of operation, and thus be priced at a premium versus a less powerful solution. In such a case, the best option may be the one with the best performance per dollar not necessarily the one with the best aggregate performance, especially if there is interest in reducing cost for the user. This situation may particularly be applicable as newer ARM-based servers arrive in the market to rival x86-based servers where the former often provision lower power and aim for energy efficiency. Taking a step further and looking at the ongoing energy crisis especially in some parts of the world, one could also envision users being charged based on power consumption, in which case performance per watt could become the deciding criteria.
(3) Vectorizing the code can be a bittersweet experience especially with gathers/scatters. We are not in an era of custom HPC processors where vectorization almost often led to improved performance. Gathers and scatters can quickly and aggressively fill load/store queue and also hinder effective memory to compute operation scheduling leading to reduced performance especially when the benefits from vectorizing the computation in the loop is limited. As a result, while vectorization with gather/scatter can improve performance, there could also be a marked degradation. As a result, it may be better to use an older target such as AVX2 instead of AVX512 that restricts vectorization with gathers/scatters. It is no surprise, therefore, that the Intel compiler defaults to AVX2 even on AVX512 hosts, and processors continue to tune processors for AVX2 even though supporting AVX512 instruction set.
(4) The best ranks versus threads configuration to use is a function of various factors such as scope of parallelization (whole application versus certain routines) with OpenMP and MPI, associated overheads (time/memory) of parallelization and the chosen parallelization strategy (for example, reserving a thread for communication as in Charm++ versus doing useful work with it). While parallelizing with just MPI (no threads) is often the most performant in HPC given incomplete application coverage with OpenMP/thread parallelism, multiple HPCaaS applications perform best with a hybrid strategy than with pure MPI.
(5) The choice of the MPI implementation and interconnect combination can be the single most important factor to determine system performance for a given application especially when the application is communication intensive or is run at a very large scale. It is best to choose the MPI implementation that is optimized for a particular hardware or vice-versa, i.e. choose the interconnect for which an optimized MPI implementation exists or is available. Similarly, it is advisable to also choose the recommended communication fabric with a particular MPI implementation and interconnect. Finally,

application performance could also be affected by the choice of runtime (e.g., srun vs mpirun), especially with hybrid ranks x threads, as different runtimes bind ranks and threads differently. It is better to use runtimes that are aware of the node configuration (in terms of cores per numa node, numa nodes per socket, sockets per node, etc.)

(6) Applications that are IO intensive benefit significantly from a parallel file system such as lustre, and not using one could significantly hurt performance.

(7) There are several options to choose from both for the binding and the distribution strategy. Clearly, the default choices are far from optimal, especially for the binding strategy. While the best binding and distribution strategy is largely a function of tasks/threads allocated per node and partially influenced by communication patterns, we found that 'auto' binding policy combined with 'block:block' distribution policy usually gives great performance when the number of tasks/threads per node is the same as number of cores per node.

(8) Finally, application performance depends on the choice of compiler used and the respective compiler options used. Given the continued development of applications (to conform to new language standards or use new language features), and also the continued evolution of compilers, the compiler options needed to successfully compiler the applications change and the compiler may no longer be even able to successfully compile or run. Being able to work with multiple compilers on the platform is useful in such a scenario.

9 RELATED WORK

We classify related work into two categories.

Benchmarks for HPC. SPEChpc 2021 [3] is a fairly recent HPC benchmark suite that is different from prior releases, SPEC MPI 2007 and SPEC OMP 2012, in that it can run in a hybrid MPI and OpenMP setting as opposed to just MPI or OpenMP. This is clearly more representative of real HPC and HPCaaS workloads. However, there are still significant differences between our proposed HPCaaS-Bench and SPEChpc 2021. First, the SPEChpc 2021 benchmarks are very small with all of them except one under 18000 lines of code. The miniWeather code for instance is only 1100 lines. The smaller codes usually imply a few important routines and thus easier to optimize for the user/compiler. In contrast, the codes in HPCaaS-Bench are significantly bigger with rarely few dominant functions. Second, SPEChpc 2021 comprises benchmarks that were submitted by the community to SPEC and those that are easily compilable by different compilers and not picked considering industry trends. There is for instance no representative for life sciences or finite element analysis/crash simulation - 2 key areas for HPCaaS. HPCaaS-Bench, on the other hand, comprises benchmarks particularly relevant for the growing HPCaaS market and the benchmarks are such that they are hard to compile and push compilers to the limit. Finally, it is unclear if they capture the behavior of HPCaaS applications, which could be quite communication and IO intensive, and use novel MPI libraries such as Charm++.

Another benchmark sometimes used in HPC is the HPC Challenge Benchmark[15]. It consists of 7 benchmarks or tests that each measures a different aspect of the system. For instance, the popular HPL benchmark primarily measures the system's peak floating point performance, and the stream benchmark measures the system's aggregate bandwidth. In big applications often modeling different physics such as those in HPCaaS, the same application may be bound by peak floating point performance or memory bandwidth in different phases, and it is thus difficult to reason about its performance on any given system by looking at the performance of these benchmarks on that system. Another popular benchmark suite is the NAS Parallel Benchmark Suite[9], which has both kernels like HPC Challenge and full applications. However, the applications are all derived from CFD and do not have the same variety as HPCaaS-Bench. Finally, there is the Parsec benchmark suite[11], but it only focuses on shared memory programs for chip multiprocessors.

Hardware/Software Exploration. There is publicly available information around the various options one could use when launching parallel jobs such as with a workload manager like slurm[6] including the binding and distribution strategies. However, there is little guidance on which options give the best performance in a given situation. In the of binding strategy, for instance, it is hard to know that the default strategy would give considerably bad performance versus other strategies based on public information. Among other work, a recent paper[10] compares scaling performance of ofi versus ucx for Slingshot, but in general, there is limited work around comparing interconnects and MPI implementations, or compilers.

10 CONCLUSION AND FUTURE WORK

In this work, we focus on addressing the problem that end users of HPCaaS currently face when it comes to choosing among the many available service providers. These providers may be offering software/application as a service or platform as a service. In either case, the user is faced with many combinations of hardware and software options to use with the application. To solve this problem, we first identify five key verticals and applications that should capture a wide user base in HPCaaS space to form HPCaaS-Bench. We then experiment with various hardware/software combinations ranging from compilers, compiler options, compiler targets, MPI implementations, interconnects, communication frameworks, node configurations and price performance using specific applications from HPCaaS-Bench.

From this exploration of various hardware and software choices that are available to users, we identify eight lessons or best practices and summarize their implications. We believe this could help other users in HPCaaS and serve as a guide as they decide on a particular service provider or service by that provider. We also hope to release and expand HPCaaS-Bench with a few more applications such as seismic reservoir and electronic design automation which could become a standard way for users to compare performance across service providers. Finally, we also plan to include a few popular AI workloads that are also increasingly facing HPCaaS adoption.

REFERENCES

[1] 2020. HPC as a Service Market. https://www.alliedmarketresearch.com/high-performance-computing-as-a-service-market

[2] 2021. Life Sciencies Glossary. https://www.scilife.io/glossary/life-science

[3] 2021. SPEChpc 2021 Benchmark Suites. https://www.spec.org/hpc2021/

[4] 2021. STAC-A2 report. https://stacresearch.com/a2

[5] 2022. OpenRadioss. https://www.openradioss.org/

[6] 2022. Slurm Workload Manager. https://slurm.schedmd.com/sbatch.html

[7] 2023. AMD Zen 4, url = "https://www.guru3d.com/news-story/amd-zen-4-die-cache-sizeslatencies-and-transistor-counts-detailed.html".

[8] 2023. Computational Fluid Dynamics Market: Global Trends. https://www.imarcgroup.com/computational-fluid-dynamics-market

[9] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. 1991. The Nas Parallel Benchmarks. *The International Journal of Supercomputing Applications* 5, 3 (1991), 63–73. https://doi.org/10.1177/109434209100500306 arXiv:https://doi.org/10.1177/109434209100500306

[10] M. Bareford, D. Henty, W. Lucas, and A. Turner. 2022. OpenFabrics and UCX. *Cray User Group* (2022).

[11] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[12] Fischer Black and Myron Scholes. 1973. The Pricing of Options and Corporate Liabilities. *Journal of Political Economy* 81, 3 (1973), 637–654. http://www.jstor.org/stable/1831029

[13] René Carmona, Pierre Del Moral, Peng Hu, and Nadia Oudjane. 2012. An Introduction to Particle Methods with Financial Applications. In *Numerical Methods in Finance*, René A. Carmona, Pierre Del Moral, Peng Hu, and Nadia Oudjane (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–49.

[14] ChipsandCheese. 2022. The Golden Cove microarchitecture. https://chipsandcheese.com/2021/12/02/popping-the-hood-on-golden-cove/

[15] Jack Dongarra and Piotr Luszczek. 2011. *HPC Challenge Benchmark*. Springer US, Boston, MA, 844–850. https://doi.org/10.1007/978-0-387-09766-4_156

[16] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. 2013. Accelerating Financial Applications on the GPU. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* (Houston, Texas, USA) *(GPGPU-6)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/2458523.2458536

[17] Steven L. Heston. 2015. A Closed-Form Solution for Options with Stochastic Volatility with Applications to Bond and Currency Options. *The Review of Financial Studies* 6, 2 (04 2015), 327–343. https://doi.org/10.1093/rfs/6.2.327 arXiv:https://academic.oup.com/rfs/article-pdf/6/2/327/24417457/060327.pdf

[18] Mark T. Nelson, William Humphrey, Attila Gursoy, Andrew Dalke, Laxmikant V. Kalé, Robert D. Skeel, and Klaus Schulten. 1996. NAMD: a Parallel, Object-Oriented Molecular Dynamics Program. *The International Journal of Supercomputer Applications and High Performance Computing* 10, 4 (1996), 251–268. https://doi.org/10.1177/109434209601000401 arXiv:https://doi.org/10.1177/109434209601000401