Programming tools for Analogue Quantum Computing in the High-Performance Computing Context – A Review

Mateusz Meller[1,2], Vendel Szeremi[1], and Oliver Thomson Brown[2]

[1]Hartree Centre, Science and Technology Facilities Council (STFC), UK Research and Innovation (UKRI)
[2]EPCC, The University of Edinburgh, UK

Recent advances in quantum computing have brought us closer to realizing the potential of this transformative technology. While significant strides have been made in quantum error correction, many challenges persist, particularly in the realm of noise and scalability. Analogue quantum computing schemes, such as Analogue Hamiltonian Simulation and Quantum Annealing, offer a promising approach to address these limitations. By operating at a higher level of abstraction, these schemes can simplify the development of large-scale quantum algorithms. To fully harness the power of quantum computers, they must be seamlessly integrated with traditional high-performance computing (HPC) systems. While substantial research has focused on the integration of circuit-based quantum computers with HPC, the integration of analogue quantum computers remains relatively unexplored. This paper aims to bridge this gap by contributing in the following way:

Comprehensive Survey: We conduct a comprehensive survey of existing quantum software tools with analogue capabilities.

Readiness Assessment: We introduce a classification and rating system to assess the readiness of these tools for HPC integration.

Gap Identification and Recommendations: We identify critical gaps in the landscape of analogue quantum programming models and propose actionable recommendations for future research and development.

1 Introduction

Quantum computing (QC) offers a fundamentally different computational approach with the potential to revolutionise fields such as cryptography, quantum chemistry, and material science [1]. Major vendors are currently focused on scaling gate-based quantum computers [2–5]. However, fault tolerance is essential to unlocking the full potential of these machines, and while significant progress has been made, substantial challenges remain [6].

Analogue quantum computing offers a potential solution to some of the challenges gate-based quantum computers face. Recent research [7–10] suggests that these machines are promising candidates for addressing a range of practical problems, including Quantum Hamiltonian Simulation (QHS) [11] and combinatorial optimisation [12, 13]. However, due to their relatively lower popularity, there is a shortage of software tools which could fully leverage potential of analogue quantum computing [8].

Regardless of type, quantum devices will not replace classical computers. Instead, they are seen as hardware accelerators, similar to GPUs, capable of boosting the computational power of high-performance computing (HPC) systems [14]. For instance, quantum computers could accelerate quantum chemistry problems in the strong-correlation regime while leaving the weakly correlated interactions to classical methods [15]. While the search for specific applications is ongoing, it is clear that quantum computing will require close integration with classical processors, presenting challenges at both hardware and software levels to become effective.

In this work, we focus on the software side of things. More precisely, we review existing programming models and tools that target analogue quantum computation and compile to analogue devices. Furthermore, we assess their readiness for HPC-QC integration. Our preliminary research shows that this space is underexplored compared to gate-based quantum computing. As such, there is a need to develop new programming models and frameworks to fully exploit analogue quantum devices.

The rest of the paper is organised as follows: In section 2, we motivate our work and give the necessary background. Next, in section 3, we cover related reviews on analogue quantum computing, quantum programming models and HPC-QC integration. Then, section 4 outlines a taxonomy and methodology for our analysis. In section 5, we collect existing quantum software frameworks supporting analogue computation. We perform a detailed analysis of their suitability for HPC environments. We close with a discussion of our results in section 6 and an outlook for future work in section 7.

2 Background

The following section outlines the background for this work. We begin by defining high-performance computing and quantum computing. Then, we introduce the categories of analogue quantum computing, highlighting their potential advantages. Finally, we explore the concept of programming models, their importance, and the motivation for studying them in the context of quantum computing.

2.1 High-Performance Computing

The objective of High-Performance Computing (HPC) is to solve problems that are intractable for standard desktop machines. It is generally achieved by partitioning the studied problem and performing massively parallel computation. For years, the dominant approach to realise this was connecting many commodity processors via networks to create large-scale distributed systems. To realise data transfer, the dominant communication protocol is Message-Passing Interface (MPI) [16]. As such, any software tool hoping for high usage in the HPC domain must be amenable to MPI to be adopted [17].

However, since 2004, a new paradigm has taken hold where modern HPC systems become more complex and built of specialised devices – so-called accelerators. One such accelerator is a general-purpose graphical processing unit (GPGPU) [18]. Indeed, in 2018, 40% of systems in the Top500 list [19] were accelerated predominantly with GPUs [17]. As of today, most new systems at the top of the list are accelerated. We note that accelerators are not new in the HPC space – in the early 1980s ICL Distributed Array Processors were installed in the physics department at the University of Edinburgh, and the programming model and challenges were remarkably similar to that of modern GPU architectures [20, 21]. The total dominance of accelerated architectures however, is a recent development.

Given the variety of available hardware, a significant challenge is the portability of

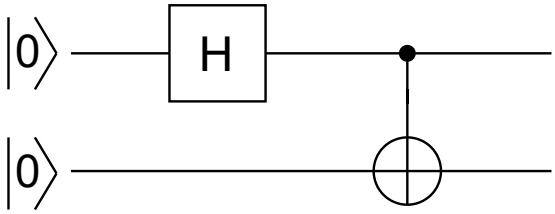

Figure 1: Quantum circuit preparing Bell state from zero state. Lines represent wires or qubits; square with H denotes Hadamard gate which puts qubit in superposition; followed by two-qubit controlled NOT gate. A quantum circuit diagram is a temporal description, unlike a classical electronic circuit diagram, which is spatial.

the software - making the same code run efficiently on different devices. Moreover, as the complexity of supercomputers grows and emerging accelerator technologies enter the market, the problem will become even more severe [17].

The main application of HPC is in the fields related to scientific computing, where the problems studied have computational-intensive characteristics.

In recent years, interpreted languages such as Python have become popular in scientific computing. The reason being their flexibility and increased developer productivity. However, for the most part, the dominant large-scale codes running on supercomputing systems are compiled, as native code has better performance characteristics [22] and compilation offers an opportunity to detect and correct errors ahead of runtime. In practice, those codes should be considered the leading candidates for adding quantum routines and using quantum computers at the industrial scale.

Summarising, the grand challenges faced by the HPC community are *clarity*, *productivity*, *portability* and *performance*. There is a lot of legacy baggage, and most practical quantum computing will need to be integrated with existing code rather than focusing solely on developing new applications. Therefore, adhering to the constraints of the HPC field is crucial for emerging technologies and software tools.

2.2 Quantum Computing

The fundamental concept of quantum computing is that of a quantum bit, or *qubit* for short. A qubit is a system living in two-dimensional Hilbert space, which, similar to a classical bit, has two possible states labelled $|0\rangle$ and $|1\rangle$. Contrary to the classical picture, a qubit can be in any linear combination (or superposition) of those two states. Then the state of a single qubit system is described by,

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \alpha, \beta \in \mathbb{C}.$$

For the system of n qubits, the corresponding Hilbert space is created by taking the Kronecker tensor product of the components, producing space that has dimension 2^n. The system's state is a unit vector embedded in this vector space. As the problem size grows exponentially, the general simulation of an n-qubit system is prohibitively expensive for classical computers.

Qubits in superposition of states can interfere, producing creative or destructive interference patterns. Upon measurement, by leveraging interference, desired states can be amplified while suppressing the others. This phenomenon is another source of the promised power coming from quantum computing.

Furthermore, qubits can be *entangled*. Entanglement is a concept without classical analogue, where the qubits are correlated with each other and cannot be represented as a

direct product of single-qubit states. This results in operations on one qubit affecting all others with which it is entangled.

In quantum mechanical formalism, the quantum system's state evolves in time as a unitary transformation. The unitary transformation ensures that the state is normalised. In the standard circuit-based model of quantum computation, unitary transformations are building blocks of the program and are referred to as *quantum gates*. Quantum circuits have a visual representation (see fig. 1), which simplifies reasoning about the quantum algorithms.

The quantum complexity advantage is achieved by combining the superposition, entanglement and constructive/destructive interference of qubits.

2.3 Analogue QC

Fundamentally, any real-world implementation of a quantum computer is an analogue machine. For example, in neutral-atom Rydberg-interaction machines, the qubit is implemented by a neutral atom, with the two distinct quantum states being the atom's energy levels. Then, operations applied to qubits are performed by the inherently continuous laser pulses. Continuing with this example machine, we achieve a digital or gate-based quantum computer by specifically tuning or calibrating the allowed pulses, yielding a discrete basis gate set.

On the other hand, by allowing ourselves to control the device arbitrarily with continuously parameterised pulses, we stay at the analogue level. Operation on the analogue level provides an advantage by giving access to operations native to the device. By taking fewer steps, we can achieve the same results with higher efficiency in specific scenarios. Analogue QC can be roughly split into three categories: Analogue Hamiltonian Simulation (AHS), Adiabatic Quantum Computing (AQC) and Quantum Annealing (QA).

In the literature on analogue quantum computing, there is also a discussion of analogue pulses. While serving as a common interface to programming the quantum device (e.g. Bloqade [23] or Pulser[24]), many packages in the field implement either limited Hamiltonians or are not flexible enough for arbitrary AHS (for example, Qiskit Pulse [25] in general lacked global addressing to realise fully analogue system evolution). In cases where limited analogue pulses serve as a reinforcing technique to standard gate-based quantum computing, these are referred to as semi-analogue quantum programming [9].

2.4 Analogue Hamiltonian Simulation

Given the many-body Hamiltonian $H = \sum H_{i,j}$, where $H_{i,j}$ describes the local interactions between system components i, j, we want to simulate the system evolution for time t given by e^{-iHt}. The standard method to approximate this evolution in the gate-based formalism is via the Lie-Trotter product formula [26], a process often referred to as *Trotterization*. The error induced by the use of Trotterization is inversely proportional to the Trotter time-step dt.

If we want to implement this approximation using a circuit model with a realistic, fixed gate set, we face a few challenges. As summarised in [27], we can generally assume that implementing an arbitrary term in the Trotterized product requires at least one two-qubit gate. Two-qubit gates generally take longer to physically implement and this overhead places a lower bound on the achievable Trotter time-step dt regarding the number of operations required and the device's capabilities to switch on and off required controls. Also, for smaller dt, it is hard to implement gates with high fidelity, further increasing errors in the computation.

However, rather than implementing this evolution with quantum circuits, we can directly program Hamiltonians of the analogue quantum computer (another term used in literature for the quantum device in this specific use case is *analogue quantum simulator*) and synthesise desired target Hamiltonians [10]. One avoids overheads associated with the gate-based implementation and implements Hamiltonian evolution with a shorter pulse duration. Moreover, expensive interactions in gate-based formalism (multi-qubit gates) can be applied globally [11].

Continuing with the previous example, interaction modelling Trotterized term is implemented natively and executed for time dt. Therefore, if we compare circuit-based runtime to Hamiltonian-oriented analogue runtime, the simulation takes $O(t/dt)$ vs $O(t)$ respectively [27]. Based on the previous discussion, the scheme achieves better time-to-solution. Moreover, one can expect slower error buildup from the number of operations used [27].

The natural limitation of this approach is that the underlying Hamiltonian of the studied system must map to the device Hamiltonian (i.e. Hamiltonian, which describes the possible evolution of the qubits controlled by the device) [11]. The problem of mapping a device Hamiltonian to an arbitrary problem Hamiltonians is studied by the optimal control community. For general Hamiltonians, it is a hard problem [10]. This challenge implies that doing analogue Hamiltonian Simulation is a highly specialised problem.

One might fairly point out that the analogue computation has significant noise and error resilience difficulties. This bottleneck is true both in the classical and quantum case. Nevertheless, in the Noisy Intermediate-Scale Quantum (NISQ) era [28], where noise and error rates are limiting parts of QC, and devices suffer from a lack of scarce resources for overheads caused by error correction, the centre of the stage is taken by the algorithms that are resilient to noise, such as variational quantum eigensolver (VQE) [29]. In those cases, we can get away with using analogue devices [27].

Indeed, a growing body of science shows the usefulness of performing pulse-level quantum computation (provided the pulses can be sufficiently well controlled) not only in Quantum Hamiltonian Simulation [11] already mentioned but also for Quantum Approximate Optimization Algorithm (QAOA) [12], VQE [30], and machine learning (ML) [31].

From the programming perspective, Hamiltonian Simulation is of interest, as the natural formalism of building operators is high-level. One can employ Hamiltonian Modeling Languages (HML), which have a low barrier to entry.

2.5 Adiabatic Quantum Computing

A specific form of AHS, called Adiabatic Quantum Computing (AQC) is based on the Adiabatic Theorem, which roughly states that if we start in the eigenstate (e.g. ground state) of a given time-dependent Hamiltonian $H(t)$ during the evolution under the Schrödinger equation, the system will stay in the instantaneous Hamiltonian eigenstate, provided the change in $H(t)$ is "slow enough" [32].

Formally, using the definition from Aharonov [33] and the definition of k-local Hamiltonian, i.e. Hamiltonian, which is a sum of Hamiltonians acting non-trivially on at most k particles:

Adiabatic Quantum Computation is defined by two k-local Hamiltonians, H_0, H_1, which act on n particles with $m \geq 2$ states. The ground state of H_0 is a non-degenerate product state. The output is given by defining a schedule function $s(t) : [0, t_{run}] \to [0, 1]$, and t_{run} such that the final state of adiabatic evolution of the $H(s) = (1 - s)H_0 + sH_1$ is ε-close in Euclidean norm to the ground state of H_1. The algorithm implementing AQC is known as a Quantum Adiabatic Algorithm (QAA).

The history of adiabatic quantum computing starts with an established classical algorithm - Simulated Annealing (SA) [34]. In 1988, it was established that it is possible to use ground states of quantum Hamiltonians to encode solutions to combinatorial problems [35]. Shortly after, Apolloni et al. [36] proposed implementing a quantum-inspired algorithm called Quantum Annealing (QA), analogous to simulated annealing. The initial idea behind QA was that instead of thermal fluctuations (used in SA), simulated quantum fluctuations and simulated quantum tunnelling were means to solve combinatorial optimisation problems. In the optimisation space landscape, with many local minima, thanks to quantum tunnelling, QA, in principle, can penetrate hills better to find the optimal solution. In those early days, QA was also known as quantum stochastic optimisation (as introduced in the original article [35]) whereas now, the original QA executed on a classical devices is referred to as Simulated Quantum Annealing (SQA) for clarity.

In parallel, starting from 1999, another concept was evolving. While early work on quantum computing was defined in a quantum circuit model, mainly through Deutsch and Penrose [37], who introduced it, Farhi et al. [38, 39] realised that by leveraging the adiabatic theorem, one could map problems to the Hamiltonians and perform computation. Soon after, it was shown that AQC is a universal model of quantum computation, equivalent to circuit model [33]. Moreover, there exist routines which map between those two models, with polynomial overhead only [32]. What followed was the re-implementation of seminal quantum algorithms to their adiabatic form. Examples include: Grover's search [40], the Deutsch-Jozsa algorithm [41, 42] and prime factorisation [43]

While initially, QA was a classical algorithm, with the progress in quantum computing and information, the experimental story for QA is slightly different. Based on the developments in the control of quantum systems and theoretical understanding of AQC, Brooke et al. [44, 45] realised experimentally a quantum annealer, i.e. a device which implements QA algorithm on a quantum hardware which approximately performs adiabatic quantum computation. Approximately is a key word here, as most AQC theory deals with closed quantum systems - systems isolated from the environment. In practice, such systems do not exist. After all, there is always the presence of noise due to thermal and electromagnetic fluctuations [46]. Moreover, quantum annealers implement 'stoquastic'[1] AQC [32]. That is, the Hamiltonians that are being evolved are stoquastic, which means the off-diagonal terms in the computational basis are non-positive. Hamiltonians of this form do not suffer from a sign problem and are easier to solve[2] [47]. As such, general AQC is different from real-world stoquastic AQC implementations modelled with transverse-fields Ising Hamiltonians [32]. Indeed, while AQC is a universal model of computation, stoquastic AQC is not [32]. This is the root of many controversies surrounding QA hardware and its prospective advantage over classical algorithms [48, 32, 49, 46].

Due to the distinction between AQC and stoquastic AQC and the intended use case, stoquastic AQC (StoqAQC) is also known simply as Quantum Annealing in literature [32, 46]. For the rest of this review, we will use terms interchangeably, favouring the term QA when describing work done on real-world quantum annealers.

2.5.1 Applications

Adiabatic quantum computing, being a universal model of computation, has a similar set of applications as circuit QC. As such, standard quantum algorithms are implementable

[1]The term 'stoquastic' comes from a combination of stochastic and quantum.

[2]Indeed, problems encoded in stoquastic Hamiltonians can be efficiently solved by quantum Monte Carlo methods.

in this formalism. From the QA point of view, the main application is that of combinatorial optimisation. Devices provided by D-Wave allow for solving Binary Quadratic Models (BQM) in the form of the Ising Model and Quadratic Unconstrained Binary Optimisation (QUBO) problems [46]. It is generally possible to map many vital problems to this representation. Notable examples include Maximum Independent Set (MIS) problems [50] or satisfiability (SAT) problems [51]. Still, the search for useful applications continues and remains one of the significant challenges of quantum computing.

2.5.2 Advantages Over Circuit Model

Given existing work on the quantum circuit model and its many advantages, a fair question to ask is: "Why bother with AQC and QA?" Setting aside pure scientific interest in trying things differently, there are practical reasons for considering those concepts, especially in the era of NISQ devices, where there is a strong drive to demonstrate practical quantum advantage.

The first clear advantage over gate-based systems is the scaling. To date, the pioneering QA vendor, D-Wave, offers quantum annealer chips with over 5000 qubits. While those have limited connectivity, the scale is beyond any other technology, bringing us closer to the regime where a clear advantage over classical methods can be shown. Indeed, recently, King et al. [52] have demonstrated quantum advantage in a simulation of nonequilibrium dynamics of a magnetic spin system quenched through a quantum phase transition.

Another advantage from the perspective of this work is that the programming model for quantum annealers is at a higher level than the circuit model. Although it might be argued that QA is programmed closer to the hardware with use of pulses and qubit topologies, in AQC and AHS, in general, one can reason about the problem at the level of Hamiltonian terms, breaking away from the high granularity of gate-based paradigm. In fact, Ocean SDK [53] offered by D-Wave for solving QUBO problems is strikingly similar to classical software used for combinatorial optimisation problems. Such an interface, in turn, makes it easier to integrate quantum annealers as accelerators within existing code bases. Lowering barriers to technology adoption increases the likelihood we can tackle practical problems earlier. Moreover, both the natural applications for QA and the programming model fit well into the hybrid HPC-QC integration model, where it is relatively easy to partition problems and offload to different devices (GPU, CPU or QPU). Indeed, recently, D-Wave introduced a hybrid solver which decomposes the encoded problem and solves it on their hybrid quantum-classical cluster [54].

One promising avenue is parallel quantum annealing [49]. Exploratory work by Pelofske et al. [55] has shown that it is possible to solve many independent problems using QA during a single annealing process. The main observation was that while there was a slight drop in accuracy, the speed-up was considerable exhibiting good scaling for the studied problem. From the perspective of HPC, any parallelisation strategy on emerging technologies is of interest.

The final consideration simply is that given many different quantum hardware technologies, each with its advantages and disadvantages, and without a clear winner, it is possible that the future HPC-QC integrated systems will host different, highly specialised QPUs. In a way, this would push the existing heterogeneous computing trend to the limit. While it would create its own set of challenges for interoperability and programming, it is certainly a feasible scenario for which we need to prepare.

2.5.3 Limitations

From the AQC point of view, given their equivalency, the limitations are shared with the quantum circuit model and QC in general. The additional limitation is that even though equivalent, the mapping between representations creates polynomial overhead. For certain quantum algorithms, this is a bottleneck for any advantage over classical algorithms. A notable example is a Grover search, which, in its circuit form, provides a quadratic complexity advantage. So, even if the mapping is possible, it might not be practical to map circuit-based algorithms to adiabatic form [32].

Another issue is finding a suitable evolution time or schedule. The fundamental condition for AQC and QA is that the change in $H(t)$ is slow enough. "Slow enough" is bounded and depends on the assumptions about the Hamiltonian. In the worst scenario, the time evolution t scales as $O(1/\Delta^3)$ – for the most general Hamiltonian [56] and $O(log(\Delta)/\Delta^2)$ – for bounded, infinitely differentiable Hamiltonian [57], where Δ is minimal energy gap in the energy (eigenvalue) spectrum or simply spectral gap [32]. While this provides scaling estimations, for general Hamiltonians, estimating Δ is a hard problem. Therefore, selecting a good evolution time is a challenge and a significant limitation of the method. A possible solution is to estimate it experimentally. However, at a scale beyond classical computation, this might lead to unverifiable solutions and, in specific scenarios, suppress the quantum advantage [32].

Due to similar reasons corresponding to the approximation of spectral gap Δ, it is hard to analyse the complexity of AQC/QA algorithms and their time-to-solution. In the quantum circuit model, the algorithm's cost is its depth or number of gates. While one could use the depth of the circuit needed to simulate an adiabatic algorithm on a gate-based device, this would assume that the circuit model is fundamental [32]; that would also skew the results as it would include polynomial overheads due to mapping between models, which would be prohibitive for detailed analysis required in the field of HPC. Another option is to use runtime t_{run}. However, it requires an appropriate energy scale of the Hamiltonian to be meaningful. Using the definition of AQC with schedule functions (outlined before), Aharanov [33] introduced the concept of cost for the adiabatic algorithm as

$$cost = t_{run} \max_s ||H(s)||,$$

where $||\circ||$ is the operator norm. The *cost* is a dimensionless quantity, which protects from making the cost of the algorithm arbitrarily small by multiplying Hamiltonian by some size-dependent factor. In this formalism, t_{run} can be compared to an equivalent circuit-based simulation of an adiabatic algorithm and *cost* can be analogous to circuit gate count [32].

As outlined by Albash and Lidar [32], for a limited set of algorithms where spectral gap analysis can be performed, it was shown that AQC provides an advantage over classical algorithms. Those examples mainly refer to seminal quantum algorithms. Usually, however, the spectral gap analysis does not show any advantage, the results are inconclusive, or no information is available. Indeed, in many cases, the gap during Hamiltonian evolution becomes exponentially small with the size of the system and vanishes at the phase transition, causing $t_{run} \to \infty$.

Strikingly, there are no advantage guarantees for the original application of QA – that of combinatorial optimisation problems [32]. Quantum computing for combinatorial optimisation is a very active area of research due to commercial interest and availability of QA hardware. Currently, the promise is that thanks to quantum tunnelling, quantum annealers can succeed in finding minima in glassy optimisation landscapes [52]. Moreover, it

was shown only recently that quantum effects such as superposition and quantum entanglement, considered the source of quantum advantage, are present in QA devices at scale [58].

Whereas noise mitigation and error correction are a primary focus in the field of gate-based quantum computing, they are somewhat limited in QA [46]. However, it is clear that the impact of the noise is severe. For example, D-Wave devices are affected significantly by the environment when the runtime is longer than a couple of microseconds [49]. It is still an open question if and how to optimally mitigate errors on analogue quantum annealers [46].

While for years, the standard platform for annealers was super-conducting devices, today, trapped-ions and neutral-atoms hold much promise [46]. Software tools for those platforms are discussed in the following sections.

Overall, AQC and QA are fields with many promises but also many challenges. For interested readers, we refer to extensive reviews [32, 46].

2.6 Integrating Quantum Computers with HPC

It is established that to leverage quantum computers fully, those machines must be tightly integrated with HPC systems.

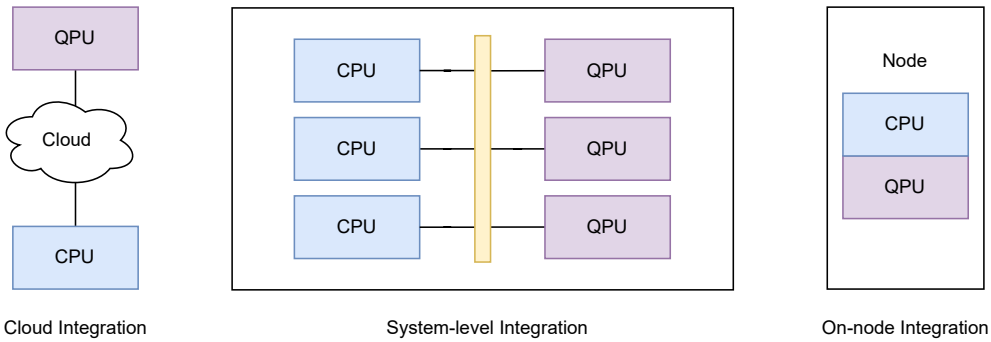

Figure 2: Proposed forms of HPC-Quantum integration. From left to right subfigures represent tighter integration with lower latency at the cost of technical challenges.

How exactly such integration will look is still an open problem. Current proposed forms of integration (fig. 2) can be roughly split into three categories.

First, there is a loose integration where logically classical and quantum systems are co-located; however, physically, access to quantum computers is given through a cloud interface. Although the easiest to realise, this setup has many bottlenecks from the HPC point of view. For many supercomputing systems, compute nodes have no access to the internet for security reasons. Moreover, offloading to the physically distant device penalises the workflow by large communication latency, which can be a bottleneck for hybrid quantum-classical algorithms.

The next step towards tighter integration is if we co-locate classical and quantum computers at the system level, within physical proximity and connect with a shared network. From the point of view of HPC, system-level integration offers dramatic growth in performance and logical coherence for quantum-accelerated scientific workflows. Nevertheless, it is significantly harder to realise it. In practice, the main challenge for such a setup would be interoperability and support for many different controls and functionalities.

Finally, we achieve the closest coupling between classical and quantum processors by performing on-node integration. It should be noted that given the sensitivity to noise of the

quantum devices, on-chip integration is unrealistic. In practice, this level of integration assumes a logical hardware model where classical host shares classical information with a classical co-processor which has direct access to quantum hardware. Aside from the engineering challenges of shielding the QPU, there are additional challenges related to the synchronisation of data between host and device (common to GPU and other accelerator programming). Nevertheless, at this stage, the latency is negligible, and existing codes can be seamlessly offloaded to the quantum part, potentially resulting in significant speed-ups.

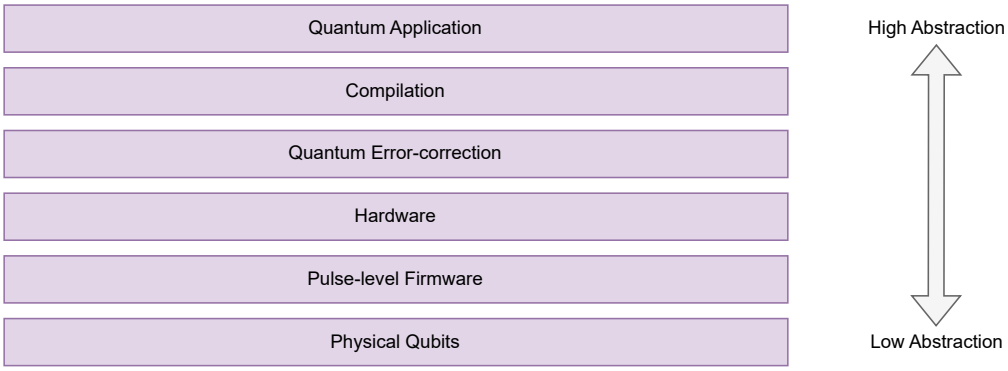

Figure 3: Layers of a quantum software stack. From top to bottom, layers proceed from the higher abstraction levels to lower, starting with quantum application layer and ending with physical qubits of the target platform.

On a software level, to integrate quantum computers within the HPC ecosystem, a coordinating layer needs to be introduced which allocates and schedules computing resources for the task at hand. In general, the quantum software stack consists of multiple layers (as shown in fig. 3): *application layer, compilation layer, quantum error-correction layer, hardware layer, pulse-level firmware layer* and finally *physical qubits*.

2.6.1 Application layer

From an HPC point of view, the role of the quantum application layer is to first implement a quantum algorithm that solves the target problem and, second, interact with established classical HPC software, for example, in a hybrid workflow. This constrains new quantum programming tools in that they need to interact with classical tools.

2.6.2 Compilation layer

The domain-specific application layer with high abstractions is lowered to the hardware at the compilation layer. Given the current state of things (i.e., many different platforms), compilers must simultaneously be hardware agnostic and hardware aware. Indeed, this opens room for the use of different intermediate representation (IR) dialects for different tasks [59–63], such as general translation, general circuit optimisation, followed by hardware-specific gate optimisation [64]. Elsharkawy et al. [65] stress the need for standardisation in this space, and indeed, there are many attempts to do so (OpenQASM 3.0 [66] and QIR [67] to name two).

2.6.3 Error-correction layer

The next layer in the stack is responsible for the error correction. For a detailed introduction and overview of the state-of-the-art in this domain, we refer to references [68, 69, 6].

In the current age of NISQ devices, this layer is mostly absent. However, many groups work on experimental realisation of fault-tolerance and subsequent implementation of error-correcting software. From the HPC perspective, the error-correction schemes are expected to require non-negligible amounts of classical compute power [70, 71]. This not only puts an additional constraint on the hardware integration efforts, but any error-correcting software also needs to be able to interface with classical programming models and HPC environment. From a programming model perspective the distinction is whether the user implicitly allocates and operates on logical or physical qubits. Logical qubits are assumed to be error corrected, and the programming model may elide the details of the error correction routine. On the other hand the user may choose to work with physical qubits, implementing their own error correction.

2.6.4 Hardware layer

At the hardware layer, the operating system environment allocates classical and quantum resources to execute the compiled code. Again, in the HPC setting, this layer has to interface with classical resource management systems (for example, SLURM [72]), as it is responsible for the proper transpilation and mapping of the code onto the device. Moreover, this mapping has to be dynamic because more than one job can be scheduled onto a given device. Moreover, during the execution, the hardware layer is responsible for acquiring and communicating the results of mid-circuit measurements, which are needed for error correction and some quantum algorithms.

2.6.5 Pulse layer

From the software point of view, the final pulse control layer is responsible for executing the compiled quantum program according to the constraints of the actual quantum device. Here, the device processes operations based on its Instruction Set Architecture (ISA). For the most part, this layer is specific to the backend and hidden from the standard quantum software user. However, with evidence that such fine-grained control can be beneficial in the NISQ era [73], some software providers decided to introduce pulse-level control [25, 66, 74, 75]. In the space of analogue quantum computing, on the other hand, this is the layer with which the user has to interact, creating a barrier to entry for users without a specific platform background.

To date, most work on integrating HPC-QC systems, especially on the software level, has been focused on the quantum circuit model. Aside from the work by D-Wave on their hybrid solver, the backend details of which are private, the space of analogue quantum computing is unexplored in this regard.

Summarising, the expected architecture within the HPC system is that QPU will be an accelerator attached to a heterogeneous node (fig. 4). As a matter of fact, given the specialisations and strengths of different quantum platforms, it is plausible that the actual architecture will resemble something like the one presented in fig. 5. Moreover, across the software stack, there is a need for tight integration between quantum device components and classical software at every stage.

To tackle real-world challenges, frameworks for programming quantum devices must support at least classical and, preferably, quantum communication. However, at the current stage, quantum communication between compute platforms is beyond our reach, and as such, we don't consider it in this work. For a review of quantum communication techniques and progress, we refer to reference [76].

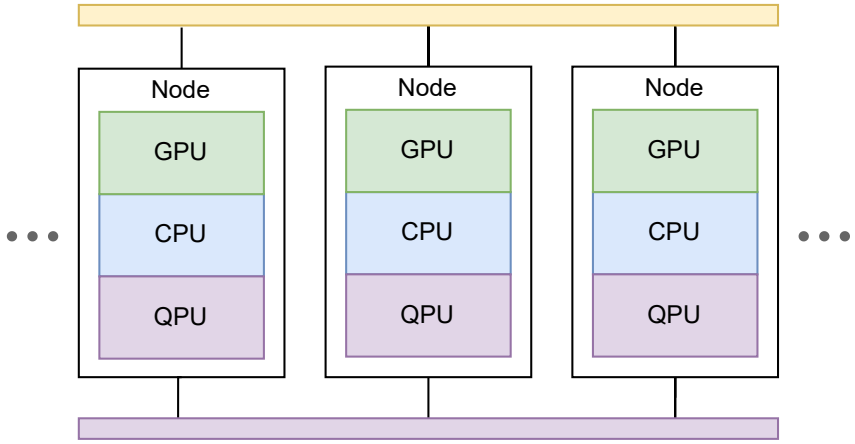

Figure 4: Example HPC architecture with integrated QPU. Stack of GPU, CPU and QPU represents a computing node; Yellow lines connecting nodes are classical interconnects; Purple lines connecting nodes are quantum interconnects used for quantum communication.

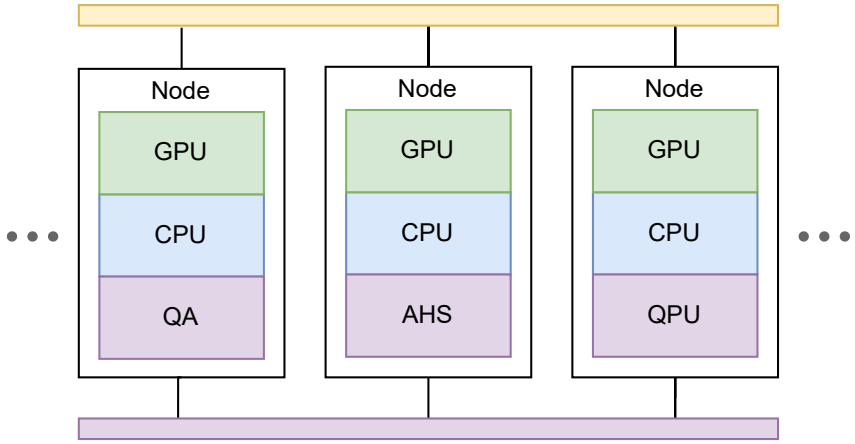

Figure 5: Example of a more specialised heterogeneous HPC architecture with different types of QPUs. All connections retain meaning from figure 4. QA refers to Quantum Annealer; AHS to Analogue Hamiltonian Simulator; QPU to gate-based quantum machine.

2.7 Programming Models

The concept of a *programming models* is heavily used in the field of HPC [77]. There, it is usually known as *parallel programming model* to hint that it refers to supercomputing parallel systems. But what is a programming model? For the purpose of this paper, based on [78], we define it as an abstraction of hardware architecture and memory. The programming model creates a framework which allows us to reason about the implementations of algorithms and their place in a program. In a way, the programming model allows for standardising how software communicates with hardware. This approach has a twofold advantage:

1. Software frameworks can implement existing programming models to provide constructs for common use cases on a given hardware.

2. Hardware producers can develop machines supporting a given programming model. This leads to better software portability.

As such, programming models have a great impact on new hardware and software. It

is crucial to have efficient, powerful, and high-level programming models for any new technology. Below, we outline motivation from the classical computing point of view and quantum computing point of view, respectively, to better illustrate the importance of developing programming models.

2.7.1 Programming models in classical computation

As the traditional HPC requires efficient resource use, there have been many developments in the programming models space over the years, in tandem with hardware development [17]. The two most common parallel programming models in HPC are the *Shared-memory* model and the *Distributed-memory* model. In the shared-memory model, all the workers have a uniform view of memory and can read and write to it freely. In the distributed-memory model, workers have a local memory pool and need to communicate with other workers via networks. Other popular models are the *Kernel-based* or *Host-Device* model, for example, employed by CUDA [79] to offload work to GPUs, and the *Hybrid* model, where approaches from remaining models are intermixed for maximal performance.

Besides the hardware advances, it can be argued that the significant breakthroughs and continuous scaling can be attributed to the use and development of appropriate programming models for the task, software, and hardware at hand. Seeing the impact of those on classical computation, it is clear that similar work needs to be done for quantum computing to fully enable the technology and fulfil its promises in a practical setting.

Designing good programming models is difficult without a clear, optimal solution. Programming model creators need to balance three major factors. First, the programming model needs to be flexible enough to allow for the development of novel scientific applications. Second, it should be relatively easy so new users can quickly become productive. Finally, it should be constrained enough to penalise bad design and produce performant solutions.

2.7.2 Programming models in quantum computation

In the current age of NISQ devices, the main quest is showing quantum advantage. Although several groups have demonstrated quantum supremacy in recent years [80, 52], the focus has shifted to practical applications [81]. At the research stage, the quantum programming models' requirements differ from those of their classical counterparts. As an emerging technology in the early stages the emphasis is put on low-level control so the experimenters can explore all the device's capabilities. Furthermore, performance considerations could be set aside in return for increased productivity and ease of usage for the scientists.

Nevertheless, it is desirable that when the transition towards the software engineering field happens, this transition will be as painless as possible. Therefore, it is important to consider performance, ease of use for non-domain experts, and interoperability with other classical programming models. After all, we need to consider that with high probability, quantum chemistry on a quantum computer code will run next to, or from within, Fortran code.

Therefore, the objectives for both existing and new quantum programming models should encompass:

- Support for existing hardware - so the model can be quickly adopted.
- Multiple levels of abstraction - so that domain experts can experiment and have high control over the device but also new users can quickly become productive

- Performance considerations for existing and future hardware (including classical processors), especially in spaces known to be bottlenecks. For example, a performant programming model could focus on limiting data movement, limiting communication overheads for future distributed systems or including support for hybrid programming models.

Given those objectives, there is motivation, both from classical and quantum points of view, to develop existing and new programming models.

3 Related Work

Several existing reviews cover three subfields of quantum computing: Adiabatic Quantum Computing, HPC-QC integration, and Quantum Programming Models. Most of those target their domain solely, however, and to the best of the author's knowledge, there is no work analysing analogue QC programming models and existing software from the HPC-QC integration point of view.

Gill et al. [1] perform a detailed comparative analysis of existing quantum software tools. By providing taxonomy and an extensive literature review, they outline existing challenges and gaps in knowledge that need to be overcome to realise quantum advantage. Additionally, they consider future directions and futuristic applications such as post-quantum cryptography.

Similarly, Serrano et al. [82] analyse existing software and hardware platform capabilities. They look at the quantum software development frameworks, compilers, simulators and error-correction tools. Resch and Karpuzcu [83] provide an overview of quantum computing technology, requirements to enter the "quantum-era", and turning to requirements for scalability and future outlooks.

Fingerhuth et al. [84] analysed open-source software in quantum computing, looking at the documentation, support channels, availability of tutorials, number of contributions and code readability metrics, among others.

Works by Heim et al. [85] and by Garhwal et al. [86] provide extensive review of existing programming languages targeting quantum computation, while Chong et al. [87] outlines requirements of a quantum compiler design and capabilities of existing compilers.

In the Adiabatic Quantum Computing and Quantum Annealing fields, Albash and Lidar [32] provide an extensive introduction to and review of AQC. Rajak et al. [49] cover Quantum Annealing. In [88], authors look at the experimental realisation of QA and its real-world usage. They highlight methods for achieving large-scale QA, motivating future research. In analogue quantum computation, Daley et al. [11] outlined the prospective source of quantum advantage in Quantum Hamiltonian Simulation. Clinton et al. [27] cover existing and promising algorithms that could be implemented with pulses on near-term hardware. Ella et al. [89] cover quantum programming at pulse level, quantum-classical processing and possible benchmarks for quantum controllers.

Regarding HPC-QC integration, Humble et al. [14] give perspectives on how QC-accelerated HPC systems could look. Recent work by Elsharkawy et al. [65] provides an extensive review of quantum frameworks, creating a blueprint for assessing their suitability for meaningful integration with classical supercomputing systems. Their work serves as a direct inspiration for this manuscript.

A review by Au-Yeung et al. [90] provides an excellent summary of the challenges of enhancing HPC with QC from the application point of view. They introduce and analyse thoroughly existing quantum algorithms, discussing examples from quantum chemistry electronic structure problem and computational fluid dynamics (CFD). Importantly, the

review considers all existing paradigms of quantum computing, including digital and analogue approaches. Finally, the paper identifies the challenges of interfacing quantum codes within the HPC context. They conclude that the main obstacles requiring addressing are mismatched clock speeds between the host and device, quantum data en-/de-coding, and data interconnections (with associated overheads).

A recent paper by Kaya et al. [91] contributed an important step towards HPC-QC integration. The group introduced the Munich Quantum Software Stack (MQSS) as a framework for integrating disaggregated quantum accelerators within the HPC system. Their work focuses on compilation, job scheduling and optimisation in an HPC-centric way. As a part of the stack, they introduce a Quantum device management interface (QDMI), which allows adding any type and number of QPUs. Additionally, Quantum Resource Manager (QRM) handles target agnostic/specific optimisation, physical fitting onto the device and scheduling. Overall, the paper fills many gaps in the field.

In a similar spirit, Beck et al. [92] provide a summary of work and efforts over the past five years on the topic of HPC-QC integration. They introduce another hardware-agnostic framework for such integration, realised as a suite of software tools for resource management, offloading, and hybrid quantum-classical application deployment.

Mohseni et al. [93] released an extensive review outlining challenges, outlooks and proposed solutions to build a quantum supercomputer. From the HPC integration point of view, some of the highlights include a new quantum API as an extension to HPE Cray Programming Environment (CPE), which supports languages typical to HPC (C, C++, Fortran); A plugin for Slurm that allows for quantum resource management; and finally, adaptive circuit knitting for better, distributed scaling.

Those recent, extensive reviews highlight that the field shifted to a serious effort of integrating quantum computers with HPC systems. However, neither paper looks at nor discusses in detail analogue quantum computing (QDMI, in principle, allows the addition of analogue quantum devices to the environment. However, the programming tools are predominantly gate-based).

4 Software Rating Criteria

When considering the existing software frameworks and tools, the distinctions in employed programming models are more nuanced than simple categorisation. Because of the inherently different nature of quantum computation, programming models were an essential part of theoretical and practical development from the beginning of quantum computing technology [86]. Over the years, many programming libraries and frameworks have arisen, sometimes with great variety [82]. As such, there are many different ways to classify existing solutions. Indeed, other than focusing on the abstractions of the memory, hardware, and software executed, we can consider things like level of abstraction, employed programming paradigm and more. Below, we outline additional categories used for framework assessment in this paper, thus creating a taxonomy for the following analysis.

Similarly to Elsharkawy et al. [65], we assess the viability of the quantum software from the HPC-QC integration point of view; as a result there is an overlap with the categories selected for our analysis.

4.1 Language Characteristics

The central factors that influence programming models and programming languages alike are their programming paradigm, syntax, and semantics. For example, paradigm affects

what a program should do or how it should be done. Likewise, the data and behaviour of the program can be organised in either object-oriented (OOP) or functional manner. Finally, the control flow can be executed in procedural or event-driven ways, to name a few. Decisions at this level affect the general interaction. Moreover, if the programming model is a Domain-Specific Language (DSL), the selected syntax and semantics affect the readability, conciseness, and expressiveness of the resulting model, each of which, in turn, further influences the developer's productivity.

As is usually the case, there are no right or wrong choices, only tradeoffs. Both OOP and Functional programming have their strengths and weaknesses. Therefore, from the standpoint of this work, we rate the software libraries based on their productivity and how well they fit within a typical HPC context.

We present the table 1 with rating definitions for the language characteristics category.

Description	Rating
Choice of programming paradigm inhibits the productivity Additionally, the syntax is inconsistent, making the programming model hard to read and understand.	1
Programming paradigm choice is challenging for the HPC domain The general design is readable but with some ambiguity.	2
Choice of programming paradigm and syntax neither inhibits nor helps significantly in solving domain problems.	3
Selected paradigm and syntax make the software easy to understand and provides flexibility to solve wide range of problems.	4
Exceptional choice of paradigm leads to solutions that are highly performant, extensible and easy to integrate within the HPC ecosystem.	5

Table 1: Rating scheme for software tool language characteristics.

4.2 Software Performance

Borrowing from work by Elsharkawy et al. [65], we also consider the software scalability and performance as a metric. In a traditional HPC space, given that the current supercomputing systems are usually large-scale distributed clusters, the capability to scale with minimal overheads dictates the viability of a given software tool. In quantum computing, the primary consideration is given to hardware limitations and scalability regarding the number of qubits, their connectivity, and the maximum circuit depth that can be executed on the device. However, while initially not a priority, the actual software scalability becomes of growing importance. At the level of developing quantum programs, mapping arbitrary gates and circuits to corresponding instruction sets and topologies supported by devices is a challenging problem. To achieve high software scalability few components need to be balanced. These include: compilation times, use of quantum resources, and communication between hardware components.

Below, in table 2 we outline the rating for assessing the scalability and performance of the software.

Description	Rating
Lack of optimisations leads to inefficient, slow execution and high resource consumption.	1
Some consideration was given to the optimisation of quantum resources, however, there are noticeable bottlenecks.	2
With some optimisation of quantum resources, the programming model meets basic requirements, but has scalability limitations	3
Programming model and compilation are performant with reasonable resource usage	4
The programming model provides exceptional performance, is highly optimised and incurs minimal latency between classical and quantum co-processor.	5

Table 2: Rating scheme for programming model scalability and performance.

4.3 Tools and Ecosystem

It is often the case that the adoption of a framework by the community mimics the snowball effect. That is, the more popular the framework is, the faster the community of users grows. To start this process, the programming model must have an ecosystem of tools that make it easy to develop new code and integrate this code within existing codebases. The typical considerations include: Do development tools exist for the model? Is there a set of libraries supporting the development? Is there an active community which can help the newcomers?

Moreover, in fields that require extensive domain knowledge, such as quantum computing, specialised tools are needed, such as those that allow the visualisation of results and inspection of the system. From the HPC point of view, specialised profilers that support analysis of parallel execution are further important. A good programming model should provide those tools or easily integrate with existing solutions.

In table 3 we outlined the rating around the tooling for a given programming model:

Description	Rating
Limited or no development tools available, small and inactive community	1
Basic development tools such as simple debugger or inspector Small but moderately active community.	2
Adequate development tools for common use cases. Stable, active community.	3
Strong development tools with support for debuggers and profilers. Additional support for basic specialised tools (e.g. visualisation). Large and highly active community.	4
An exceptional ecosystem with a set of specialised libraries making the framework easy to adopt and use for a wide array of problems.	5

Table 3: Rating scheme for programming model tooling and development ecosystem.

4.4 Applications Suitability

As the main objective in the NISQ era is to show quantum utility, it is, therefore, crucial that the quantum toolchain makes it easy to implement potential candidates. If so, what

is the family of applications that are easily implementable? Likewise, given the continuous search for viable applications, a good framework should be robust enough to accommodate future changes and novelty coming in the post-NISQ era.

When analysing analogue quantum computing software, we look at its target applications and capability to truly gauge its specific domain. Below, in table 4, we outline the rating system for this category.

Description	Rating
Supports only generic operations at the low level (gates, pulses). No domain-specific structure.	1
Framework has higher level constructs, making some domain easier to implement.	2
Framework has a well-defined set of applications.	3
The framework implements a wide range of algorithms targeting well defined domain (e.g. combinatorial optimisation)	4
As 4 but also supports algorithms beyond projected use-cases.	5

Table 4: Rating scheme for programming model suitability for target application.

Given the natural set of applications for pulse-level analogue quantum computing, we will search for constructs that allow for easier Quantum Hamiltonian Simulation, Adiabatic evolution, and combinatorial optimisation.

4.5 Learning Curve

Another critical factor for the programming model's usability and adoption is the learning curve. Two primary considerations revolve around how easy it is to learn and be productive in a given framework and how complex the basic concepts and constructs in the model are. It should be noted that the learning curve of the programming model is not only due to the design choices but can also be due to the underlying domain in which the given framework operates (here, quantum computing). Nevertheless, the creators of the framework should put effort into helping the user learn by documenting and providing tutorials.

Table 5 outlines the rating categories for the programming model complexity.

Description	Rating
The programming model is extremely difficult to learn either due to complex concepts or lack of clarity. No documentation or code examples.	1
The framework is difficult to learn, requiring effort and extensive experience. Documentation and examples are lacking.	2
Neither easy nor hard to learn. Documentation is complete with some examples provided.	3
The programming model is easy to learn with clear documentation and examples describing some corner cases.	4
All effort has been made, so the framework is easy to learn, even for beginners. Documentation, examples and tutorials make a textbook.	5

Table 5: Rating scheme for the learning curve of the programming model.

4.6 Future prospects

When selecting software for a new project, an important consideration is the future prospects. Is the tool becoming more popular or obsolete? Is there active development with innovation and bug fixes implemented? Another aspect of software tools is maturity. With maturity comes experience and safety. Mature software usually has a large user base that has tested it extensively over the years, reducing the number of bugs. Additionally, due to high usage, the changes in mature frameworks are slower, more calculated, and less drastic, preserving backward compatibility.

Because quantum computing is a relatively new field, especially on the software side, most packages are still in active development or even at the experimental stage. It can be expected that many of the proposed solutions will not survive. This is a natural part of the dynamic field, such as quantum computing. However, in recent years, major vendors moved towards standardisation of their APIs; notable examples include references [94], [67] and [95]. In the quantum annealing space, D-Wave, given its long presence on the market, also has a somewhat stable offering.

The categories for the software's future prospects are presented in the table 6, below.

Description	Rating
Limited future prospects, project unmaintained or abandoned. It is in the pre-alpha stage as a prototype.	1
Framework has an uncertain future. It is either incomplete, alternatively, the competition offers better solutions.	2
The software tool has stable future prospects. It is likely to remain relevant but with limited growth.	3
The framework has a stable release and a promising future. It is possibly used by one of the large companies resulting in growing popularity and innovation.	4
Excellent future prospects. Framework is adopted by many users and is tied with many major players in a field.	5

Table 6: Rating scheme for the future prospects of the programming model.

4.7 Parallelism

The great success of HPC comes mainly from the use of different layers of parallelism. From the programmer's point of view, whether the parallelism is handled automatically (implicit) or requires explicit control dramatically affects the workflow and productivity. Moreover, it is essential to consider the supported granularity levels of parallelism. In the classical setting, this would refer, for example, to loop parallelism or task parallelism. In a quantum setting, one could consider shot-level or operator-level parallelism, where the expectation value calculation is distributed across many QPUs. Ideally, classical and quantum parallelism should be easily intermixed. Finally, the important consideration of parallel computation is scalability. That is, how well does the programming model scale to a large number of processors, be they CPUs or QPUs?

Considering all that, we present the rating for the parallelism employed in a programming model below, in table 7.

Description	Rating
There is no support for parallelism.	1
Basic support for parallelism but either limited or inefficient.	2
Support for classical or quantum parallelism with limited scaling.	3
Good support for parallelism, including quantum parallelism. Scales well to a large number of processors.	4
Exceptional support for parallelism on different levels, leveraging quantum and classical resources. HPC ready.	5

Table 7: Rating scheme for the parallelism capabilities of the programming model.

4.8 Data Management

Another aspect affecting the programming model is the data management or simply the memory model. The ubiquitous memory models in the HPC field are the shared-memory and distributed-memory models. In quantum computing, the standard model is the shared-memory model, where the CPU and QPU communicate using shared buffers and registers. In this setup, the classical processor can pass data to the register, then the quantum device performs a quantum program, and finally, the CPU retrieves the results. While this model is straightforward, there is limited consideration about who handles synchronisation or if multiple CPUs/QPUs can access the same buffer, opening possibilities for race conditions. Moreover, due to overheads from measurements and data loading onto the QPU, communicating data from quantum devices via classical channels will generally be a more considerable bottleneck than in the current HPC world. While there is ongoing research work on distributed quantum computing (summarised well in [76]), existing solutions are not mature enough to be directly integrated into existing quantum software frameworks [65].

Here, in table 8 we define rating for data management, borrowing from [65].

Description	Rating
There are no memory model considerations.	1
Framework supports shared memory model.	2
The programming model supports distributed classical memory with QPU access on-node or across nodes.	3
As 3 but also includes data mapping.	4
The programming model supports both distributed classical and quantum memory.	5

Table 8: Rating scheme for how the programming model handles data management.

4.9 Resource Management

In classical HPC, resource management is prevalent in that to fully utilise the available compute resources, the resources need to be scheduled and allocated carefully. There are a few standard considerations here. For example, if a resource (e.g. GPU) can be used by two jobs simultaneously. Another consideration would be the on-node availability of the resource. Likewise, similar considerations follow since QPU can be treated as an accelerator. Elsharkawy et al. [65]. introduced a further concept called *instruction pinning*,

which considers where the instructions should be executed - on a QPU controller or the classical host. Once again, from the standpoint of the HPC user, the ideal framework would give control of the resource management on a few levels, ranging from low-level, fine-grained control to complete abstraction, with some default but unoptimised settings.

In table 9, similarly to data management, we work with and simplify the rating scheme from [65] to consider different levels of accessibility:

Description	Rating
No resource management considerations.	1
Framework supports on-node resource management. (local QPU or qubits).	2
The programming model supports across-node resource management.	3
The programming model additionally supports instruction pinning.	4
Framework offers full qubit control across the whole system.	5

Table 9: Rating scheme for how the programming model handles resource management.

4.10 Hybrid Programming

With the slowdown of Moore's Law, the HPC community trends towards heterogeneous computing. In this paradigm, compute clusters incorporate specialised hardware, different to standard CPUs. The most prevalent accelerator is GPU. It is well expected that QPU will serve as an additional accelerator in the toolkit. Therefore, any HPC-ready framework must support a hybrid or heterogeneous programming model. The framework should be aware of and operate well on the CPU-QPU interactions at the most basic level. However, it is plausible (as proposed by NVIDIA and Quantum Machines [96]) that other devices might be useful for quantum computation, for example, in the error-correction setting [70]. Finally, given the efforts in the HPC field and common practice of using different frameworks for different levels of parallelism (for example, OpenMP with MPI), one should consider if a proposed programming model is interoperable with existing hybrid parallel programming models.

In table 10 we have outlined the rating for the hybrid programming functionality within the software programming toolkit in the table below.

Description	Rating
No support for hybrid programming, limited ability to leverage heterogeneous architectures	1
Basic support for hybrid programming, but with limitations or inefficiencies.	2
Reasonable support for hybrid programming, but with some limitations.	3
Good support for hybrid programming, effective utilisation of heterogeneous architectures.	4
Exceptional support for hybrid programming, highly optimised for performance and scalability.	5

Table 10: Rating scheme for the capability of the programming model for hybrid programming.

4.11 Portability

Two common challenges in the HPC space are portability and interoperability. For the most part, current supercomputing systems have various architectures, and the hardware comes from different suppliers. While the programming model usually falls into shared-memory, distributed-memory, or GPU accelerated category, different vendors require different ways to write software targeting their hardware. This is especially evident in the accelerator space. This creates a bottleneck, as software written for one supercomputer is not necessarily performant on a different machine. Because of this, much effort has been put into developing frameworks (RAJA [97], Kokkos [98], OpenMP [99], OpenACC [100]) that close this gap and support multiple platforms while abstracting hardware details from the users.

In quantum computing, it is still unclear which technology will become dominant. Each platform has its advantages and disadvantages. For superconducting qubits, this would be fast operation times and high-fidelity gates at the cost of short coherence time. Neutral atoms have excellent connectivity and long coherence times, but the application of operations is slow. Given the current state of things, it might be possible that future supercomputing systems accelerated by QPUs will use various quantum devices, each highly specialised for different needs. Therefore, it is crucial that any quantum software tools support different platforms and, from the HPC standpoint, facilitate seamless communication between them. For this paper, we have created a few categories outlined in table 11 below.

Description	Rating
Poor portability. Difficult to port to different hardware platforms or software stacks.	1
Below average portability. Some limitations or challenges when porting.	2
Average portability. Can be ported to different platforms but with some effort.	3
Good portability. Can be easily ported to different hardware platforms and software stacks.	4
Exceptional portability. Highly portable and adaptable to different environments.	5

Table 11: Rating scheme for programming model portability.

5 Analogue Programming Tools Analysis

Having defined the categories and metrics for the qualitative analysis, this section presents and discusses programming tools that support analogue quantum computation in more detail. While selecting the available software tools, we faced a dilemma: if we focus strictly on the packages that allow for adiabatic quantum computing (or annealing), the list of tools is severely limited. If we relax our conditions, on the other hand, to software that enables programming at the analogue pulse level and allows for the digitalisation of adiabatic evolution, then the set of available packages is extensive and includes well-known SDKs from major vendors. After the initial review, it was clear that many software libraries provided an interface for adiabatic evolution or the quantum adiabatic algorithm (QAA). However, most software that offers an interface for QAA is, in fact, digitising the algorithm during transpilation/compilation. This is reasonable as most existing platforms

Name	Abstraction	Host Language	Target Platform
Bloqade [23]	Pulse control	Julia/Python	NA
Pulser [24]	Pulse control	Python	NA
qupulse [101]	Pulse control	Python	SC
Artiq [102]	Pulse control	Python	TI
QGL [103]	Pulse control	Python	SC
LabOneQ [104]	Pulse control	Python	SC
Qua [105]	Pulse control	Python-like custom language	SC, NA
OpenPulse [106]	Pulse control	OpenQASM3 [66]	SC, NA
QiskitPulse [25]	Pulse control	Python	SC
Quantify [107]	Pulse control	Python	SC, NV, Spin
Q-CTRL Boulder Opal [108]	Pulse control	Python	SC, NA, TI
Qibolab [74]	Pulse control	Python/C	Compiles to Qua/LabOneQ
PulseLib [109]	Pulse control	Python	Agnostic
Amazon Braket (AHS) [110]	Pulse control	Python	QuEra NA
JaqalPaw [111]	Analogue Pulses as gates for Jaqal [112]	Python/Jaqal	TI
Qadence [113]	Digital-Analogue QAOA	Python	Compiles to Pulser
SimuQ [10]	HML	Python	NA, DW, SC, TI
QHDOPT [114]	QUBO	Python	Compiled w. SimuQ
D-Wave Ocean SDK [53]	QUBO with built-in helpers	Python	DW
C-to-D-Wave [115]	Compiles part of C to D-Wave	C	DW
XACC [116]	Middleware framework with Analogue Pulses	C++/Python	SC, TI, DW
Pennylane [75]	General QC framework with Analogue Pulses	Python	Agnostic

Table 12: Quantum Software with support for some form of analogue quantum computation. HML refers to Hamiltonian Modeling Language. Platform abbreviations: NA - Neutral Atoms, SC - Superconducting, TI - Trapped Ions, DW - D-Wave Quantum Annealer, NV - Nitrogen-Vacancy Center.

do not offer capabilities to perform true analogue adiabatic evolution. However, it creates a gap in software tooling, where the platforms with analogue capabilities are not fully utilised.

Therefore, first in table 12, we have collected all programming models with analogue quantum computing interfaces, be it AHS, QA or pulse level control. We observed that, generally, there are a large number of quantum control software packages. While several of them offer programming at the pulse level, in general, many don't support global interactions required for the analogue Hamiltonian quantum simulation or analogue quantum adiabatic algorithm. The most common use of those is for device calibration experiments and direct system control. To date, there has been limited work done using those packages for QAA.

It is true that using a hardware simulator (e.g. Qiskit Pulse simulator), it is possible to create a setup which, in principle, can model a wide range of Hamiltonians (as shown, for example, by Greenaway et al. [117]). However, for some platforms (e.g. transmon superconducting qubits), such a setup is currently impossible to realise in a physical setting [3].

As such, by analogue pulse, we understand a fully analogue pulse, with the possibility of global beams and arbitrary continuous waveforms and functions for amplitude, phase, and frequency. That is, a fully analogue pulse is capable of QAA or Hamiltonian simulation with no (or limited) Trotterization.

With that in mind, for a detailed analysis, we have selected tools that either allow for QA or are capable of full AHS on existing devices. These include SimuQ [10], Pulser [24], Bloqade [23], JaqalPaw [111], D-Wave Ocean SDK [53], QHDOPT [114], Amazon Braket [110], C-to-Dwave from Los Alamos National Laboratory (LANL) [115] and XACC [116]. The radar charts are presented in figure 6. Then, an exhaustive description of each model follows.

5.1 SimuQ

Criterion	Rating
Language Characteristics	4
Performance	3
Tooling	2
Applications	4
Learning Curve	4
Future Prospects	4
Parallelism	1
Data Management	2
Resource Management	1
Hybrid Programming	1
Portability	3

Table 13: Rating for the SimuQ.

SimuQ [10] is an open-source, Python-based, domain-specific language designed for Quantum Hamiltonian Simulation. The package operates explicitly on a higher abstraction level to reduce the barrier to entry for non-experts, requiring minimal physics domain

[3] While preparing this manuscript, IBM introduced that Qiskit Pulse will be deprecated and Qiskit 2.0 SDK will offer fractional gates instead.

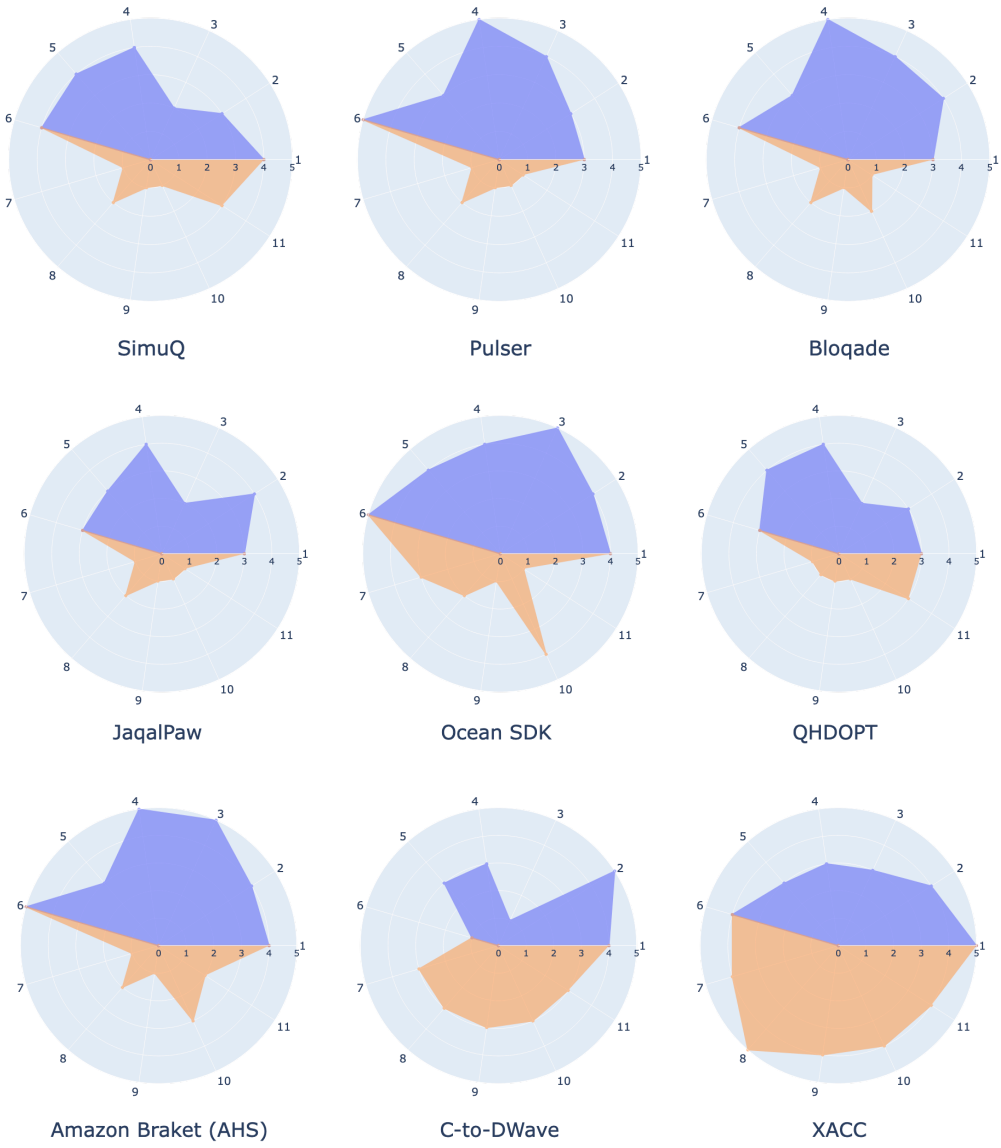

Figure 6: Radar plots representing the scores achieved by each software tool according to our rating scheme. Each number on the angular axis corresponds to categories outlined in section 4. Blue colouring covers general programming tool characteristics, whereas orange colour represents HPC-specific categories.

expertise. With a custom compilation framework, SimuQ targets NISQ devices from differ-

ent vendors, including IBM's Qiskit Pulse [25] for superconducting transmon qubit devices, QuEra's Bloqade [23] for neutral atom arrays, and quantum circuits for other general machines.

SimuQ, aside from OOP and procedural paradigm, offers a unique Hamiltonian Modelling Language (HML), which allows for accessible operator building. Those features make SimuQ readable, concise and easy to learn. Performance-wise, while their solver-based compiler includes some optimisations from the quantum resources standpoint, as authors note, there is a limited consideration given to the classical cost of compilation (although it was still more performant than Qiskit compiler at the time SimuQ was published [10]). From the development perspective, there is limited tooling available. Moreover, we have found that SimuQ fails to find a solution for some Ising Hamiltonians on the QuEra platform. The application domain is well-defined, with many constructs for achieving a wide range of goals. Finally, given that the QHDOPT [114] project adopted SimuQ, we deem it to have a stable future.

From an HPC perspective, SimuQ assumes a restricted Heterogeneous Quantum-Classical computation model in that the classical machine defines the quantum program and performs compilation, which takes the form of a pulse schedule that is offloaded to the quantum device. There are no explicit assumptions when it comes to data management. The framework employs an implicit shared-memory model between classical and quantum devices. SimuQ assumes local qubits (sites) allocation, and in that manner, little consideration is given to resource management. It also does not support classical communication and hybrid programming. Finally, given its compilation infrastructure, it is portable, targetting D-Wave, IBM, IonQ and QuEra and, being Python-based, it can be easily installed on most systems.

5.2 Pulser

Criterion	Rating
Language Characteristics	3
Performance	3
Tooling	4
Applications	5
Learning Curve	3
Future Prospects	5
Parallelism	1
Data Management	2
Resource Management	1
Hybrid Programming	1
Portability	1

Table 14: Rating for the Pulser.

Pulser [24] is a package aiming for the accessible design and execution of analogue pulse sequences, targeting neutral atoms. It is flexible enough that it should be executable on any such platform. It is embedded in Python, and a QuTip [118] simulator is included. The data structures are organised in an object-oriented manner with a sequence builder at the centre. The syntax is generally clear and readable; however, as a quantum hardware control software, it is low-level. The execution model lets the user offload the pulse sequence onto the backend. However, there is no classical control during the execution. The user can

generally select a local emulator or a remote Pasqal system in the cloud. Alternatively, one can also perform tensor network simulation remotely. From the performance point of view, certain routines (such as qubit placement) implemented in Python can be less performant than those written in a compiled language. While some consideration is given to the optimisation of quantum resources, much is left to the backend or the user. The framework has extensive tooling, which includes a built-in emulator for verification and visualisation tools. Additionally, it has a growing community with a recent, stable release (1.0.0). While Pulser has a certain amount of higher-level constructs for more straightforward pulse definition, it does not target specific applications, e.g., a quantum adiabatic algorithm needs to be implemented from the bottom up. One interesting feature of Pulser is that, besides hosting a library of device specifications, it allows users to define their own analogue devices and experiment freely. Pulser has a bright future because it is tied to Pasqal hardware, is actively developed, and serves as a base for a new digital-analogue framework, Qadence [113].

From the HPC point of view, Pulser does not have any direct consideration for classical or quantum parallelism. The employed memory model is a shared-memory model with local resource control and no communication. There is no support for hybrid programming either. Finally, from the portability point of view, Pulser supports only the Pasqal devices through the cloud (or Microsoft Azure). Up until recently, the main intermediate representation for the pulse sequences was in a JSON format, which, while not ideal, has a potential for future portability, assuming different platforms can compile from it. On the other hand, from the classical perspective, being a Python package distributed by pip, it is portable and can be installed on most HPC systems.

5.3 Bloqade

Criterion	Rating
Language Characteristics	3
Performance	4
Tooling	4
Applications	5
Learning Curve	3
Future Prospects	4
Parallelism	1
Data Management	2
Resource Management	1
Hybrid Programming	2
Portability	1

Table 15: Rating for the Bloqade.

Bloqade [23] is a software library from QuEra targeting the design and execution of analogue quantum Hamiltonian dynamics. It targets solely neutral atom architecture. It allows for easy generation of pulses and waveforms for quantum registers with arbitrary topologies. Furthermore, the package features a fast simulation of the entire Hilbert space and its subspaces subject to the Rydberg blockade. The sequences of pulses can be further compiled and executed on QuEra quantum devices using the Amazon Braket [110] service. The employed programming paradigm mixes functional capabilities with OOP constructs. In general, the API is low-level. However, it also provides higher-level features, such as

creating general observables using operator expressions. This results in a clear and expressible library suitable for most neutral atom experiments. The framework is written in Julia with an additional interface in Python, making it performant and classically portable. Additionally, simulations of the quantum systems can be offloaded to GPUs using CUDA. The overall tooling ecosystem is extensive, with built-in emulator and visualisation packages. With complete documentation and a set of examples, Bloqade is relatively easy to learn, with the main obstacle being the domain knowledge. Moreover, it has an active community, and with QuEra devices being offered by Amazon Braket service, it has a promising future.

Looking at Bloqade from the HPC perspective, there is a form of quantum parallelism on QuEra devices, which allows task parallelism on a single device. The assumed execution model does not allow classical control during the QPU execution. There is limited consideration of memory model and resource accessibility, with mostly shared-memory, site (qubit) access provided by the system. There is no communication consideration, be it classical or quantum. While there is no direct functionality for partitioning problems for hybrid accelerators (e.g. GPUs), the fact that the emulation part already uses CUDA means there is future potential in this space. Finally, from the quantum portability perspective, the only supported platform is a neutral atom device from QuEra. On a Braket system, the intermediate representation has the form of JSON, which might offer some potential in the future.

5.4 JaqalPaw

Criterion	Rating
Language Characteristics	3
Performance	4
Tooling	2
Applications	4
Learning Curve	3
Future Prospects	3
Parallelism	1
Data Management	2
Resource Management	1
Hybrid Programming	1
Portability	1

Table 16: Rating for the JaqalPaw.

JaqalPaw [111] is an offering from the Sandia National Laboratory for QSCOUT Trapped Ion devices. JaqalPaw is written as a framework in Python where the user defines custom gates in terms of pulses and waveforms. These are later imported to Jaqal assembly language format [112] files and freely used for further quantum processing. The gates are defined in OOP form, where a set of gates is defined as a class, and each gate definition is a method of said class. The built-in pulse definitions provide a high level of device control, which can correlate with the device's calibration data. From the QAA or Hamiltonian Quantum Simulation point of view, qubits can be manipulated via global beams, opening the possibility for truly analogue execution. The resulting programming model is straightforward, flexible and expressive but low-level, requiring domain knowledge from the user. Performance-wise, there are no direct optimisation routines. However,

the resulting constructs are at the (quantum) assembly level, leaving much space for the user to implement their own optimisations. Regarding the tooling ecosystem, a JaqalPaw emulator is shipped with the library. However, it has limited support for Jaqal's built-in multi-qubit gates (such as the Mølmer–Sørensen gate), making the development and verification of custom pulses rather challenging. From our experience, it creates a barrier to entry for users new to the field of trapped ions control. With JaqalPaw, a basic white paper is attached, explaining the structure and workflow. There are also some examples in the project repository. Overall, JaqalPaw is relatively easy for experts in the field to learn, although additional tutorials would make it easier for newcomers. The user community is relatively small; however, the project is actively updated. Application-wise, the objective of JaqalPaw is low-level control and flexibility. It does not have a domain-specific structure (or algorithms) implemented. Overall, the JaqalPaw provides a unique solution, giving it a stable future, but given its current low popularity, it is unclear how it will grow. From the HPC point of view, there are no considerations for parallelism or communication and data management. In practice, JaqalPaw should be considered as an extension to Jaqal assembly language. Indeed, the resources are allocated via Jaqal in an on-node, shared-memory model. Additionally, neither JaqalPaw (too low level) nor Jaqal directly supports hybrid programming. Finally, portability is limited as the JaqalPaw is tied to Jaqal and QSCOUT control systems.

5.5 D-Wave Ocean SDK

Criterion	Rating
Language Characteristics	4
Performance	4
Tooling	5
Applications	4
Learning Curve	4
Future Prospects	5
Parallelism	3
Data Management	2
Resource Management	1
Hybrid Programming	4
Portability	1

Table 17: Rating for the D-Wave Ocean SDK.

When it comes to quantum annealing frameworks, the D-Wave offering remains the primary option. D-Wave Ocean software development kit (SDK) [53] is a collection of open-source libraries for solving combinatorial optimisation problems. Among others, they offer tools for graph mapping, constraint compilation, and encoding of problems that fit QUBO formalism. Then, the stack offloads the selected program to one of a few backends. Those include classical simulated annealing backend, D-Wave Advantage machines via their API and a new offering - hybrid solver, which decomposes the problems at hand among available devices, including QPUs and GPUs. From the programming point of view, Ocean SDK offers a mix of programming paradigms across different levels of abstraction, making the resulting code readable and easy for developers to use. Performance-wise, the D-Wave offering is efficient, providing good mapping onto the device with reasonable resource usage. The wide array of tools and libraries provides excellent development experience and allows

for detailed experimentation. Extensive documentation and a set of examples ensure that the learning curve of this programming model is relatively flat. Overall, D-Wave is a leader in the QA field, and this framework's future prospects are bright.

In terms of HPC metrics, the employed execution model only allows the problem to be offloaded to the backend. There is no classical control during the execution (or it is hidden from the user). Parallelism has limited low-level control, with the main parallel construct being *TilingComposite* that tiles small problems multiple times over the structured Chimera or Pegasus samplers. However, with a hybrid solver, it is easy to partition the problem and run large tasks. It is hard to assess the memory model and resource management precisely as the backend is proprietary. However, the user has a certain degree of control via Sampler API. Based on their offering, we assume the memory model is distributed classically (due to how the hybrid solver is organised). The accessibility of resources on the quantum annealer is at the node level. The hybrid solver offers good hybrid programming that is out of the box. It also allows for defining new components, justifying the high score achieved. Unfortunately, the code produced by the framework is not portable, as the only target is D-Wave quantum annealers.

5.6 QHDOPT

Criterion	Rating
Language Characteristics	3
Performance	3
Tooling	2
Applications	4
Learning Curve	4
Future Prospects	3
Parallelism	1
Data Management	1
Resource Management	1
Hybrid Programming	1
Portability	3

Table 18: Rating for the QHDOPT.

QHDOPT [114] is a new, Python-based library implementing Quantum Hamiltonian Descent [119] on a set of existing quantum computers. QHDOPT's objective is to lower the barriers of entry for users from different science domains who lack expertise in quantum computing. It is built on top of SimuQ, which compiles to target systems. The general design is OOP, with clear, simple and concise constructs. The user solves the problem first by defining the Q matrix and constraints for the problem. Then, a model is created, which is then compiled and offloaded to the target backend. Given the project's main aim (being easy to use), the user's control over the computation is quite limited. From the development perspective, there are limited available tools, and some features are incomplete. Similarly, the user community is still small. Even though the documentation is still being created, it is relatively easy to learn in line with the authors' objective. At the current stage, given the project's freshness, we assume that its future is promising. It fills a niche in a field, but it is unclear how well it is going to be adopted by the wider community.

Looking at the project from the HPC point of view, there are no direct constructs for parallelism, data, and resource management. Similarly, there is no support for commu-

nication or hybrid programming. However, the package is portable both classically and quantum-wise. It can offload to D-Wave quantum annealers and IonQ devices, and being written in Python, it can be easily installed on most supercomputing systems.

5.7 Amazon Braket - Analog Hamiltonian Simulation

Criterion	Rating
Language Characteristics	4
Performance	4
Tooling	5
Applications	5
Learning Curve	3
Future Prospects	5
Parallelism	1
Data Management	2
Resource Management	1
Hybrid Programming	3
Portability	2

Table 19: Rating for the Amazon Braket.

Amazon Braket [110] is a cloud quantum computing service that offers a range of quantum machines, including Gate-based superconducting processors from Rigetti and IQM, gate-based ion-trap processors from IonQ, and Neutral atom-based quantum processors from QuEra. For the purpose of this work, we only consider the Neutral atom platform, which is capable of Analog Hamiltonian Simulation (AHS). Braket offers a module with the same name (AHS), which allows for defining pulses and allocating qubit registers that are later mapped to the device. The programming model is, to a large extent, similar to Bloqade. From the performance perspective, although the library is Python-based, it also offers C++ API, enabling integration within HPC workflows (although quantum resources are still accessed remotely). AHS provides excellent tooling, as found in Bloqade and more, coming directly from Braket features. The model is relatively easy to learn; however, like Bloqade, the user requires domain knowledge about the neutral atom platform. We believe that the AHS, being tied to the Amazon cloud service, has a promising future, especially given the range of devices offered. From the HPC point of view, parallelism and resource management are handled similarly to Bloqade's. That is, the user can build an atom register (the system assumes that atoms are available), and, depending on the problem size, execute independent tasks in parallel. There is no explicit way to specify communication and manage data on the device or in the broader cluster. However, Amazon Braket offers hybrid jobs, allowing easy coupling between classical and quantum processors. Finally, the programming model has limited portability, for now, targeting only QuEra devices and Amazon cloud systems.

5.8 C-to-D-Wave

Another, albeit experimental, set of projects targeting quantum annealing is software developed by Los Alamos National Laboratory (LANL). Starting from the bottom-up, the QMASM [120] is a macro-based assembly language for D-Wave systems. Its main purpose is to create QUBO or Ising problem formulations at a low level. Higher-level operations (e.g. gates) can be created using the !begin−macro directive. Additionally, it offers a

Criterion	Rating
Language Characteristics	4
Performance	5
Tooling	1
Applications	3
Learning Curve	3
Future Prospects	1
Parallelism	3
Data Management	3
Resource Management	3
Hybrid Programming	3
Portability	3

Table 20: Rating for the C-to-DWave.

!bqm_type directive, which can specify how to interpret weights and the strength of the interaction within the formulation. Available options are "qubo" and "ising". This allows users to express parts of the program as QUBO and parts of it as Ising-model Hamiltonians.

Next, the edif2qmasm [121] project translates hardware-description language programs - Verilog/VHD - to QMASM, allowing offloading to D-Wave annealers. The main advantages of this approach are that one can intermix classical, standard language constructs with specifications for quantum operations, and further, one has control over bit/qubit widths, fully controlling the available resources (something seen in more recent frameworks, such as OpenQASM3).

Finally, two projects are building upon the previous two. C-to-D-Wave [115] compiles a small subset of C language to the form that can, in principle, run on a D-Wave annealer. In that way, one can, for example, define a C function that defines the MAXCUT problem, which is then compiled to Verilog and down to QMASM for D-Wave API consumption. The C-to-Verilog compilation uses Clang tools and hence achieves good scalability; however, no further optimisations are considered on the lower levels. Another project is QA Prolog [122], which, again, has extended functionality to define Ising-model Hamiltonian functions. This fits well into the language combined with built-in constraint-logic programming. Because it produces classical Hamiltonians, it can, in principle, offload work to classical annealers as well. Although it has definitely interesting use cases and would be of much interest to the computer science and physics communities, from the HPC point of view, Prolog as a language has limited support and is rather niche.

While those projects are primarily proofs-of-concept, they are interesting because the stack attempts to integrate QA offloading from higher-level classical languages (C, Prolog). Even though the abstraction level could be considered high, as the user just specifies their problem, which is then compiled and executed on the D-Wave system, overall, there is limited potential for further HPC integration. There is little to no consideration for optimisation, data and resource management. Moreover, the project seems to be discontinued, with the last commits dating back to 5 years (2019). As such, although attractive, we deem those repositories unhelpful for future HPC-QC integration.

5.9 XACC

XACC [116] is a framework targeting general QC technology with an extensive feature list for HPC integration. It is built from three primary layers: front-end, middle-end

Criterion	Rating
Language Characteristics	5
Performance	4
Tooling	3
Applications	3
Learning Curve	3
Future Prospects	4
Parallelism	4
Data Management	5
Resource Management	4
Hybrid Programming	4
Portability	4

Table 21: Rating for the XACC.

and backend, each providing different functionality. Thanks to this modular, compiler-like architecture, the set of supported front-end libraries and backends is immense. The programming model employed in XACC is a host-device, or kernel-based, popular in hardware accelerator space. Here, similarly, a QPU is treated as just another accelerator. The code representation is transformed to an intermediate representation (IR) residing at the middle layer, with numerous optimisations implemented for further compilation. Finally, IR is lowered to an appropriate quantum circuit format or pulse schedule for a target backend. XACC offers many levels of abstraction, supporting standard gate-based circuit building and a number of pre-implemented quantum algorithms and circuit generators. It also supports pulse-level programming. However, it is somewhat limited in this regard, bound to the QuaC [123] simulator backend. Additionally, it supports offloading to the D-Wave devices, making it an extremely versatile framework for different modes of quantum computing. From the memory model and resource management point of view, XACC supports the Message-Passing Interface (MPI), which is ubiquitous in HPC and used for large-scale distributed computing. Due to the employment of MPI and an efficient compilation stack, we also rate XACC high in the performance category. While the framework seems stable enough, we found it somewhat hard to get working on different supercomputing systems. Additionally, it seems the project progress stalled, with much momentum being moved to CUDA-Q [124].

While XACC is the main contender for tight HPC-QC integration, it lacks enough capability for robust analogue quantum computation, mainly in the space of AHS. One advantage of XACC is that its architecture is extensible by modules, making it relatively future-proof. In that way, some of its shortcomings could be addressed by providing an analogue-based extension package.

6 Discussion

Having analysed the existing programming models for analogue quantum computation, we follow with a discussion, including gathered insights and limitations of this study.

6.1 Insights

The emerging insight from our analysis is that by and large, the existing frameworks designed explicitly for analogue quantum computation are not ready for integration with

HPC systems. This is unsurprising, as they are mainly at the experimental and development levels. Nevertheless, as there is growing evidence of the utility of quantum processing at the analogue level, it is vital to either redesign existing or introduce new software tools that make HPC integration easier.

The main limitation of existing libraries and DSLs can be traced down to the implicit execution model. In principle, it is hard to think about classical operations on a device for AQC (aside from error correction). As such, the software usually does not consider the execution model extensively, instead using a restricted model with remote submission to a QPU. Additional limitations include little consideration of data and resource management and scalability in terms of software. Furthermore, for the most part, any consideration of interaction with classical systems is limited.

XACC framework is a definite outlier here, providing varied offerings and supporting offloading to D-Wave systems and programming at the pulse level. Given its modular architecture, host language being C++ and support for MPI, it is the main contender for HPC-QC integration. Indeed, our results here are confirmed by Elsharkawy et al. [65], who also analysed XACC as a part of their study.

Another limitation of existing tools, from an HPC point of view, is the extensive use of Python as a host language. While Python provides much flexibility and facilitates productivity, it is not ideal for the constraints of high-performance systems. This problem is remedied to some extent by implementing time-critical portions of code in compiled languages. An additional benefit is that Python has a low barrier to entry and is suitable for writing new scientific code. However, the actual challenge in the HPC community is adopting and extending existing code bases, which are primarily written in C and Fortran. Therefore, any HPC-worthy software toolkit has to have (at least) a C API that can be easily integrated with existing codebases.

In the QA space, D-Wave's offering reigns supreme. This can mainly be attributed to the fact that D-Wave was the first and leading vendor of quantum annealers, while other major players focused on developing digital quantum computers. Nevertheless, with the growing popularity and realisation of neutral atom and trapped ion systems, for which it is relatively easy to switch modes of operation (between digital and analogue), there is a need for open-source tools that will make it easy to program quantum annealing on those machines.

As a side note, more recently, a hybrid version of digital-analogue quantum computer [125, 126] emerged with promising results for QAOA [12]. To this day, however, it was not widely adopted, and more recently, a package called Qadence [113] emerged. In general, both AQC and digital-analogue schemes remain at the experimental level without major software development kits. This presents a gap for future exploitation. Moreover, offerings from D-Wave focus only on a specific, restricted form of Ising Hamiltonian, and as such, their programming model only caters to this restricted model. However, in general, AQC is universal and equivalent to circuit-based QC. This hints that there is a need for programming models that allow for the practical exploitation of fully capable analogue quantum devices to realise AQC.

6.2 Limitations

Our selection for specific use cases and applications limits the available hardware with desired capabilities. Because most of the existing hardware control software is tied to a specific platform, by extension, this also limits the corresponding control software. This is the natural limitation of this review. Furthermore, the analysis would benefit from the deeper benchmarking of the associated packages. One natural candidate for benchmarking

would be the compilation performance. Nevertheless, due to the characteristics of most packages, this is currently impossible as the packages are either too low level and the resulting pulse sequence is directly submitted to the device, or the compilation details are hidden on the remote backend. This further highlights the need for open-source projects that handle instruction offloading to the device.

Another major limitation of this study is that the metrics or benchmarks presented here are qualitative and are based on the authors' judgment. Unfortunately, the space of quantum programming models is still at the experimental stage and is very dynamic. As such, there is lack of major code bases which use those technologies. The study would benefit from other metrics such as Source lines of code (SLOC), Cyclomatic Complexity [127], Halstead metrics [128] and Fault Analysis of large-scale repositories. It is too early for that and deeper analysis must be done in the future.

However, we argue that this space is currently stuck in a cycle. For larger code bases to be produced, the integration challenges need to be solved first. Naturally, in the future, we expect that the range of tools will converge to a few major players, and more quantitative analysis should be possible. The study should be repeated in this space.

7 Conclusion

Quantum computing offers great promise of advantage on some classically hard problems. Nevertheless, to truly leverage this emerging technology, besides improvements in hardware, there is much work to be done in the software toolchain space. With the current low-level circuit programming model, it is clear that programming quantum computers at scale is challenging. An analogue quantum computing offers some remedy to this by possibly opening ways for higher-level abstraction constructs. The additional benefit of analogue techniques is the capability to leverage existing NISQ hardware better.

In this work, we have reviewed existing quantum software tools with analogue capabilities and examined frameworks that can run AHS and QA in detail. The conclusion is that the existing tools lack capabilities for seamless integration in HPC environments, making them hard to adopt for useful applications.

While there is a large set of pulse-level hardware control libraries, using those for the general quantum community remains a niche application that requires extensive domain knowledge. As such, there is a need to develop new tools that scale and are easy to use, built for HPC and for scientists from other domains.

Similarly, the research groups working on AHS made giant leaps into uncharted territories, probing regimes unattainable for classical computation. Nevertheless, we identify that the community should focus efforts on building programming models for integration with existing physical codes, such as codes used to simulate many-body systems, to speed up those and shift the technology from experimental setups towards production and industry.

Acknowledgements

We thank the anonymous reviewers for their constructive feedback. Additionally, we would like to thank Dr. Joseph Lee for his helpful discussions during early stages of drafting this review. The authors acknowledge support from the Engineering and Physical Sciences Research Council grant number EP/Z53318X/1.

References

[1] Sukhpal Singh Gill et al. "Quantum computing: A taxonomy, systematic review and future directions". Software: Practice and Experience **52**, 66–114 (2022).

[2] "IBM's roadmap for scaling quantum technology | IBM Quantum Computing Blog". url: https://www.ibm.com/quantum/blog/ibm-quantum-roadmap. (accessed: 2024-08-07).

[3] "Google Quantum AI". url: https://quantumai.google/. (accessed: 2024-08-07).

[4] "Quantinuum | Hardware". url: https://www.quantinuum.com/hardware. (accessed: 2024-08-07).

[5] "Quantum Computing | Rigetti Computing". url: https://www.rigetti.com/. (accessed: 2024-08-07).

[6] Earl Campbell. "A series of fast-paced advances in Quantum Error Correction". Nature Reviews Physics **6**, 160–161 (2024).

[7] Yunong Shi et al. "Resource-Efficient Quantum Computing by Breaking Abstractions". Proceedings of the IEEE **108**, 1353–1370 (2020). arXiv:2011.00028.

[8] Zhiding Liang et al. "Towards Advantages of Parameterized Quantum Pulses" (2023). arXiv:2304.09253.

[9] Kaitlin N. Smith et al. "Programming physical quantum systems with pulse-level control". Frontiers in Physics**10** (2022).

[10] Yuxiang Peng et al. "SimuQ: A Framework for Programming Quantum Hamiltonian Simulation with Analog Compilation". Proceedings of the ACM on Programming Languages **8**, 2425–2455 (2024). arXiv:2303.02775.

[11] Andrew J. Daley et al. "Practical quantum advantage in quantum simulation". Nature **607**, 667–676 (2022).

[12] David Headley et al. "Approximating the quantum approximate optimization algorithm with digital-analog interactions". Physical Review A **106**, 042446 (2022). arXiv:2002.12215.

[13] Zhiding Liang et al. "Hybrid Gate-Pulse Model for Variational Quantum Algorithms" (2022). arXiv:2212.00661.

[14] Travis S. Humble, Alexander McCaskey, Dmitry I. Lyakh, Meenambika Gowrishankar, Albert Frisch, and Thomas Monz. "Quantum Computers for High-Performance Computing". IEEE Micro **41**, 15–23 (2021).

[15] Bela Bauer et al. "Quantum algorithms for quantum chemistry and quantum materials science". Chemical Reviews **120**, 12685–12717 (2020). arXiv:2001.03685.

[16] Message Passing Interface Forum. "MPI: A message-passing interface standard version 4.1". (2023). url: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[17] Victor Artigues et al. "Evaluation of performance portability frameworks for the implementation of a particle-in-cell code" (2019). arXiv:1911.08394.

[18] Zhe Fan, Feng Qiu, A. Kaufman, and S. Yoakum-Stover. "GPU Cluster for High Performance Computing". In SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing. Pages 47–47. (2004).

[19] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. "TOP500 List - June 2024 | TOP500". url: https://www.top500.org/lists/top500/list/2024/06/. (accessed: 2024-08-12).

[20] EPCC. "A brief history of EPCC". url: https://www.epcc.ed.ac.uk/about-us/brief-history-epcc. (accessed: 29/11/2024).

[21] P. D. Stephens and J. K. Yarwood. “Providing multi-user access to distributed array processors”. Software: Practice and Experience **16**, 531–539 (1986).
[22] Wei-Chen Lin and Simon McIntosh-Smith. “Comparing Julia to Performance Portable Parallel Programming Models for HPC”. In 2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). Pages 94–105. St. Louis, MO, USA (2021). IEEE.
[23] QuEra (2023). url: `https://github.com/QuEraComputing/Bloqade.jl/`.
[24] Henrique Silvério et al. “Pulser: An open-source package for the design of pulse sequences in programmable neutral-atom arrays”. Quantum **6**, 629 (2022). arXiv:2104.15044.
[25] Thomas Alexander et al. “Qiskit pulse: programming quantum computers through the cloud with pulses”. Quantum Science and Technology **5**, 044006 (2020).
[26] H. F. Trotter. “On the product of semi-groups of operators”. Proceedings of the American Mathematical Society **10**, 545–551 (1959).
[27] Laura Clinton, Johannes Bausch, and Toby Cubitt. “Hamiltonian simulation algorithms for near-term quantum hardware”. Nature Communications **12**, 4989 (2021).
[28] John Preskill. “Quantum Computing in the NISQ era and beyond”. Quantum **2**, 79 (2018). arXiv:1801.00862.
[29] Kishor Bharti et al. “Noisy intermediate-scale quantum (NISQ) algorithms”. Reviews of Modern Physics **94**, 015004 (2022). arXiv:2101.08448.
[30] Zhiding Liang et al. “NAPA: Intermediate-Level Variational Native-Pulse Ansatz for Variational Quantum Algorithms”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and SystemsPages 1–1 (2024).
[31] Milan Kornjača et al. “Large-scale quantum reservoir learning with an analog quantum computer” (2024). arXiv:2407.02553.
[32] Tameem Albash and Daniel A. Lidar. “Adiabatic Quantum Computing”. Reviews of Modern Physics **90**, 015002 (2018). arXiv:1611.04471.
[33] Dorit Aharonov et al. “Adiabatic Quantum Computation is Equivalent to Standard Quantum Computation”. SIAM Journal on Computing **37**, 166–194 (2007).
[34] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. “Optimization by Simulated Annealing”. Science **220**, 671–680 (1983).
[35] B. Apolloni, C. Carvalho, and D. de Falco. “Quantum stochastic optimization”. Stochastic Processes and their Applications **33**, 233–244 (1989).
[36] B Apolloni, N Cesa-Bianchi, and D De Falco. “A numerical implementation of "quantum annealing"”. Rapporto interno, Dipartimento di Scienze dell’Informazione Università degli Studi di Milano**55** (1988). url: `https://cds.cern.ch/record/192546`.
[37] David Elieser Deutsch and Roger Penrose. “Quantum computational networks”. Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences **425**, 73–90 (1997).
[38] Edward Farhi et al. “Quantum Computation by Adiabatic Evolution” (2000). arXiv:quant-ph/0001106.
[39] Edward Farhi et al. “A Quantum Adiabatic Evolution Algorithm Applied to Random Instances of an NP-Complete Problem”. Science **292**, 472–475 (2001).
[40] Jérémie Roland and Nicolas J. Cerf. “Quantum search by local adiabatic evolution”. Physical Review A **65**, 042308 (2002).
[41] M. S. Sarandy and D. A. Lidar. “Adiabatic Quantum Computation in Open Systems”. Physical Review Letters **95**, 250503 (2005).
[42] Zhaohui Wei and Mingsheng Ying. “A modified quantum adiabatic evolution for the Deutsch–Jozsa problem”. Physics Letters A **354**, 271–273 (2006).

[43] Shuxian Jiang et al. “Quantum Annealing for Prime Factorization”. Scientific Reports **8**, 17667 (2018).
[44] J. Brooke et al. “Quantum Annealing of a Disordered Magnet”. Science **284**, 779–781 (1999).
[45] J. Brooke, T. F. Rosenbaum, and G. Aeppli. “Tunable quantum tunnelling of magnetic domain walls”. Nature **413**, 610–613 (2001).
[46] Sheir Yarkoni et al. “Quantum Annealing for Industry Applications: Introduction and Review”. Reports on Progress in Physics **85**, 104001 (2022). arXiv:2112.07491.
[47] L. Hormozi et al. “Nonstoquastic Hamiltonians and Quantum Annealing of an Ising Spin Glass”. Physical Review B **95**, 184416 (2017). arXiv:1609.06558.
[48] Tameem Albash and Daniel A. Lidar. “Demonstration of a scaling advantage for a quantum annealer over simulated annealing”. Physical Review X **8**, 031016 (2018). arXiv:1705.07452.
[49] Atanu Rajak et al. “Quantum Annealing: An Overview”. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences **381**, 20210417 (2023). arXiv:2207.01827.
[50] Sheir Yarkoni, Hao Wang, Aske Plaat, and Thomas Bäck. “Boosting quantum annealing performance using evolution strategies for annealing offsets tuning.”. In Sebastian Feld and Claudia Linnhoff-Popien, editors, QTOP@NetSys. Volume 11413 of Lecture Notes in Computer Science, pages 157–168. Springer (2017). url: http://dblp.uni-trier.de/db/conf/kivs/qtop2019.html#Yarkoni0PB17.
[51] Ting-Jui Hsu, Fengping Jin, Christian Seidel, Florian Neukart, Hans De Raedt, and Kristel Michielsen. “Quantum annealing with anneal path control: application to 2-sat problems with known energy landscapes” (2018). arXiv:1810.00194.
[52] Andrew D. King et al. “Computational supremacy in quantum simulation” (2024). arXiv:2403.00910.
[53] D-Wave Systems Inc. “Ocean™ Developer Tools | D-Wave”. url: https://www.dwavesys.com/solutions-and-products/ocean/. (accessed: 2024-08-12).
[54] D-Wave Systems Inc. “D-Wave Hybrid Solver Service: An Overview”. url: https://www.dwavesys.com/media/4bnpi53x/14-1039a-b_d-wave_hybrid_solver_service_an_overview.pdf. (accessed: 2024-10-27).
[55] Elijah Pelofske, Georg Hahn, and Hristo N. Djidjev. “Parallel quantum annealing”. Scientific Reports **12**, 4499 (2022).
[56] Sabine Jansen, Mary-Beth Ruskai, and Ruedi Seiler. “Bounds for the adiabatic approximation with applications to quantum computation”. Journal of Mathematical Physics **48**, 102111 (2007).
[57] Alexander Elgart and George A. Hagedorn. “A note on the switching adiabatic theorem”. Journal of Mathematical Physics **53**, 102202 (2012).
[58] Jeffrey Burt. “D-Wave Is Still Making The Case For Annealing Quantum Computing”. url: https://www.nextplatform.com/2024/06/18/d-wave-is-still-making-the-case-for-annealing-quantum-computing/. (accessed: 2024-08-12).
[59] Tiffany M. Mintz et al. “QCOR: A Language Extension Specification for the Heterogeneous Quantum-Classical Model of Computation”. ACM Journal on Emerging Technologies in Computing Systems **16**, 22:1–22:17 (2020).
[60] Thien Nguyen et al. “Quantum Circuit Transformations with a Multi-Level Intermediate Representation Compiler” (2021). arXiv:2112.10677.
[61] Alexander McCaskey and Thien Nguyen. “A MLIR Dialect for Quantum Assembly Languages” (2021). arXiv:2101.11365.

[62] Jingzhe Guo et al. “isQ: An Integrated Software Stack for Quantum Programming”. IEEE Transactions on Quantum Engineering **4**, 1–16 (2023).
[63] Michael B. Healy et al. “Design and architecture of the IBM Quantum Engine Compiler” (2024). arXiv:2408.06469.
[64] Tobias Schmale et al. “Backend compiler phases for trapped-ion quantum computers”. In 2022 IEEE International Conference on Quantum Software (QSW). Pages 32–37. (2022). arXiv:2206.00544.
[65] Amr Elsharkawy et al. “Integration of Quantum Accelerators with High Performance Computing – A Review of Quantum Programming Tools” (2023). arXiv:2309.06167.
[66] Andrew W. Cross et al. “OpenQASM 3: A broader and deeper quantum assembly language”. ACM Transactions on Quantum Computing **3**, 1–50 (2022). arXiv:2104.14722.
[67] Alan Geller. “Introducing Quantum Intermediate Representation (QIR)”. url: https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/. (accessed: 2024-10-27).
[68] Daniel Gottesman. “An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation” (2009). arXiv:0904.2557.
[69] Joschka Roffe. “Quantum Error Correction: An Introductory Guide” (2019). arXiv:1907.11157.
[70] Yaniv Kurman et al. “Benchmarking the ability of a controller to execute quantum error corrected non-Clifford circuits” (2024). arXiv:2311.07121.
[71] Ben Barber et al. “A real-time, scalable, fast and highly resource efficient decoder for a quantum computer”. Nature Electronics (2025). arXiv:2309.05558.
[72] Andy B. Yoo, Morris A. Jette, and Mark Grondona. “SLURM: Simple Linux Utility for Resource Management”. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, Job Scheduling Strategies for Parallel Processing. Pages 44–60. Berlin, Heidelberg (2003). Springer.
[73] Lucas Lamata et al. “Digital-analog quantum simulations with superconducting circuits”. Advances in Physics: X **3**, 1457981 (2018).
[74] Stavros Efthymiou et al. “Qibolab: an open-source hybrid quantum operating system”. Quantum **8**, 1247 (2024).
[75] Ville Bergholm et al. “PennyLane: Automatic differentiation of hybrid quantum-classical computations” (2022). arXiv:1811.04968.
[76] David Barral et al. “Review of Distributed Quantum Computing. From single QPU to High Performance Quantum Computing” (2024). arXiv:2404.01265.
[77] Christoph Kessler. “Models for Parallel Computing: Review and Perspectives”. Mitteilungen-Gesellschaft für Informatik eV, Parallel-Algorithmen und Rechnerstrukturen Pages 13–29 (2007). url: https://www.diva-portal.org/smash/get/diva2:261583/FULLTEXT01.pdf.
[78] “Introduction to Parallel Computing Tutorial | HPC @ LLNL”. url: https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial#Models. (accessed: 2024-08-12).
[79] John Nickolls et al. “Scalable parallel programming with CUDA”. ACM SIGGRAPH 2008 classes Pages 1–14 (2008).
[80] Frank Arute et al. “Quantum supremacy using a programmable superconducting processor”. Nature **574**, 505–510 (2019).

[81] Nils Herrmann et al. “Quantum utility – definition and assessment of a practical quantum advantage”. In 2023 IEEE International Conference on Quantum Software (QSW). Pages 162–174. (2023).
[82] Manuel A. Serrano et al. “Quantum Software Components and Platforms: Overview and Quality Assessment”. ACM Computing Surveys **55**, 164:1–164:31 (2022).
[83] Salonik Resch and Ulya R. Karpuzcu. “Quantum Computing: An Overview Across the System Stack” (2019). arXiv:1905.07240.
[84] Mark Fingerhuth, Tomáš Babej, and Peter Wittek. “Open source software in quantum computing”. PLOS ONE **13**, e0208561 (2018).
[85] Bettina Heim et al. “Quantum programming languages”. Nature Reviews Physics **2**, 709–722 (2020).
[86] Sunita Garhwal, Maryam Ghorani, and Amir Ahmad. “Quantum Programming Language: A Systematic Review of Research Topic and Top Cited Languages”. Archives of Computational Methods in Engineering **28**, 289–310 (2021).
[87] Frederic T. Chong, Diana Franklin, and Margaret Martonosi. “Programming languages and compiler design for realistic quantum hardware”. Nature **549**, 180–187 (2017).
[88] Philipp Hauke et al. “Perspectives of quantum annealing: Methods and implementations”. Reports on Progress in Physics **83**, 054401 (2020). arXiv:1903.06559.
[89] Lior Ella et al. “Quantum-classical processing and benchmarking at the pulse-level” (2023). arXiv:2303.03816.
[90] R. Au-Yeung et al. “Quantum algorithms for scientific computing”. Reports on Progress in Physics **87**, 116001 (2024).
[91] Ercüment Kaya. “A Software Platform to Support Disaggregated Quantum Accelerators”. In Workshop: International Workshop on RESource DISaggregation in High Performance Computing (RESDIS). Atlanta GA USA (2024). IEEE.
[92] Thomas Beck et al. “Integrating quantum computing resources into scientific HPC ecosystems”. Future Generation Computer Systems **161**, 11–25 (2024).
[93] Masoud Mohseni et al. “How to Build a Quantum Supercomputer: Scaling Challenges and Opportunities” (2024). arXiv:2411.10406.
[94] Gadi Aleksandrowicz et al. “Qiskit: An Open-source Framework for Quantum Computing”. url: https://zenodo.org/records/2562111. (accessed: 2024-03-22).
[95] Krysta Svore et al. “Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL”. In Proceedings of the Real World Domain Specific Languages Workshop 2018. Pages 1–10. RWDSL2018New York, NY, USA (2018). Association for Computing Machinery.
[96] NVIDIA and Quantum Machines. “NVIDIA DGX Quantum”. url: https://www.nvidia.com/en-us/data-center/dgx-quantum/. (accessed: 2024-10-27).
[97] Richard D. Hornung and Jeffrey A. Keasler. “The RAJA Portability Layer: Overview and Status”. Technical Report LLNL-TR-661403. Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States) (2014).
[98] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. Journal of Parallel and Distributed Computing **74**, 3202–3216 (2014).
[99] OpenMP Architecture Review Board. “OpenMP Application Programming Interface 5.2”. (2021). url: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.
[100] OpenACC-Standard.org. “The OpenACC Application Programming Interface 3.3”.

(2022). url: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf.
[101] Humpohl Simon et al. "qutech/qupulse: qupulse 0.10". url: https://zenodo.org/doi/10.5281/zenodo.2650139. code: qutech/qupulse.
[102] Sébastien Bourdeauducq et al. "Artiq 1.0". url: https://doi.org/10.5281/zenodo.51303. code: m-labs/artiq v1.0.
[103] Raytheon BBN Technologies Quantum Group (2020). url: https://github.com/BBN-Q/QGL.
[104] Zurich Instruments. "zurich-instruments/laboneq: Labone q 2.39.0". url: https://www.zhinst.com/europe/en/quantum-computing-systems/labone-q.
[105] Quantum Machines. "Qua: Quantum universal assembly". url: https://www.quantum-machines.co/technology/qua-universal-quantum-language/.
[106] David C. McKay et al. "Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments" (2018). arXiv:1809.03452.
[107] Qblox and Orange QS. "Quantify os and scheduler". url: https://quantify-os.org/.
[108] Q-CTRL. "Boulder opal: Quantum control infrastructure software for reasearch and development professionals building the future.". url: https://docs.q-ctrl.com/boulder-opal.
[109] Aniket S. Dalvi et al. "Graph-Based Pulse Representation for Diverse Quantum Control Hardware" (2024). arXiv:2409.08407.
[110] Amazon Web Services. "Amazon Braket". url: https://aws.amazon.com/braket/.
[111] Daniel Lobser et al. "JaqalPaw: A Guide to Defining Pulses and Waveforms for Jaqal" (2023). arXiv:2305.02311.
[112] B. C. A. Morrison et al. "Just Another Quantum Assembly Language (Jaqal)". 2020 IEEE International Conference on Quantum Computing and Engineering (QCE)Pages 402–408 (2020).
[113] Dominik Seitz et al. "Qadence: a differentiable interface for digital-analog programs" (2024). arXiv:2401.09915.
[114] Samuel Kushnir et al. "QHDOPT: A Software for Nonlinear Optimization with Quantum Hamiltonian Descent" (2024). arXiv:2409.03121.
[115] Mohamed W. Hassan, Scott Pakin, and Wu Chun Feng. "C to D-Wave: A high-level C compilation framework for quantum annealers". In 2019 IEEE High Performance Extreme Computing Conference, HPEC 2019. Page 8916231. IEEE (2019).
[116] Alexander J. McCaskey et al. "XACC: A System-Level Software Infrastructure for Heterogeneous Quantum-Classical Computing" (2019). arXiv:1911.02452.
[117] Sean Greenaway et al. "Analogue Quantum Simulation with Fixed-Frequency Transmon Qubits". Quantum **8**, 1263 (2024). arXiv:2211.16439.
[118] J. R. Johansson, P. D. Nation, and Franco Nori. "QuTiP 2: A Python framework for the dynamics of open quantum systems". Computer Physics Communications **184**, 1234–1240 (2013).
[119] Jiaqi Leng et al. "Quantum Hamiltonian Descent" (2023). arXiv:2303.01471.
[120] Scott Pakin. "A quantum macro assembler". In 2016 IEEE High Performance Extreme Computing Conference (HPEC). Pages 1–8. (2016).
[121] Scott Pakin. "Targeting Classical Code to a Quantum Annealer". In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. (2019). url: https://dl.acm.org/doi/10.1145/3297858.3304071.

[122] Scott Pakin. "Performing fully parallel constraint logic programming on a quantum annealer". Theory and Practice of Logic Programming **18**, 928–949 (2018).
[123] Otten Matthew (2016). url: https://github.com/0tt3r/QuaC.
[124] Jin-Sung Kim et al. "CUDA Quantum: The Platform for Integrated Quantum-Classical Computing". In 2023 60th ACM/IEEE Design Automation Conference (DAC). Pages 1–4. (2023).
[125] Adrian Parra-Rodriguez et al. "Digital-analog quantum computation". Physical Review A **101**, 022305 (2020).
[126] Ana Martin et al. "Digital-analog quantum algorithm for the quantum Fourier transform". Physical Review Research **2**, 013012 (2020).
[127] T.J. McCabe. "A Complexity Measure". IEEE Transactions on Software Engineering **SE-2**, 308–320 (1976).
[128] Maurice Howard Halstead. "Elements of software science". North Holland Amsterdam and N.Y. (1977). url: https://www.semanticscholar.org/paper/Elements-of-software-science-%3A-M.H.-Halstead%3A-by-US-Remoortere/75b145b52470d2e793b8cf96ba55a3690cd42ec9.