Leveraging Hybrid Cloud HPC with Multitier Reactive Programming

Daniel Sokolowski*, Jan-Patrick Lehr†, Christian Bischof‡ and Guido Salvaneschi§
*Reactive Programming Technology, †‡Scientific Computing, §Programming
*†‡Technical University of Darmstadt, Germany, §University of St. Gallen, Switzerland
{*sokolowski@cs., †jan-patrick.lehr@, ‡christian.bischof@}tu-darmstadt.de, §guido.salvaneschi@unisg.ch

***Abstract*—The advent of cloud computing has enabled large-scale availability of *on-demand* computing and storage resources. However, these benefits are not yet at the fingertips of HPC developers: Typical HPC applications use *on-premise* computing resources and rely on static deployment setups, reliable hardware, and rather homogeneous resources. This hinders (partial) execution in the cloud, even though applications could benefit from scaling beyond on-premise resources and from the variety of hardware available in the cloud to speed up execution.**

To address this issue, we orchestrate computationally intensive kernels using a high-level programming language that ensures advanced optimization and improves execution flexibility—enabling hybrid cloud/on-premise HPC deployments. Our approach is based on multitier reactive programming, where distributed code is defined within the same compilation unit and computations are placed explicitly using placement types. We adjust placement based on performance characteristics measured before execution, apply our approach to a shortest vector problem (SVP) solver from cryptanalysis, and evaluate it to be effective.

***Index Terms*—High Performance Computing, Hybrid Cloud Computing, Multitier Programming, Reactive Programming**

Daniel Sokolowski, Jan-Patrick Lehr, Christian Bischof and Guido Salvaneschi, "Leveraging Hybrid Cloud HPC with Multitier Reactive Programming," 2020 IEEE/ACM International Workshop on Interoperability of Supercomputing and Cloud Technologies (SuperCompCloud), Atlanta, GA, USA, 2020, pp. 27-32, doi: 10.1109/SuperCompCloud51944.2020.00010.

I. Introduction

The rise of cloud computing made large amounts of computational resources available. In contrast to on-premise high-performance computing (HPC) clusters, cloud resources are on-demand bookable. They are, however, less reliable, have a slower network, and less predictable performance. Nonetheless, recent studies conclude that HPC applications can benefit from using the cloud in addition to on-premise clusters [1]–[4], especially if they are less communication-intensive, e.g., embarrassingly parallel or tree-structured. In hybrid cloud/on-premise (HyCloud-HPC) runs, applications can scale flexibly and on-demand beyond the limits of on-premise resources.

For efficient HyCloud-HPC execution, implementations must allow various setups differing in scale and resource mix, and support setup changes at run-time. Applications have to combine heterogeneous resources from various environments with independent resource managers, enabling to scale beyond on-premise resources, speed up execution, and meet critical deadlines. Long run times, unknown computational demands, and resource availability require to dynamically add and remove resources to/from the execution, allowing to scale reactively, and tolerate hardware failures. These new flexibility requirements are also relevant for exascale computing [5].

Most HPC applications fall short of these requirements because they are implemented for a fixed hardware configuration and, typically, only support static execution setups. Many applications rely on checkpoint/restart for fault tolerance, e.g., using the Fault Tolerance Interface [6]. More flexible approaches are less common, as their use is prohibitively complex [7]. Also, more flexible constructs in common parallelization paradigms like MPI and OpenMP are practically not used; for MPI see [8]. They maximize efficiency, but force developers to deal with a lot of low-level details, and make flexible implementations impractically hard. Prior work on HyCloud-HPC thus integrates environments at the scheduler level, e.g., Univa's Grid Engine [9] as the successor to Sun's Grid Engine [10]. Another approach combines resource management with workflow management [11]. However, these solutions predominantly support static setups. In particular, they do not allow to easily combine a job's resources from various environments and prohibit to change them dynamically, e.g., to add cloud resources on-the-fly.

In this paper, we simplify the implementation of flexible, dynamically scaling, fault-tolerant HyCloud-HPC applications: We increase the level of abstraction and use multitier reactive programming (MRP), which addresses the demands of more flexible HPC applications, and simplifies their development drastically. Our approach is inspired by the observation that PGAS languages [12] share similarities with multitier languages [13]. The latter have been recently explored in combination with managed runtimes, e.g., Java and Scala, and reactive abstractions, relieving developers from both manual memory management and message passing. Hence, we use MRP for orchestration and native code for compute-intensive kernels, resulting in flexibility at the higher-level and efficiency in the kernels. As a result, our approach allows applications to combine on-premise and cloud resources flexibly. Resources are added or removed dynamically without a unified scheduler, integrating various environments and independent resource management approaches. This enables to scale applications on-demand beyond on-premise available resources to speed up their execution. The contributions of this paper are:

- We identify application requirements for HyCloud-HPC execution, and suggest to meet these with a hybrid implementation model using multitier reactive programming.
- We propose a 3 step approach for HyCloud-HPC: (1) kernel identification and extraction, (2) high-level topology and inter-process communication re-implementation using MRP, and, (3) performance profile based placement.
- We demonstrate our approach with a shortest vector problem (SVP) solver, a key algorithm for evaluating the strength of lattice-based quantum-resistant cryptography.
- We evaluate our SVP solver implementation in on-premise, cloud, and HyCloud-HPC executions, showing improved flexibility and effectiveness of the approach.

Accordingly, Section II shows that MRP helps in HyCloud-HPC applications. Section III explains our approach to enable HPC applications for HyCloud-HPC. Section IV demonstrates it with a SVP solver, evaluated in Section V. Finally, we discuss related work (Section VI) and conclude (Section VII).

II. Hybrid Cloud HPC: A Case for Multitier Reactive Programming

The hybrid on-premise and cloud execution poses new requirements to HPC applications:

MPMD Development: MPI aims to simplify inter-process communication with a model that is especially helpful in *Single Program Multiple Data* (SPMD) applications. For flexible, component-level scalable applications, it is preferable to encapsulate different parts into multiple components combined in a *Multiple Program Multiple Data* (MPMD) fashion. However, developing separate programs distributes the control flow and severely increases the complexity.

Asynchronous Control Flow: For efficient use of highly parallel systems, it is crucial to avoid waiting time. Therefore, task-based work packaging and asynchronous communication are essential.

Flexible Execution Setup: Applications must support various run-time configuration setups and adaptation during execution. This is required to add or remove resources for adaptive scaling on the fly, and to enable fault-tolerance, as cloud resources are unreliable and may fail at any time.

The complexity of low-level libraries, e.g., MPI, limits application developers to meet these requirements. We propose increasing the level of abstraction, identifying that multitier [13] and functional reactive programming [14] abstractions solve the issues, thus simplifying HyCloud-HPC applications:

Joint MPMD Development: Multitier programming enables the joint development of distributed systems within one compilation unit. This makes distributed control flow explicit and does not break the application logic at network boundaries. The communication code is generated and injected by the compiler. Multitier modules [15] allow developers to split-up applications in sensible ways, retaining joint development and correctness guarantees.

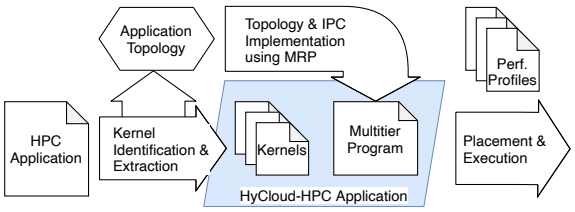


Fig. 1: Approach to enable applications for HyCloud-HPC.

Asynchronous Control Flow: Reactive programming simplifies asynchronous code, accounting for varying execution times and avoiding unnecessary synchronization, and it improves code readability and data flow visibility [16].

Explicit Architecture and Reactivity: Multitier abstractions allow domain-specific architecture definitions, e.g., client-server, star, ring, etc. Unlike in low-level libraries, the topology definitions are embedded into the language and are checked for correctness at compile-time. Reactive abstractions are designed to describe behavior based on change. Together they enable developers to easily implement mechanisms that react to changes in the program state, such as new or disrupted process connections.

Recent implementations of multitier reactive abstractions use managed languages and automated memory management. Unfortunately, managed languages with virtualization and garbage collection are usually less performant than optimized native code. Therefore, we suggest implementing frequently executed kernels in optimized C or C++ and to orchestrate them using multitier reactives. We believe this (1) ensures best performance on a computationally intensive, non-parallelized or shared-memory parallelized level and (2) benefits from the convenience, flexibility, and correctness of MRP on the orchestration level. The combination enables the implementation of flexible and fault-tolerant HyCloud-HPC applications.

III. Breaking Chains: Freeing HPC Applications for HyCloud-HPC Execution

Fig. 1 shows our approach to HyCloud-HPC applications: Starting from an HPC application, we identify computationally intensive sections, and extract these as optimized kernels. Then, we re-implement the application topology, including all inter-process communication (IPC) using MRP. Finally, the resulting application is placed onto available resources guided by the initially obtained performance profiles. We now describe the three steps in more detail, then provide background regarding the used technology, before applying the approach to an SVP solver in Section IV.

a) Kernel Identification and Extraction: We use PIRA [17] to identify computationally intensive kernels based on automated performance measurement and profile analysis. Once the kernel functions are identified, the user, currently manually, extracts the kernel's source code. The user inspects the final generated profile for the call-sites of the identified kernel functions for the required data. The function bodies are inspected for side effects, etc. Given this information, the user

can wrap the required code into a single function that is then to be called through the Java Native Interface (JNI).

b) Architecture and IPC Implementation: We identified ScalaLoci [18] as a suitable multitier language, as it supports programming in a high-level language (Scala) in complex user-defined systems. Moreover, through the integration of REScala [19], ScalaLoci features multitier reactive programming. We propose to isolate each kernel in a separate program, i.e., a ScalaLoci peer, and invoke extracted kernel functions via JNI. The communication between peers can take advantage of reactive signals and event streams. This offers comparatively easy mechanisms to implement support for dynamically connecting and disconnecting peers.

c) Peer Placement: For each peer, an executable program is generated, setting up the network connections in a listener-connector fashion and executing the peer. They are run using the available resource managers, e.g., cluster schedulers or cloud-typical container orchestration platforms. The decision on which resource to deploy which peer is crucial for performance. Kernel efficiency may vary drastically depending on the resource type. Moreover, placement affects data locality and observed communication characteristics. HyCloud-HPC amplifies this challenge due to the vast variety of heterogeneous and specialized hardware in the cloud, while on-premise resources are typically rather homogeneous.

We reuse the performance insights obtained in kernel identification using PIRA. They should be generated for each available resource type and a representative set of configurations. With this, developers manually decide the optimal placement.

In ongoing work, we are developing decentralized placement abstractions. They allow automating placement in a decentralized fashion, ideal to combine multiple independent resource management domains like in HyCloud-HPC. We plan to augment the PIRA performance profiles with runtime monitoring and support placement adaptation at run-time (dynamic placement) to continuously optimize performance.

A. Kernel Identification with PIRA

PIRA automates performance measurement and profile analysis in the build–run–analyze cycle and iteratively adjusts the performance instrumentation towards the target application's kernels. The tool uses a whole-program call graph for its analyses, which it builds in a pre-processing step. PIRA then iteratively performs three stages:

Build: Initially, a vanilla version is built for baseline measurements. Subsequently, Score-P [20] and a custom LLVM-based instrumentation-plugin is used.

Run: The target application is run with the user-provided configuration to record profiling data.

Analyze: A low-overhead instrumentation is generated heuristically based on static source-code features. After profiling runs, also the obtained timing information is considered. Short-running functions are filtered, and long-running functions are expanded in the call graph.

The tool supports two modes for kernel identification: (1) function run time for a single input data set, or, (2) function performance models constructed for multiple input data sets [21]. A function's run time is evaluated using threshold values, computed based on the application's profile. The performance models are constructed empirically using Extra-P [22], which assumes that applications are bulk-synchronous.

Listing 1: ScalaLoci example architecture: A ring of *Manager* peers, each connected to a set of *Worker* peers. On *Manager* we refer to both neighbors as *Manager* or as *Prev* and *Next*.

```
1 @peer type Prev <: { type Tie <: Single[Next] }
2 @peer type Next <: { type Tie <: Single[Prev] }
3 @peer type Manager <: Prev with Next {
4   type Tie <: Single[Prev] with Single[Next] with
5             Multiple[Worker] }
6 @peer type Worker <: { type Tie <: Single[Manager] }
```

B. Multitier Programming with ScalaLoci

ScalaLoci explicitly describes the system architecture by *peers* and *ties*. A peer is a system component to be executed as an individual process and represented as a type. Connections between peers are defined by ties and their arities, i.e., *optional*, *single*, or *multiple*, *cf.* Listing 1. ScalaLoci is suitable for complex architectures, e.g., hexahedral meshes, as in the LULESH proxy application [23].

When MPMD systems are developed jointly, data and functionality have to be assigned to the respective programs. In ScalaLoci, the types of methods and values are augmented with a peer type, obtaining *placement types*, *cf.* Listing 2. Placement types tell the compiler how to distribute the application across its components. Remote value accesses and procedure calls are explicit in the syntax, showing which operations happen locally and which rely on potentially expensive communication. In addition to standard type checking, ScalaLoci verifies at compile-time that all remote accesses are indicated correctly, and adhere to the defined system architecture and privacy annotations. The communication code is injected by the compiler accordingly.

Listing 2: ScalaLoci placement example: part (1) and sum (2) are *placed values*. sum is only accessible locally; part can be retrieved from connected peers. doWork (3–5) and updateSum (6–7) are methods on worker and manager. Manager calls doWork on each worker (8). It updates part (4) and calls updateSum on the manager (5), which obtains and sums up part (7).

```
1 var part: Int on Worker = placed { 0 }
2 val sum: Int localOn Manager = placed { 0 }
3 def doWork(): Unit on Worker = placed {
4   part = // Some processing returning an integer
5   remote[Manager] call updateSum() }
6 def updateSum(): Unit on Manager = placed {
7   sum = part asLocalFromAll foldLeft(0)(_+_._1) }
8 placed[Manager] { remote[Worker] call doWork() }
```

C. Reactive Programming with REScala

REScala combines reactive values with an event system. It provides data types for *events* and *signals*. Events allow

Listing 3: Reactives example: part (1) and sum (2–3) are signals. sum is updated automatically when any part changes. When the workTick event (4) fires (8), each worker updates its part (5–7).

```
1 val part: Var[Int] on Worker = placed { Var(0) }
2 val sum: Signal[Int] localOn Manager = placed {
3     Signal { part asLocalFromAll foldLeft(0)(_+_._1) } }
4 val workTick: Evt[Unit] on Manager = placed { Evt[Unit]() }
5 placed[Worker] {
6     workTick asLocal observe {
7         part.set( /* Some processing returning an integer */ ) } }
8 placed[Manager] { workTick.fire(()) }
```

imperative change propagation, while signals update themselves and automatically propagate their change when one of their functional dependencies changes. Both abstractions simplify the implementation of encapsulated and composable code, with clear data flow visualization [16], [24]. Listing 3 re-implements Listing 2 using REScala reactives.

IV. Case Study: Shortest Vector Problem Solver

To demonstrate our approach, we apply it to a non-parallelized version of p3Enum [25], a shortest vector problem (SVP) solver. This code contains all optimizations of p3Enum, except parallelization, and thus does not (1) process multiple bases in parallel, and (2) parallelize the enumeration.

SVP is a so far quantum-computer safe problem in lattice cryptography and searches for the shortest non-zero vector, given an n-dimensional basis for a D-dimensional lattice. Burger et al. [25] provide a thorough explanation and propose p3enum. In a loop, it generates a random basis, sequentially applies two basis reductions (the second with pruning and early exit), before enumerating a pruned search tree. It terminates once a vector is found that is shorter than a predefined length. The algorithm is non-deterministic due to the random bases and extreme pruning. Thus, it typically requires multiple trials for a solution. The processing of a basis is independent, making the program tree-structured and thus suitable for HyCloud-HPC [2].

A. Kernel Identification and Extraction

We apply PIRA with four iterations, each running the SVP solver 25 times with a reasonably small problem to identify the application's kernels. Manual validation proved four iterations reasonable. 25 repetitions is a trade-off between accuracy and time to completion, addressing the random nature of the algorithm. The randomness also requires to run PIRA in the first mode, i.e., kernel identification based on runtime thresholds. The model-based kernel identification fails, as randomness violates Extra-P's bulk-synchronous algorithm assumption.

PIRA identifies two main kernels for the SVP solver: (1) a third party library function used with different parameters for both BKZ basis reductions and (2) the enumeration function, which performs a tree search and applies the extreme pruning strategy. Both functions are also manually validated to be the main contributors to application run time.

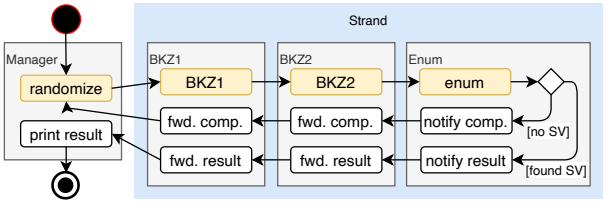


Fig. 2: Overview of the ScalaLoci based SVP solver.

Listing 4: SVP solver architecture: A central *Manager* is connected to multiple strands, each subdivided into three peers *BKZ1*, *BKZ2*, and *Enum*, which wrap the individual kernels.

```
1 @peer type Manager <: { type Tie <: Multiple[BKZ1] }
2 @peer type BKZ1 <: {
3   type Tie <: Single[Manager] with Single[BKZ2] }
4 @peer type BKZ2 <: {
5   type Tie <: Single[BKZ1] with Single[Enum] }
6 @peer type Enum <: { type Tie <: Single[BKZ2] }
```

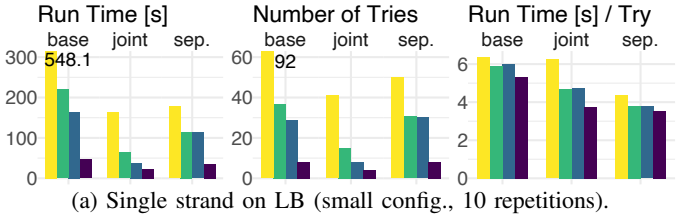
We extract and wrap the kernels manually to be callable via the JNI. The two BKZ reductions are wrapped in two functions, *BKZ1* and *BKZ2*, that only vary in their parameters. The enumeration is wrapped as the *enum* function. Moreover, we want to reuse the basis randomization and wrap it as *randomize* function. Passing values via JNI involves serialization, copy, and de-serialization, and can cause significant run time overhead for big data structures like the involved matrices. We implement a matrix data structure that resides in a shared buffer and thus is directly accessible by Scala and the kernels.

B. Architecture and IPC Implementation

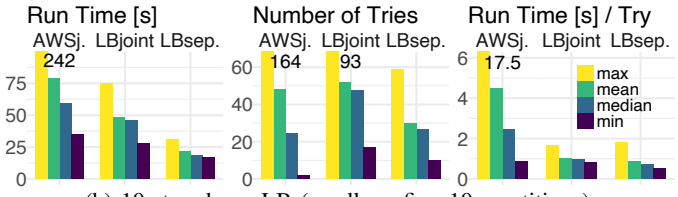
Fig. 2 shows our SVP solver implementation with the architecture given in Listing 4. For each strand, the manager generates random bases, which get processed alongside the strand's peers. After enumeration, the strand either signals completion to receive a new basis or provides the found shortest vector, terminating the entire application.

Listing 5: Flexible task assignment: The assignedTasks signal (1–5) holds the task assignment state; the number of assigned tasks per strand. It updates when a strand connects or disconnects, and increments when a strand reports a task completion. startedTasks (6–9) provides each strand an individual event stream, firing a new task whenever the strand's assignment changes. The BKZ1 peer accesses its tasks via asLocal (11) and provides its results as event stream (10), observed by BKZ2.

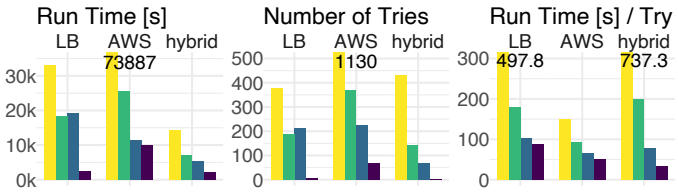
```
1 val assignedTasks: Signal[AssignmentState] on Manager = placed {
2   Events.foldAll(AssignmentState(Map.empty)) { state => Seq(
3     remote[BKZ1].joined >> state.addRemote,
4     remote[BKZ1].left >> state.removeRemote,
5     completedTasks >> state.assignNextTaskForStrand) } }
6 val startedTasks = on[Manager] sbj { strand: Remote[BKZ1] => {
7   assignedTasks.map(_.state.get(strand.hashCode)).changed
8     .map { _.map { taskId =>
9       (strand.hashCode(), taskId, freshTask( /* ... */ )) } }.flatten } }
10 val bkz1Results = on[BKZ1] {
11   startedTasks.asLocal.map(task => /* bkz1 */ ) }
```



(a) Single strand on LB (small config., 10 repetitions).



(b) 10 strands on LB (small config., 10 repetitions).



(c) 20 strands on AWS and LB, and hybrid (big config., 5 repetitions).

Fig. 3: Run time measurements and related number of tries.

In HyCloud-HPC, strands need to be able to join and leave the computation, i.e., connect and disconnect to/from the manager independently. Listing 5 shows all code required for this flexibility. This allows flexibility to combine independently scheduled resources on-premise and in the cloud, and ensures resistance to strands' fault.

C. Peer Placement

We generate a dockerized program for each peer and use TCP for communication. We target an on-premise cluster and AWS Fargate. The performed PIRA measurements indicate similar kernel performance in both environments.

The manager peer has to be reachable for every strand. Therefore, we execute it in the cloud due to network policies. The pipeline fashion of the strands requires careful placement; providing dedicated resources to each peer will only saturate the slowest peer's resources. This requires more intelligent placement, e.g., holistic or dynamic placement. A naive alternative is to run all peers of a strand concurrently within a shared resource. We investigate this and more in the following.

V. Performance Evaluation

We evaluate our implementation on the Lichtenberg HPC system (LB), equipped with Intel Xeon E5-2680 v3 Processors, and on AWS Fargate, running the software on AWS EC2. We test two setups *separate* and *joint*, running every strand's peers in three separate processes or in a single process within different threads. The manager is a dedicated process. As the baseline, we use the non-parallelized version of p3Enum [25]. On LB and AWS we assign one dedicated CPU core and 2 GB memory[1] to each process; manager processes on AWS have

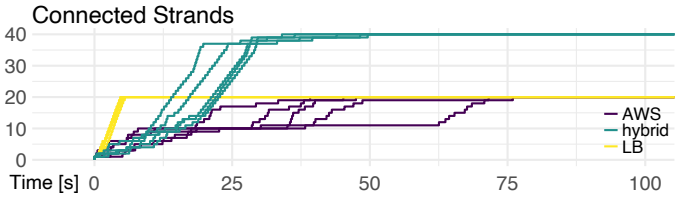


Fig. 4: Connected strands over time for experiments Fig. 3c.

2 CPU cores and 4 GB[1]. We measure two configurations: *small* is of dimension 78 and usually solved within minutes, *big* is of dimension 100. We do not report a baseline for big because it timed out after 24 hours in 18 out of 20 LB runs.

We report the minimum, maximum, median, and mean of the run time and the number of enumerated bases (tries) for various deployments in Fig. 3. The number of tries varies drastically due to the program randomness, and so does the run time, requiring more repetitions for statistically stable results. We use their quotient for more reliable comparisons.

Fig. 3a shows that our implementation's performance is close to the baseline. The slightly reduced run time per try for the joint setup is caused by the small share of work performed by the manager running in a separate process, which for the baseline is included in the strand's process. The strand of the separate setup required triple the resources than in the joint setup, but its performance is only slightly better due to the pipeline effect described in Subsection IV-C. The difference in the joint and separate setup measurements in Fig. 3b also confirms a significant waste of resources. Accordingly, we only consider the more efficient joint setup in the following.

Fig. 3b and Fig. 3c confirm that the SVP solver performs similarly well on AWS compared to LB. However, for the shorter measurements in Fig. 3b, the performance looks worse. This effect is mainly caused by the less coordinated startup of the strands on AWS, causing higher delays between their startup and joining. This is negligible for the longer measurements in Fig. 3c but is visible in Fig. 4, which shows the number of connected strands in the first 100 seconds after startup for the big configuration measurements. LB strands connect almost simultaneously due to coordinated startup, while the AWS containers are deployed separately with varying delays. This confirms the worse predictability regarding availability and performance of the used commodity cloud resources compared to on-premise clusters, but also shows that our HyCloud-HPC approach can leverage these effectively.

In Fig. 3c, we compare longer running measurements on LB and AWS and join the resources of both environments then for hybrid measurements. Combining the on-premise cluster resources and the cloud improves the run time significantly, underlining our approach's effectiveness.

VI. Related Work

a) PGAS/APGAS Languages: PGAS is a parallel programming model, allowing processes to access data from each

[1]These are the lowest memory assignments possible for the related amounts of CPU cores at AWS Fargate; our software requires far less.

other through abstractions emulating a shared memory space. Remote access requires special syntax, but communication is transparent. De Wael et al. [12] classify PGAS languages. IBM's X10 [26] distributes arrays over *places*, partitions of the global address space. *Asynchronous activities* encapsulate processing and data transfer. Chapel [27] has a similar data distribution model with both data and task parallelism, and also other distributed data structures than arrays. Fortress [28] features constructs for explicit and implicit parallelism.

b) Multitier Programming: Multitier languages remove the separation of code between communicating processes. Weisenburger et al. [13] provide an overview of existing solutions. Most approaches focus on the Web and client-server architecture. In contrast, ML5 [29] and ScalaLoci support *generic* software architectures.

VII. Conclusion

In this paper, we propose a new hybrid programming model for HyCloud-HPC applications: orchestrating optimized, native kernels with multitier reactive programming. We describe our approach to increase the flexibility of existing HPC applications, enabling hybrid on-premise and cloud deployments. We apply the approach to a shortest vector problem solver and evaluate its performance. We show that MRP is suitable to efficiently combine on-premise and cloud resources, even allowing us to add resources at run-time dynamically—to speed up HPC applications and meet critical deadlines flexibly.

Acknowledgments

This work has been co-funded by the German Research Foundation (DFG, No. 322196540 and 383964710, SFB 1053 and 1119), by the Hessian LOEWE initiative (emergenCITY and Software-Factory 4.0), and by the German Federal Ministry of Education and Research (BMBF) and the Hessian Ministry of Higher Education, Research and the Arts (HMKW) within their joint support of the National Research Center for Applied Cybersecurity ATHENE. Calculations were performed on the Lichtenberg HPC system at TU Darmstadt.

References

[1] M. A. S. Netto, R. N. Calheiros, E. R. Rodrigues, R. L. F. Cunha, and R. Buyya, "HPC cloud for scientific and business applications: Taxonomy, vision, and research challenges," *ACM Comput. Surv.*, vol. 51, Jan. 2018.

[2] A. Gupta and D. Milojicic, "Evaluation of HPC applications on cloud," in *2011 Sixth Open Cirrus Summit*, pp. 22–26, 2011.

[3] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B. Lee, V. March, D. Milojicic, and C. H. Suen, "Evaluating and improving the performance and scheduling of hpc applications in cloud," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 307–321, 2016.

[4] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn, "Case study for running HPC applications in public clouds," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, (New York, NY, USA), p. 395–401, Association for Computing Machinery, 2010.

[5] A. Geist and D. A. Reed, "A survey of high-performance computing scaling challenges," *The International Journal of High Performance Computing Applications*, vol. 31, no. 1, pp. 104–113, 2017.

[6] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High performance fault tolerance interface for hybrid systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, ACM, 2011.

[7] G. Georgakoudis, L. Guo, and I. Laguna, "Reinit^{++}: Evaluating the performance of global-restart recovery methods for MPI fault tolerance," in *High Performance Computing* (P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, eds.), (Cham), pp. 536–554, Springer International Publishing, 2020.

[8] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of MPI usage in open-source HPC applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[9] Univa Corporation, "Univa Grid Engine." https://www.univa.com/products/univa-grid-engine.php, 2020. [Online; accessed 08/03/2020].

[10] W. Gentzsch, "Sun Grid Engine: towards creating a compute power grid," in *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 35–36, 2001.

[11] G. Mateescu, W. Gentzsch, and C. J. Ribbens, "Hybrid computing – where HPC meets grid and cloud computing," *Future Generation Computer Systems*, vol. 27, no. 5, pp. 440 – 453, 2011.

[12] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter, "Partitioned global address space languages," *ACM Comput. Surv.*, vol. 47, pp. 62:1–62:27, May 2015.

[13] P. Weisenburger, J. Wirth, and G. Salvaneschi, "A survey of multitier programming," *ACM Comput. Surv.*

[14] C. Elliott and P. Hudak, "Functional reactive animation," in *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, (New York, NY, USA), p. 263–273, Association for Computing Machinery, 1997.

[15] P. Weisenburger and G. Salvaneschi, "Multitier modules," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)* (A. F. Donaldson, ed.), vol. 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 3:1–3:29, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[16] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini, "On the positive effect of reactive programming on software comprehension: An empirical study," *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1125–1143, 2017.

[17] J.-P. Lehr, A. Hück, and C. Bischof, "PIRA: Performance instrumentation refinement automation," in *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*, AI-SEPS 2018, (New York, NY, USA), p. 1–10, Association for Computing Machinery, 2018.

[18] P. Weisenburger, M. Köhler, and G. Salvaneschi, "Distributed system development with ScalaLoci," *Proc. ACM Program. Lang.*, vol. 2, Oct. 2018.

[19] G. Salvaneschi, G. Hintz, and M. Mezini, "REScala: Bridging between object-oriented and functional style in reactive applications," in *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, (New York, NY, USA), p. 25–36, Association for Computing Machinery, 2014.

[20] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, (Berlin, Heidelberg), pp. 79–91, Springer Berlin Heidelberg, 2012.

[21] J.-P. Lehr, A. Calotoiu, C. Bischof, and F. Wolf, "Automatic instrumentation refinement for empirical performance modeling," in *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pp. 40–47, 2019.

[22] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC13), Denver, CO, USA*, pp. 1–12, ACM, November 2013.

[23] D. Sokolowski, P. Martens, and G. Salvaneschi, "Multitier reactive programming in high performance computing." 6th Workshop on Reactive and Event-based Languages & Systems, 2019.

[24] G. Salvaneschi and M. Mezini, "Debugging for reactive programming," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, (New York, NY, USA), p. 796–807, Association for Computing Machinery, 2016.

[25] M. Burger, C. Bischof, and J. Krämer, "p3Enum: A new parameterizable and shared-memory parallelized shortest vector problem solver," in *Computational Science – ICCS 2019*, (Cham), pp. 535–542, Springer International Publishing, 2019.

[26] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, pp. 519–538, Oct. 2005.

[27] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, Aug. 2007.

[28] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, and S. Tobin-Hochstadt, "The Fortress Language Specification," tech. rep., Sun Microsystems, Inc., March 2008. Version 1.0.

[29] T. Murphy, VII., K. Crary, and R. Harper, "Type-safe distributed programming with ML5," in *Proceedings of the 3rd Conference on Trustworthy Global Computing*, TGC '07, (Berlin, Heidelberg), Springer-Verlag, 2008.