An Empirical Study of High Performance Computing (HPC) Performance Bugs

***Abstract*—Performance efficiency and scalability are the major design goals for high performance computing (HPC) applications. However, it is challenging to achieve high efficiency and scalability for such applications due to complex underlying hardware architecture, inefficient algorithm implementation, suboptimal code generation by the compilers, inefficient parallelization, and so on. As a result, the HPC community spends a significant effort detecting and fixing the performance bugs frequently appearing in scientific applications. However, it is important to accumulate the experience to guide the scientific software engineering community to write performance-efficient code.**

In this paper, we investigate open-source HPC applications to categorize the performance bugs and their fixes and measure the programmer's effort and experience to fix them. For this purpose, we first perform a large-scale empirical analysis on 1729 HPC performance commits collected from 23 real-world projects. Through our manual analysis, we identify 186 performance issues from these projects. Furthermore, we study the root cause of these performance issues and generate a performance bug taxonomy for HPC applications. Our analysis identifies that inefficient algorithm implementation (39.3%), inefficient code for target micro-architecture (31.2%), and missing parallelism and inefficient parallelization (14.5%) are the top three most prevalent categories of performance issues for HPC applications. Additionally, to understand how the performance bugs are fixed, we analyze the performance fix commits and categorize them into eight performance fix types. We further measure the time it takes to discover a performance bug and the developer's efforts and expertise required to fix them. The analysis identified that it's difficult to localize performance inefficiencies, and once localized, fixes are complicated with a median patch size (LOC) of 35 lines and are mostly fixed by experienced developers.

***Index Terms*—Empirical Study, HPC, Performance Optimization**

I. Introduction

High performance computing (HPC) enables the scientific and engineering community to solve large-scale computational problems. These computational problems have a wide range of applications, such as climate modeling and forecasting [1], cancer research [2], drug discovery [3], nuclear energy [4], national security [5], automotive design [6] and so on. However, unlike other applications, HPC applications require processing a large volume of data and solving complex computational problems at high speed. For this purpose, the HPC applications execute the computational problems on massively parallel architectures. However, writing a performance-efficient and scalable HPC application is a challenging task.

One of the challenges in writing an efficient HPC application arises due to complex underlying hardware architecture. HPC applications are targeted to run on multi-core CPU architectures, GPUs, and related accelerators. However, these micro-architectures have complex memory, and execution model [7]. Inefficient implementation of computational kernels for the target micro-architecture incur significant performance overhead [8]. Additionally, compilers often generate suboptimal code resulting in low performance [9]. Unnecessary and redundant computation is another source of inefficiency found in HPC applications [10]. As a result, the developers put significant effort into identifying and fixing the performance bugs in the HPC applications. Nevertheless, it is important to accumulate the lessons learned in fixing the performance bugs, inform the HPC developers about the common performance pitfalls, and further guide the software engineering researcher to develop novel tool support.

Previously, many empirical studies were conducted in various domains to understand the fundamental root causes of performance bugs [11]–[13] and how developers dealt with them [14]. However, to the best of our knowledge, no empirical study on HPC performance bugs has been undertaken, even though it is crucial for overall software quality in the HPC domain.

In this paper, we perform an empirical study of HPC performance commits to bridge the knowledge gap between the HPC application developer and the real-world HPC performance bugs. Apart from that, we also analyzed how complicated HPC performance bugs are to identify and fix, as well as the developers' experience to fix the performance bugs. In particular, we aim to answer the following research questions:

- **RQ1 (Causes of inefficiency):** What are the common types of performance bugs in HPC applications?
- **RQ2 (Optimizing performance):** How are performance bugs optimized in the context of HPC applications?
- **RQ3 (Localization and bug fixing effort):** How difficult is it to localize performance inefficiencies in HPC applications? How complicated is it to fix those performance bugs?
- **RQ4 (Developer's expertise):** Does domain expertise play a role to fix the performance inefficiency? How skilled are the HPC application developers to solve the performance bugs?

However, an empirical study on HPC application performance commits is challenging. First, the performance commits often lack adequate information to identify the root cause of performance bugs. Second, performance commits often involve multiple bug fixes in a large number of source files. As a result, it becomes challenging to identify the root cause of the bug and characterize the performance fixes.

In this study, we first carefully select 23 open-source HPC projects with more than 1000 commits and 20 stars. Through repository mining on these selected projects, we

identified 1729 probable performance commits. To filter false-positive, we perform a manual analysis and determined 186 performance commits. We then manually investigate each performance commit to answer the above-mentioned research questions.

By answering the above questions, we aim to understand common characteristics of performance bugs in HPC applications. From our analysis, we identified that HPC performance bugs, such as performance portability, inefficient compiler-generated code, and so on, are nontrivial and require a good understanding of underlying architecture, compiler optimization, programming language, and the algorithm of the computational kernel. Based on our analysis, we created an HPC performance bug taxonomy with 10 main categories of performance bugs. Besides the performance bug understanding, our finding also comes up with a performance optimization catalog that can assist developers in fixing the performance bug more effectively and efficiently. Finally, our analysis also pointed out that HPC performance issues are difficult to localize and require more complicated solutions and developer experience. The finding constitutes a call for action for scientific application developers and researchers to develop performance analysis tools to detect inefficiencies, a recommendation system for the developers for optimization, and automatic generation of performance fixes.

In summary, our study makes the following contributions:

- We construct a data set of HPC performance inefficiencies that we used for our analysis. The dataset can be further utilized in future research on developing tools and techniques for detecting and fixing HPC performance bugs.
- We developed a taxonomy of HPC performance bugs or inefficiencies that can assist developers in being aware of such inefficiencies.
- Our performance optimization catalog can be utilized by developers as a guideline as well as to help researchers generate recommendation systems to fix performance issues.
- Finally, our analysis of HPC performance bug fix complexity and developer experience will help us understand the characteristics of HPC performance bugs and provide guidance to develop advanced tools for automatic detection and fixing performance bugs.

The remainder of this paper is organized as follows: Section II describes the data preparation process and research methodology of our empirical study. A comprehensive evaluation of all four research questions is presented in Section III, IV, V, and VI, respectively. Section VII, captures any threats to the validity of our study. Section VIII discusses the research findings. Finally, Section IX includes the related works to our study, and Section X draws a conclusion for this paper.

Replication: Our replication package is available at https://figshare.com/s/00c24aae3177e45db7ab for further study.

TABLE I
SUMMARY OF THE HPC APPLICATION DATASET USED IN THE STUDY

Project	# Commits	# Stars	Domain
mlpack [15]	28,204	4174	Machine Learning
LAMMPS [16]	31,659	1553	Molecular Dynamics
OpenBLAS [17]	6,600	4932	Linear Algebra
CGAL [18]	101,678	3580	Computational Geometry
FFTW3 [19]	3,138	2261	Fourier Transformation
ArrayFire [20]	5,841	3996	Tensor Library
OpenFOAM [21]	6,102	1032	Comp. Fluid Dynamics
OpenMM [22]	6,873	1099	Molecular Simulation
MFEM [23]	16,204	1097	Finite Element Methods
Kokkos [24]	9,825	1203	Programming Model
GROMACS [25]	21,256	486	Molecular Dynamics
CasADi [26]	13,882	1118	Nonlinear Optimization
libMesh [27]	22,770	543	Finite Element Library
TileDB [28]	3,739	1438	Multi Dimensional Array
QMCPACK [29]	23,785	217	Quantum Monte Carlo
preCICE [30]	4,386	493	Multi-Physics Simulator
HYPRE [31]	12,061	443	Linear Algebra
lattice-QUDA [32]	12,106	224	Quantum Chromodynamics
Ginkgo [33]	5,431	261	Linear Algebra
GlobalArrays [34]	5,224	80	Programming Model
GOMC [35]	2890	46	Molecular Dynamics
Qbox [36]	1,851	31	Molecular Dynamics
Elemental [37]	3,773	490	Linear Algebra

II. METHODOLOGY

This section explains how we prepare the dataset for HPC applications and examine the performance bugs to understand the root causes of performance bugs and classify them into the appropriate category. Figure 1 shows the overview of our research approach.

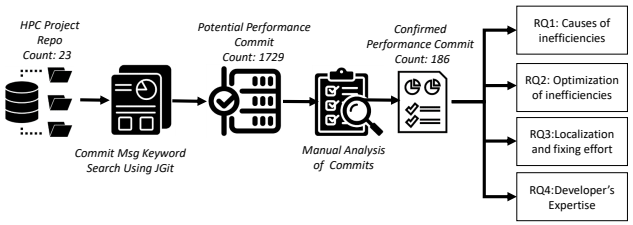

Fig. 1. Overview of Research Approach

A. Dataset Preparation

Since our study focuses on HPC applications, which is a broad domain, it was not practical to analyze all the open-source HPC applications. Therefore, we focused on 23 applications to conduct our study, and list of the applications is shown in Table I. We selected these applications which have more than 1000 commit counts and 20 stars in the GitHub repositories. These applications have existed for many years and are well-maintained, indicated by commit count and popularity indicated by star count. These projects cover a variety of important aspects of HPC domain, such as atomic/molecular parallel simulations of solid-state materials (LAMMPS), fast, flexible machine learning library in C++ (MLPACK), efficient and reliable algorithms in computational geometry (CGAL), computational fluid dynamcis (CFD) software package (OpenFOAM), and performance portable programming model (Kokkos).

B. Collecting candidate performance commits

To study the performance bugs and their fixes, we automatically mine performance-related commits from our curated list of HPC repositories by using a tool built upon

JGit [38]. To identify performance-related commits, we adopt a keyword-based searching approach similar to prior work [11] [39] [40] [41]. Specifically, our tool returns all the GitHub commits from these repositories that contain a performance-related keyword (*performance*, *speed up*, *accelerate*, *fast*, *efficient*, *optimize*, etc.) in their title or description. This keyword-based filtering resulted in only selecting 1729 candidates of performance-related commits. To filter out false positives from these candidate commits, we performed a manual analysis explained in the following subsection.

C. Manual analysis of performance commits

Similar to prior studies [11], [12], [40], [42], [43], we manually categorize HPC performance bugs by their root causes. Specifically, we performed an iterative manual analysis to analyze and categorize the HPC performance bugs that lasted over seven months. Overall, we invested over 1900 man-hours in manually examining the potential performance commits, and extracting detailed information about these performance issues and optimizations to obtain the initial taxonomy. The iterations are as follows:

Iteration I: First, all the authors analyzed the 1729 candidate commits separately to identify whether the commit was performance-related. For this purpose, the authors evaluated the commit messages, bug patches, developer's comments, and pull request discussion to find any references to performance improvement; and tagged them as performance or non-performance. However, from the source code history, it is difficult to differentiate whether it is an actual performance bug or incremental implementation and improvement. While doing manual analysis, we eliminated commits with more than 20 source file changes to avoid the risk of adding noise to the labeled data. These commits can be characterized as tangled commits [44], which makes it difficult to determine the true objective of the code. To finalize this initial iteration, we had a consensual agreement meeting among all authors to further discuss the process and resolve conflicts. With this iteration, we confirmed 186 performance bugs, similar in size to other existing studies [11], [40] that required manual inspection.

Iteration II: We further independently reviewed, analyzed, and labeled each of the 186 identified performance bugs based on the root cause of those bugs. To obtain a meaningful domain-specific taxonomy, we consider the construct of underlying microprocessor architecture (CPU vs. GPU), memory subsystem (Cache vs. global memory vs. register), aspects of parallel and concurrent computing, parallel programming models (OpenMP, CUDA, Kokkos), and compiler optimization techniques. In the end, we again had a reconciliation meeting to discuss the differences. Following an in-depth inspection and discussion, we agreed on a common label for each of the 186 performance bugs.

For both iterations, we calculated the Fleiss' kappa value between the observations of the authors for measuring inter-rater agreement. The score for classifying the commits into performance or non-performance categories is *0.64*. The Fleiss' kappa value on root-cause classification is *0.72*. Both values show *substantial agreement* (0.61-0.80:implies substantial agreement) on performance bug detection and categorization.

III. RQ1: Causes of Inefficiency

We studied 186 performance commits from 23 open-source HPC projects and categorized them into 10 major categories. Figure 2 shows the detailed taxonomy of the HPC performance inefficiencies. In the rest of the section, we discuss each category of inefficiency and its root causes.

A. Inefficient algorithm, data structure, computational kernel, and their implementation (IAD)

We found that 73 out of 186 commits (39.3%) fix the performance issues that originated from the poor algorithm and data structure design and implementation. These inefficiencies include using computationally expensive operations (29), redundant operations (16), unnecessary operations (13), use of inefficient data structure (9), repeated function calls (3), and use of improper data types (3).

1) Computationally expensive operation: Applications with this inefficiency pattern perform a set of operations that incurs high-computational overhead at the runtime. Some sources of these expensive operations include using an inefficient algorithm or computational kernel, expensive runtime evaluation instead of compile-time evaluation, expensive data-structure traversals, and higher-precision arithmetic operations. However, from the commits, we observe that this computational overhead can be bypassed on many occasions using techniques such as compile-time evaluation, algorithmic strength reduction or approximation, caching, and reduced precision arithmetic.

```
void bilateralKernel(
...
-gauss2d[ly*window_size+lx] = exp( ((x*x) + (y*y))
-    /(-2.f * variance_space));
+gauss2d[ly*window_size+lx] = __expf(((x*x) + (y*y))
+  / variance_space);
```

Listing 1. CUDA's `exp()` function is computationally expensive than `__expf()`.

For instance, the Listing 1 depicts the partial commit, ArrayFire-d0d87ab from ArrayFire. According to the commit, ArrayFire's bilateral filter kernel uses CUDA's [45] exponential function `exp()`. The `exp()` implements a double precision exponential function which is computationally expensive. On the other hand, CUDA implements a less computationally expensive alternative, `__expf()`, with fewer native instructions. However, `__expf()` suffers from accuracy loss compared to `exp()`. Since the algorithm can tolerate the loss of accuracy, using computationally expensive `exp()` instead of `__expf()` is causing performance inefficiency.

2) Redundant operations: This category of inefficiencies arises due to the iterative execution of an operation while the operation itself is invariant of the iteration, making iterative execution redundant. MFEM-2c9ee23 commit of MFEM reports such inefficiency. Listing 2 depicts the relevant changes of the commit. According to the commit, the MFEM's `MultTranspose` function performs a multiplication of a

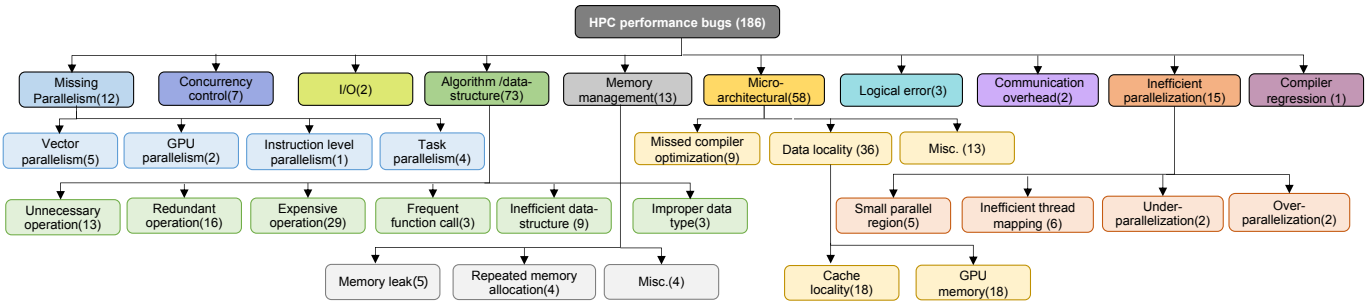

Fig. 2. Taxonomy of HPC Performance Bugs

dense matrix `loc_prol` with vectors and transposed in a hot loop via the `MultTranspose` function call. However, matrix `loc_prol` does not change over the iteration. As a result, based on the multiplication property of transpose, the transpose operation on each iteration is redundant and can be carried out outside the loop.

```
void OrderTransferOperator::MultTranspose(const
    ↪ Vector &x, Vector &y) const
...
-subY.SetSize(loc_prol.Width());
+loc_prol.Transpose();
+subY.SetSize(loc_prol.Height());
...
for (int vd = 0; vd < vdim; vd++){
    ...
-     loc_prol.MultTranspose(subX, subY);
+     loc_prol.Mult(subX, subY);
```

Listing 2. Redundant transpose operation in loop.

3) Unnecessary operations: In such cases, the inefficient code performs a computation or data access whose results were never used in the algorithm. For instance, OpenFOAM-dev-91e84b9 commit identifies such inefficiency. The Listing 3 depicts the partial commit where OpenFOAM-dev's `stochasticCollision` algorithm checks whether all parcels collide regardless of their associativity to cells. However, since parcels belonging to different cells will never collide, many collision checks are unnecessary, resulting in poor performance of the algorithm.

```
void Foam::ORourkeCollision<CloudType>::collide(
    ↪ const scalar dt)
-forAllIter(typename CloudType,this->owner(),iter1){
-    forAllIter(typename CloudType,this->owner(),
    ↪ iter2)
+// Create a list of parcels in each cell
+List<DynamicList<parcelType*>> pInCell(this->owner
    ↪ ().mesh().nCells());
+    // iterate the cell list
+    for (label celli=0;celli<this->owner().mesh().
    ↪ nCells(); celli++){
+        // compare parcels within the cell
+        forAll(pInCell[celli], i){
+           for (label j=i+1; j<pInCell[celli].size
    ↪ (); j++){
            bool massChanged = collideParcels(dt, p1,
                ↪  p2, m1, m2);
```

Listing 3. Unnecessary traversal in collision detection

4) Inefficient data-structure: We found that 7 out 9 commits of this category fix performance bugs originate from choosing an inefficient data structure library. For instance, TileDB-d51b082 commit reported that the read query implemented using `forward_list` from C++'s Standard Template Library (STL) [46] is performing slower. The `forward_list` implements a linked list data structure. As a result, traversing `forward_list` results in a poor data locality compared to other cache-efficient data structures such as STL's `vector`.

B. Inefficient code for underlying micro-architecture (MA)

We found 58 commits (31.8%) to fix the performance bugs that originated from inefficient code for underlying hardware micro-architecture. High-performance computing application's performance is susceptible to memory access latency, hazards in the instruction pipeline, branch divergence, and many other micro-architectural nuances.

1) Inefficiency due to memory/data locality: The principle of locality heavily influences modern processors design choices. According to the principle, applications tend to access a small set of data repeatedly. Modern processors implement hierarchical data storage to reduce memory latency of these frequently accessed data. Register memory, the fastest data storage, resides next to the processor. The processors implement a limited number of registers. As a result, efficient utilization of the registers is important for high performance. Cache memories residing next to registers can contain more data at a relatively higher access latency than the registers. Again, processors implement multi-level cache, where data access from the far-caches incurs higher latency. Modern GPU architectures further implement various types of cache-memory targeting application's memory access patterns. Due to the complexity of the memory hierarchy of modern processors, it is often challenging to write efficient code. Our empirical study finds 36 commits (19.4%) that fix these various memory/data locality challenges.

a) Inefficient cache access: We found 18 commits fix inefficiencies arise due to poor cache utilization resulting in high memory access latency.

One such example is false sharing. False sharing occurs when two or more threads run on two or more CPU cores and access two different memory addresses in the same cache line. Such access causes cache line eviction from one CPU core and vice versa and increases cache miss. For example, the Kokkos-75fd8bc commit reported a false sharing in it's random number pools which are shared by multiple threads. Since the array of

elements per thread is small, they share cache line and causes false-sharing. The commit fixes the issue by padding the array so that elements in the array reside in the separate cache line. The commit reported 200x improvement for 20 threads on the Intel Skylake machine running the random number test case.

Another example of such inefficiency is non-linear data access that causes poor spatial locality. Spatial locality, a kind of data locality, states that given data access, nearby memory locations will be referenced soon. To realize the property, a cache-line, the basic building block of a cache, holds consecutive memory addresses. Any high-performance application traverses memory non-linearly will fail to utilize the cache-line locality and incur significant memory latency. For instance, the cp2k-7b34ac6 commit identifies that cp2k's `Gamma_P_ia` had poor spatial locality due to its access pattern.

b) Inefficient GPU-memory access: Modern GPU processors implement multiple device memory spaces such as global, local, shared, constant, texture memory, and register files [7]. Depending on the data-access pattern, each type of memory has advantages and disadvantages. From our empirical study, we observe that 18 commits (9.7%) fix the inefficient use of GPU memory.

We found 6 GPU memory-related commits using high-latency global memory. All threads can access data stored in global memory in the GPU. However, global memory has more access latency than shared, constant, and textured memory. As a result, it is important to reduce global memory use when possible. For example, the GROMACS-8aa14d1 commit identifies that GROMACS' bonded kernel performs a reduction operation across all the threads using atomic instruction on global memory. However, performing the reduction operation on high-latency global memory is unnecessary in many cases. For example, the threads residing in the same warp can perform the reduction operations in low-latency shared memory, avoiding expensive global memory access. However, the implementation fails to realize this property and suffers from a performance bottleneck.

Due to limited space in register files, efficient register use is important to reduce register spilling. We found three commits that fix GPU-register spilling. For instance, the QUDA-b7857af commit identifies register spilling in Lattice-QUDA's `dslash4_domain_Wall_4d` kernel. The kernel used registers to store coefficients. However, since the coefficients are not updated during kernel execution, keeping them in the registers instead of constant memory increases register pressure, resulting in register spilling.

We found two commits that fix an inefficient host-device communication. Data movement between the host CPU and the GPU causes significant performance overhead. The CUDA `cudaMemcpy` copies data between the host and device synchronously. Due to this blocking call, it fails to overlap the computation and communication and thus fails to mask communication latency. For instance, the ginkgo-154aafb commit identifies that ginkgo's `residual_norm_reduction` performs a synchronous data copy using `cudaMemcpy`. However, these high latency blocking function calls can be avoided by copying the data asynchronously during kernel launch time.

If the source or destination in host memory is not allocated as pinned, the host-device memory copy will require to transfer the data to a pinned memory at first. This causes an extra data copy and causes significant performance overhead. One such example is identified in the OpenMM-926e7b9 commit, where OpenMM's `CudaCalcAmoebaVdwForceKernel` transfers the context parameter `VdwLambda` from host to device using unpinned host memory.

2) Sub-optimal code generation by compiler: On several occasions, the compiler fails to statically reason about the most opportunities of a source code. We found 9 commits (4.8%) fixed the sub-optimal code generated by the compiler.

a) Loop unrolling: Loop unrolling is a type of loop optimization where the loop body is replicated multiple times to reduce the loop iterations. Loop unrolling enables opportunities for instruction-level parallelism where the processor can perform simultaneous execution of independent instructions. Compilers often miss the opportunity for loop unrolling due to the inability to statically determine the number of loop iterations and the memory access patterns. We found 5 commits fix missed opportunities for loop unrolling. For instance, the ArrayFire-7f3fe1e commit identifies the missed opportunity of loop unrolling, resulting in suboptimal code execution. As shown in Listing 4, ArrayFire's transpose kernel uses `#pragma unroll` directive to guide the CUDA compiler to unroll the loop. However, the compiler fails to determine the memory accesses statically during compile time.

```
1 void transpose(Param<T> out, CParam<T> in,
2                  dim_type nonBatchBlkSize){
3 ...
4 #pragma unroll
5 -  for (dim_type repeat = 0; repeat < TILE_DIM;
6 -       repeat += blockDim.y) {
7 +  for (int repeat = 0; repeat < TILE_DIM;
8 +       repeat += THREADS_Y) {
```

Listing 4. Commit identifies missed loop unroll opportunity.

b) Function inlining: Function inlining is a compiler optimization technique where the body of the called function replaces a function call. On the one hand, function inlining can eliminate function call-related overhead, such as passing arguments and handling return value, and reduces register spilling. On the other hand, this optimization further opens many other intra-procedural optimization opportunities. However, the performance impact of function inlining is not always obvious and depends on the function, and it's invocation pattern. Previous literature has shown that function inlining optimization may or may not occur depending on the compiler version [9]. In our empirical study, we have found 3 performance commits manually direct the compiler to inline a function. For instance, the libMesh-e0374af commit identifies that libMesh's `BoundingBox::contains_point` method implemented small helper function `is_between` to make `contains_point()` more readable. However, the

compiler does not generate `is_between` as an inline and misses many optimization opportunities.

```
+#pragma unroll
for (int i=0; i<M; i++){
-    for (int j=0; j<N; j++){
-        ...
-        copy(v[(chirality*M+i)*N+j], clover[parity]
-        [(padIdx*stride + x)*N + intIdx%N]);
-        ...
-    }
+    Vector vecTmp = vector_load<Vector>(clover
+    + parity*offset, x +stride*(chirality*M+i));
+    for (int j=0; j<N; j++)
+        copy(v[(chirality*M+i)*N+j],
+        reinterpret_cast<Float*>(&vecTmp)[j]);
}
```

Listing 5. Vectorization of the loop achieves 1.5x speedup

C. Missing parallelism (MP)

We found that around 13 commits (7%) fix the missing parallelism in the original code. We observe a range of parallelism is introduced, including Vector parallelism/SIMD Parallelism (5), GPU parallelism (2), Instruction level parallelism (1), and Task parallelism (5).

1) SIMD parallelism:: The QUDA-5f028db commit identifies the missing SIMD parallelism. The listing 5 depicts the partial commit. As shown in the listing, QUDA's `FloatNOrder::load()` function has an inner loop performing consecutive memory access. However, the code fails to use SIMD intrinsic to get the benefits of parallel load/store operation.

2) GPU parallelism:: The commit HYPRE-a05d194 identifies missing GPU parallelism. The Listing 6 highlights the commit where it shows that HYPRE's `hypre_ParCSRRelax` function iterates a loop sequentially to read and write to an array and fails to use GPU parallelism.

```
+#if defined(HYPRE_USING_OPENMP_OFFLOAD)
+    int num_teams = (num_rows+num_rows%1024)/1024;
+    #pragma omp target teams  distribute
+    parallel for private(i) num_teams(num_teams)
+    thread_limit(1024) is_device_ptr(u_data,v_data,
+    l1_norms)
+#endif
          for (i = 0; i < num_rows; i++)
```

Listing 6. Missed GPU parallelism in HYPRE

3) Task parallelism: The commit GOMC-37a6bfd identifies that in GOMC's `CalculateEnergy::ParticleInter` function, the energy calculation of Monte Carlo simulation could not take full advantage of the parallelism provided by multi-core processors. As shown in Listing 7, the outer loop first iterates over all the trials, then performs the energy calculation, which could be divided into independent tasks for a partition range and executed in different threads to exploit task parallelism for efficiency.

```
- for (t = 0; t < trials; ++t){
+#pragma omp parallel sections private(start, end)
+ {
+    #pragma omp section
+    if(Schedule(start, end, p, 0, trials))
+    ...
```

Listing 7. Missing task parallelism in GOMC.

D. Inefficient parallelization (PO)

We found that 13 commits (7%) fix inefficient parallel code regions. Among these commits, 11 commits fix the inefficient work partitioning for the target accelerator.

Efficiently implementing a parallel algorithm on GPU architecture requires consideration of underlying GPU architecture. Modern GPU architectures are complex consisting of several streaming multiprocessors (SM) where each SM executes a set of threads known as warps. To fully utilize the GPU parallel capability, developers require to correctly split the parallel workload. An inefficient workload mapping results in load imbalance and high thread management overhead.

The commit QUDA-4ef8d8a identifies such inefficiency in QUDA's Multi-Reduce kernel. In the kernel, the block size, a CUDA programming abstraction, was incorrectly set that causing under-utilization of warps in SM.

E. Inefficient Concurrency control (ICS)

Our study found 7 commits (3.8%) fix the concurrency and synchronization-related performance bugs. These performance bugs include unnecessary locks (3), inappropriate lock granularity (1), use of inefficient locking mechanism (1), and unnecessary thread synchronization (2).

The commit OpenBLAS-3119b2a identifies unnecessary locks in OpenBLAS's `alloc_mmap` function. Listing 8 depicts the case, where the function holds the `alloc_lock` while accessing global data structure `release_info`. However, the code causes performance inefficiency when OpenMP thread parallelism is enabled. Since OpenMP already imposes a lock, this explicit locking is unnecessary.

```
static void *alloc_mmap(void *address){
+#if defined(SMP) &&  defined(USE_OPENMP)
+    LOCK_COMMAND(&alloc_lock);
+#endif
    release_info[release_pos].address = map_address;
    release_info[release_pos].func    =
        ↪ alloc_mmap_free;
    release_pos ++;
+#if defined(SMP) &&  defined(USE_OPENMP)
+    UNLOCK_COMMAND(&alloc_lock);
+#endif
```

Listing 8. Unnecessary locking for OpenMP based multi-threading.

F. Inefficient memory management (IMM)

We found 13 commits (7%) that fix inefficient memory management-related issues, including memory leak, redundant memory allocation, and repeated calls for allocation.

The OpenBlas-d744c95 commit identifies repeated allocation in OpenBLAS's `exec_threads` function. The function allocates buffer by calling `blas_memory_alloc` function. The `exec_threads` function is called by all the participating threads, which incurs a runtime overhead.

G. Other forms of inefficiencies

Our study finds a small set of other forms of inefficiencies including unintentional programming logic error (PE), IO inefficiency (IO), compiler regression (CR), and unnecessary process communication (UPC).

IV. RQ2: Fixing Performance Inefficiency

This section discusses the performance optimization techniques we frequently observed in our empirical study.

A. Micro-architecture specific optimization

We observed that 64 commits (34.4%) are related to micro-architecture-specific algorithm optimization. These optimizations include locality optimization for cache and memory (42), strength reduction (6), use of data types that reduces computation and memory overhead (4), using architecture-specific fast instruction (3), architecture-specific logic modification (3), choosing architecture specific fast kernel (3), and reduced precision arithmetic (3).

a) Operator strength reduction: Strength reduction is an optimization technique that replaces micro-architecturally expensive operations such as division, square-root, and exponential operation with a less expensive operation. For instance, the CasADi-29b6882 commit replaces the expensive integer division (20-26 clocks for 32-bit division) with a low overhead addition operation (1 clock cycle) in the `get_nz` function.

b) Use of data types that reduces computation and memory overhead: Data types with bigger sizes may incur significant latency compared to the smaller ones. When the algorithm permits, choosing a data type with lower bytes can improve performance. For instance, the GROMACS-85c36b9 commit reports that OpenCL's AMD version implements float3 as 16 bytes. By replacing the float3 with a standard 4-byte float, the kernel improves the register utilization and observes 1.25× and 1.4× for the Ewald and RF force-only kernels on AMD Vega GPU, respectively.

c) Architecture specific fast instruction: One example of such performance commit is OpenBLAS-2dfb804, where it reported OpenBLAS dgemm performance dropped on Ryzen 7 3700X CPU due to high latency `vpermpd` instruction used in the kernel. The commit replaces the instruction with low latency, high throughput `vpermilpd` instruction in the kernel.

1) Data locality optimization: The empirical study found that 42 commits (21%) are data locality optimization, including data structure optimization (7), tuning computational kernel size (6), reordering memory reference to improve temporal and spatial locality (5), reducing GPU-global memory access (5), thread-aware data access (3), improve register/cache utilization via blocking (2), avoid memory de-reference by storing data in register (2), memory pre-fetching and pinning (2).

a) Data structure optimization: Data structures that promote regular memory access improves spatial locality and thus lower the memory access latency. C++'s STL library provides `std::vector` and `std::array`. Both of the containers store objects in contiguous memory locations and thus cache efficient when accessed sequentially. On the other hand, STL's `std::list` is a doubly linked list and may incur a significant performance bottleneck in the cache. As a result, the CGAL-8855eb5 commit reported that it changed the lists to vectors in the code and observed performance improvement.

b) Reorder memory reference for data reuse: Optimization by reordering memory references improves temporal locality and thus improves cache utilization. One such example is the OpenBLAS-45fdf95 commit as shown in Listing 9, which reorders the memory references, `ptr_a0` and `ptr_b0`, to immediately reuse the pointers after first use.

```
for (k_count = k; k_count > 1; k_count -=2) {
    LOAD_A_PAIR(0);
+    ptr_a0 += 16 * 2;
    BROADCAST_B_PAIR(0, 0); MATMUL_4X(0, 0, 0);
    BROADCAST_B_PAIR(0, 1); MATMUL_4X(0, 0, 1);
    BROADCAST_B_PAIR(0, 2); MATMUL_4X(0, 0, 2);
+    ptr_b0 += 4 * 2;
    BROADCAST_B_PAIR(1, 0); MATMUL_4X(0, 1, 0);
    BROADCAST_B_PAIR(1, 1); MATMUL_4X(0, 1, 1);
    BROADCAST_B_PAIR(1, 2); MATMUL_4X(0, 1, 2);
-    ptr_b0 += 4 * 2;
    ptr_b1 += 4 * 2;
-    ptr_a0 += 16 * 2;
```

Listing 9. Reordering memory references.

c) Reducing GPU-global memory access: As we discussed in section III-B1, reducing GPU global memory reference can significantly improve application performance. For example, the QUDA-aa7049a commit eliminates the global GPU memory access by loading the data on the read-only data cache by explicitly calling `__ldg` (Read-Only Data Cache Load) intrinsic.

In addition to reducing global memory access, our study finds other GPU memory optimizations, including memory coalescing, avoiding GPU texture memory, asynchronous data transfer between host and device, and configuring shared cache sizes.

B. Domain specific optimization

We found that 27 performance commits (14.5%) are domain-specific optimization. These optimizations include eliminating unnecessary operation (8), reducing iteration or iterative data structure traversal (8), batching repeated function calls (3), domain-specific reduction of asymptotic complexity (3), avoiding unnecessary communication (2), code specialization (2) and lazy house-keeping to reduce resource management related overhead (1).

a) Eliminating unnecessary operation: This category of optimization was implemented to eliminate unnecessary function calls, traversal, memory allocation, or memory references.

For instance, HYPRE's rocSPARSE is a BLAS library for sparse computation implemented on AMD's Radeon Open Compute ROCm runtime. In the original implementation of `hypreDevice_CSRSpGemmRocsparse` function, it sorted the CSR matrix by calling the `hypre_SortCSRRocsparse` function. However, the commit HYPRE-827e799 suggests that the sort is unnecessary for this implementation and eliminating them provides "substantial performance savings".

b) Reduce loop iteration/traversal of data structure: Our study finds 8 commits eliminate unnecessary data-structure traversal by early bailing out the loop iteration. One example is OpenMM's `getByMass` function, where the code iterated the entire periodic table to search for an element that has an immediate higher mass than the target mass. However, since the periodic table is sorted by monotonically increasing element mass, iterating up to the first element that has a higher mass than the target is sufficient. The commit OpenMM-8bcff36 implements the optimization and reports that a test case enjoys a runtime speedup of 19×.

C. Guiding the compiler for missed optimization

We found 27 commits (14.5%) explicitly guiding the compiler for missed compiler optimization. These include function inlining (4), scalar replacement (1), enabling compile time evaluation (5), and various loop optimizations (17).

a) Function inlining: The libMesh-e0374af commit inlines `is_between()`, the helper function for further compiler optimization. As a result, the commit reports the single line enjoys 6.3× speedup.

b) Loop unrolling: The ArrayFire-928e77a explicitly unrolls the loop using OpenMP directive, `#pragma unroll`. Loop unrolling further enables several compiler optimizations, including vectorization and instruction-level parallelism. The commit reports an improvement of 1.2× in runtime.

c) Loop invariant code motion: The libMesh-1ad14f2 commit moves the `build_side_list` and `build_node_list` outside the loop to avoid redundant allocation and sort operations. The commit reports $O(Nsplits/Nprocs)$ times performance improvement.

D. Domain and architecture agnostic algorithm and data-structure optimization

This category of performance commits implements domain or architecture-agnostic optimizations for algorithms or data-structure traversal. We found that 17 commits (9.1%) adopt this category of optimization.

a) Reducing asymptotic complexity of search algorithm: One such example is the GROMACS-a711d41 commit that implements binary search for molecule lookup in `atoms_to_settles` function, reducing the time-complexity of lookup to $O(\log n)$. In another instance, the mlpack-198cec8 commit implements a priority queue for neighbor search algorithm where it performs the "peek" operation at $O(1)$ time.

b) Use of fast data structure interface: The data-structure libraries such as C++'s STL provides various data structure containers. However, the efficiency of the data structures depends on the use case. For instance, C++'s STL provides a contiguous data structure `std::array` and `std::vector`. However, `std::array` is a static array whose size is known at compile time and allocated inside stack memory. As a result, arrays are not re-sizable. On the other hand, std::vector is a dynamic array residing on a heap that can grow and shrink at runtime. Despite the dynamic allocation capability, allocating `std::vector` suffers from memory allocation overhead while `std::array` does not. Applications that know the size of the contiguous memory during compile-time can benefit from low overhead `std::array` over `std::vector`. We found that the ArrayFire-ee30e27 commit replaces the `std::vectors` with `std::array`.

c) Caching/Memoization/Lookup table: We found that caching and memoization eliminated redundant computation and redundant traversal (3). Additionally, static lookup tables eliminate expensive runtime evaluation (2).

For instance, CGAL's RemoveCurveFromStatusLine function searches the curve to be removed from the status line implemented using STL's multi-set container. However, the caller function, `HandleLeftCurves`, iteratively calls `RemoveCurveFromStatusLine` to remove the consecutive curves. Since multi-set stores the curves in sorted order, it is unnecessary to search for the next curve. In fact, the commit CGAL-351249b eliminates the unnecessary search by advancing the container iterator in the first call and caching it for the next call.

E. Introduce parallelism and Balancing parallel load

We found 17 commits (9.1%) introduce parallelism in code, and 11 commits (5.9%) fix the parallel load imbalance in the source code. The commits introduce parallelism by explicitly calling vector intrinsics (5), pthread based task parallelism (3), or by OpenMP-based parallel directives (2) and OneAPI (1). The commit that fixes the performance bugs related to parallel load imbalance uses techniques such as tuning task size (9), explicitly setting thread numbers (4), reducing parallelization (3), and sorting workload and schedule (1).

F. Memory management

Our empirical study finds 13 commits (7%) that fix the poor memory management-related performance bugs. These fixes include avoiding memory leaks by freeing allocated memory (5), pre-allocation to avoid repeated allocation (4), increasing memory size (1), using efficient memory management API (2), and using object reference rather than copy (1).

G. Eliminate unnecessary synchronization/barrier

Our study finds 7 commits (3.8%) fix synchronization/barrier-related performance bugs. These fixes include eliminating unnecessary lock (3), efficient lock implementation with atomic intrinsic (1), replacing strong consistency with a weaker one (1), eliminating unnecessary synchronization (1), and rewriting kernel to make memory access GPU warp-synchronous (1).

V. RQ3: Detection and Bug Fixing Effort

To understand the effort required to identify the performance bugs and the effort required to fix HPC performance bugs, we analyzed all the confirmed performance bugs. For measuring debugging and fixing effort, we adopted three metrics: (1) bug duration, (2) patch size (LOC) and (3) files changed (count). The bug duration metric is widely used in prior research works [43], [47] for measuring the debugging or bug identification complexity. This metric indicates buggy code is fixed starting from the day of its introduction to the bug fixing

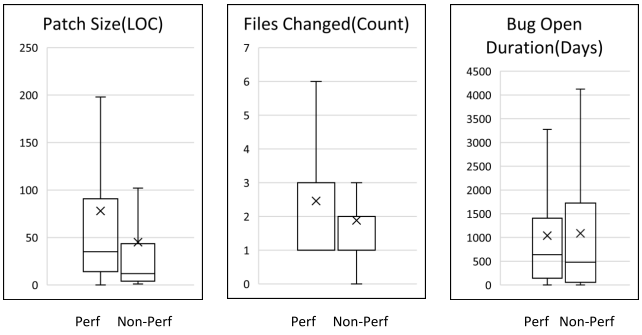

Fig. 3. Comparison of detection and bug-fixing effort ("Perf" = "performance bug", "Non-Perf" = "non-performance bug")

date. For our case, we calculated the day difference between the buggy file introduced and the bug fix commit applied. At the same time, bug-fixing complexity is measured by patch size (LOC) and files modified (count). Patch size (LOC) indicates how many lines are modified, and files modified (count) indicates how many files are modified to fix the bug. These metrics are also widely adopted by prior research [40], [43] to measure bug-fixing complexity. Intuitively, if a bug is complicated to fix, it would require applying fixes with multiple lines and multiple source file locations.

For our comparison, we have randomly selected 186 non-performance bugs from our existing 23 projects. We didn't select any non-performance bug from other repositories which would make the comparison unreliable. In our analysis presented in Figure 3, we can see on average a developer needs around 1100 days to debug and fix a performance bug as well as a non-performance bug requires around the same amount of days to resolve the issue. Besides, on average for performance issues, bug-fixing patches require around 80 lines whereas for non-performance it requires less than 50 lines. It shows that fixing performance bugs takes more effort than non-performance bugs. In addition to that, the number of files required to fix a performance bug is higher than non-performance bugs. To statistically validate the observations of performance and non-performance bugs, we conducted a Mann-Whitney U-test [48]. p-values for patch size and the number of files changed are 0.00001 and 0.0455, respectively, which is less than 0.05 (statistically significant). From this, we can conclude that debugging and fixing performance bugs require more effort than debugging and fixing non-performance bugs. On the other hand, p-value for bug open duration is 0.65272 which is much higher than 0.05. As a result, we can not statistically confirm that performance bug finding time is significantly longer than non-performance bugs.

VI. RQ4: Developer Expertise for Perf Bug Fix

Due to the complex underlying implementation of HPC applications, fixing performance bugs requires HPC domain-specific knowledge. To quantify the domain expertise of the developers in the HPC projects, similar to prior work [49] [50] [51], we define an HPC skill vector: $dev_s = (w_1, w_2, \ldots, w_n)$, where each dimension corresponds to one HPC skill category. We construct this skill vector for each developer who contributed to their respective HPC projects. For this purpose, first, we list the keywords of each skill category as shown in Table II. Then we search these keywords in all the submitted commits of each developer. If any keyword terms corresponding to an HPC skill category are present inside a developer's commit, we assume the committing developer has that skill; hence we increment the corresponding skill weight value w_i of that developer by one. We then calculate the HPC skill score for each developer by averaging their skill weights for each category as shown in Equation 1. We further performed log transformation on this score for convenient representation and statistical analysis.

TABLE II
KEYWORD TERMS USED FOR EACH HPC SKILL CATEGORY

Category	Keyword terms
gpu	gpu, __device__, cuda, nvidia, kepler, warp, opencl
vectorization	lxvp, vector, simd
memory/data locality	std::array, cache , prefetch, memory, locality
compiler optimization	unroll, const, inline, static
task optimization	openmp, omp, parallel
concurrency	lock, semaphore, mutex, barrier
data-structure library	std::array, std::vector, std::set, DynamicList, arma
memory management	leak, alloc

$$HPCSScore(dev_s) = \sum_{i=1}^{n} w_i/n \quad (1)$$

By comparing the skill scores among the developers, we aim to answer whether the fixer of performance bugs is relatively more experienced or not. Figure 4(a) shows the HPC expertise score of the performance committer vs. the overall expertise of the developers in all 23 projects we studied. The results show that the median HPC skill scores of the performance-fixing developers are higher (2.245) when compared to the median skill score of all developers (0.301). Mann-Whitney U-test on the result returned P-value as 0.001, which is less than 0.05, suggesting that this result is statistically significant. This analysis proves that performance-fixing developers have significantly higher domain expertise in HPC. Additionally, we plot the cumulative distribution function (CDF) on the HPC skill score in Figure 4(b). According to the figure, only 4.4% of the HPC developers have a higher skill score than 2.245, suggesting that the number of highly skilled developers to fix the performance bugs is very limited.

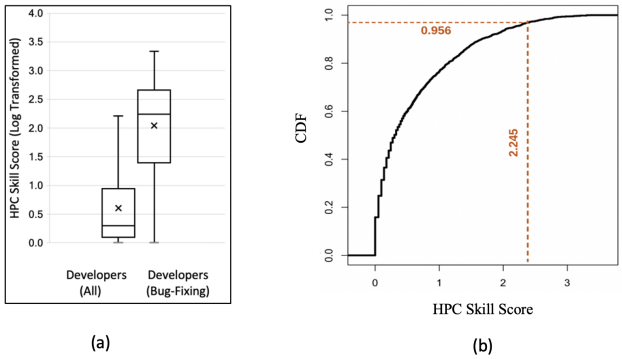

Fig. 4. (a) HPC skill score of developers - overall vs performance-fixing committer (outlier omitted), (b) CDF on HPC skill score of all developers.

VII. Threats to Validity

Threats to internal validity may be connected to how performance issues are categorized. Since there is no documentation to support the code's intention, this step may be biased. All the co-authors independently examined the performance bugs to minimize the problem and worked out disagreements until a consensus was reached. The taxonomy also adhered to meticulous manual analysis techniques, which are only confirmed once agreed upon by all authors, to determine whether the issue is performance-related or not. Nevertheless, we acknowledge that mistakes may inevitably occur during the manual procedure.

We considered the various HPC project types to ensure validity and conducted manual analysis project-wise. This allows us to compare the different kinds of bugs fairly and determine which types of bugs are most widespread in a particular project.

Other threats can be related to the external validity, which is how well our taxonomy generalizes outside of the dataset because our study evaluated 186 performance commits from 23 GitHub projects. These projects have been carefully selected, and we considered both small and large-scale projects. Additionally, it is challenging to increase the dataset because HPC is a vast field with various projects, not each of which was well-maintained at the time of our study. Apart from that, the manual analysis required much labor; each individual contributed roughly 480 hours to collecting and examining the issues. Nevertheless, we believe our findings will inform the HPC application developers and provide more insight into the future of HPC performance bug detection.

VIII. Discussion

The findings of our empirical study provide valuable insights into the HPC performance bugs and the required efforts toward mitigation.

First, our study finds a wide range of root causes of performance inefficiencies in HPC applications. To fix these performance bugs, the HPC application developers need in-depth knowledge about underlying hardware architecture, parallel programming models, data-structure libraries, compiler optimization techniques and their limitations, resource scheduling strategies of the runtime, and finally, the problem domain for domain-specific optimization opportunities. Due to cognitive overload, it becomes challenging for HPC application developers to write efficient code.

Second, our study finds that fixing performance bugs requires significantly more effort than non-performance bugs. Furthermore, fixing performance bugs require higher domain expertise. However, highly skilled developers are limited in number. As a result, maintaining highly performant scientific applications becomes challenging.

To overcome the challenges, the HPC developer community requires innovation in performance analysis tools, performance portable frameworks and runtimes, and recommender systems for writing high performance code. Our study makes two contributions towards the goal. First, it highlights the performance bugs frequently appearing in real-world HPC applications to guide the SE research community. Second, our study further curates a labeled dataset of performance bugs and fix patterns. On the one hand, the this dataset will help to develop bug detection tools; on the other hand, the fix pattern datasets will help to build recommender systems to write efficient code.

IX. Related Works

A. Studies on performance bugs

Performance bugs are a major contributor to performance degradation and resource waste in real-world software systems [41]. There has been a wide array of empirical studies on performance bugs that investigated performance bug characteristics. These studies investigate the mobile applications [43], [52], desktop or server applications [11], [13], [53], highly configurable software [42], and JavaScript systems [54]. More recently, several literatures study the domain-specific performance bugs, such as inefficient code on accelerators (i.e GPUs) [55], sub-optimal compiler code generation [9], autonomous vehicles [56], block-chain [57] and deep learning (DL) systems [58]–[61]. Unlike prior work, we perform an empirical study on the real-world HPC performance bugs.

B. Analyzing performance bugs in HPC applications

There is a substantial body of research that develops performance analysis tools for HPC applications [62]–[66]. The HPC community relies on these performance analysis tools to identify performance bugs. However, the capability of current performance tools is limited. First, performance measurement often causes significant runtime overhead for the HPC application [8]. Second, while performance profiling tools can identify the code regions that spend significant time, they often fail to guide the developer with a meaningful performance optimization strategy. It's the expert developers responsibility to fix the performance bugs. Our empirical study complements the performance analysis tools, guides novice developers toward possible optimizations, and provides insights for the tool community to develop sophisticated performance analysis tools.

X. Conclusion

In this study, we performed an empirical analysis on 1729 potential performance commits from 23 open-source HPC projects and identified 186 commits that were related to performance issues. Through manual analysis, we classified the performance bugs and fixes into ten and seven categories, respectively. Moreover, our analysis identified that fixing HPC performance bugs requires more effort and expertise than non-performance bugs. To the best of our knowledge, this is the first empirical study of the real-world HPC application performance bug. Our study provides a list of insights for HPC application developers. We hope our study will inspire software engineering researchers to innovate tools and techniques to reduce the developer burden.

REFERENCES

[1] C. Yang, W. Xue, H. Fu, H. You, X. Wang, Y. Ao, F. Liu, L. Gan, P. Xu, L. Wang *et al.*, "10m-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 57–68.

[2] H. Bhatia, F. Di Natale, J. Y. Moon, X. Zhang, J. R. Chavez, F. Aydin, C. Stanley, T. Oppelstrup, C. Neale, S. K. Schumacher *et al.*, "Generalizable coordination of large multiscale workflows: challenges and learnings at scale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–16.

[3] M. De Vivo, M. Masetti, G. Bottegoni, and A. Cavalli, "Role of molecular dynamics and related methods in drug discovery," *Journal of medicinal chemistry*, vol. 59, no. 9, pp. 4035–4061, 2016.

[4] A. Bhattacharjee and J. Wells, "Preface to special topic: Building the bridge to the exascale—applications and opportunities for plasma physics," p. 090401, 2021.

[5] P. Crozier, M. Howard, W. J. Rider, B. A. Freno, S. W. Bova, and B. Carnes, "Advanced technology and mitigation (atdm) sparc re-entry code fiscal year 2017 progress and accomplishments for ecp." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2017.

[6] A. Pasquali, M. Schönherr, M. Geier, and M. Krafczyk, "Simulation of external aerodynamics of the drivaer model with the lbm on gpgpus," *Parallel computing: On the road to exascale*, vol. 27, pp. 391–400, 2016.

[7] J. Ghorpade, J. Parande, M. Kulkarni, and A. Bawaskar, "Gpgpu processing in cuda architecture," *arXiv preprint arXiv:1202.4347*, 2012.

[8] K. Zhou, X. Meng, R. Sai, D. Grubisic, and J. M. Mellor-Crummey, "An automated tool for analysis and tuning of gpu-accelerated code in hpc applications," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[9] J. Tan, S. Jiao, M. Chabbi, and X. Liu, "What every scientific programmer should know about compiler optimizations?" in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–12.

[10] P. Su, S. Jiao, M. Chabbi, and X. Liu, "Pinpointing performance inefficiencies via lightweight variance profiling," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–19.

[11] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 77–88, 2012.

[12] M. Selakovic and M. Pradel, "Performance issues and optimizations in javascript: an empirical study," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 61–72.

[13] S. Zaman, B. Adams, and Hassan, "A qualitative study on performance bugs," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. Zurich: IEEE, Jun. 2012, pp. 199–208. [Online]. Available: http://ieeexplore.ieee.org/document/6224281/

[14] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: performance bug detection in the wild," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011, pp. 155–170.

[15] R. R. Curtin, M. Edel, M. Lozhnikov, Y. Mentekidis, S. Ghaisas, and S. Zhang, "mlpack 3: a fast, flexible machine learning library," *Journal of Open Source Software*, vol. 3, p. 726, 2018. [Online]. Available: https://doi.org/10.21105/joss.00726

[16] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Comp. Phys. Comm.*, vol. 271, p. 108171, 2022.

[17] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Denver Colorado: ACM, Nov. 2013, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/2503210.2503219

[18] The CGAL Project, *CGAL User and Reference Manual*, 5.5 ed. CGAL Editorial Board, 2022. [Online]. Available: https://doc.cgal.org/5.5/Manual/packages.html

[19] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[20] P. Yalamanchili, U. Arshad, Z. Mohammed, P. Garigipati, P. Entschev, B. Kloppenborg, J. Malcolm, and J. Melonakos, "ArrayFire - A high performance software library for parallel computing with an easy-to-use API," Atlanta, 2015. [Online]. Available: https://github.com/arrayfire/arrayfire

[21] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, "A tensorial approach to computational continuum mechanics using object-oriented techniques," *Computers in Physics*, vol. 12, no. 6, p. 620, 1998. [Online]. Available: http://scitation.aip.org/content/aip/journal/cip/12/6/10.1063/1.168744

[22] P. Eastman, M. S. Friedrichs, J. D. Chodera, R. J. Radmer, C. M. Bruns, J. P. Ku, K. A. Beauchamp, T. J. Lane, L.-P. Wang, D. Shukla, T. Tye, M. Houston, T. Stich, C. Klein, M. R. Shirts, and V. S. Pande, "OpenMM 4: A Reusable, Extensible, Hardware Independent Library for High Performance Molecular Simulation," *Journal of Chemical Theory and Computation*, vol. 9, no. 1, pp. 461–469, Jan. 2013. [Online]. Available: https://pubs.acs.org/doi/10.1021/ct300857j

[23] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. C. V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini, "MFEM: A modular finite element methods library," *Computers & Mathematics with Applications*, vol. 81, pp. 42–74, 2021.

[24] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[25] H. J. C. Berendsen, D. v. d. Spoel, and R. v. Drunen, "GROMACS: A message-passing parallel molecular dynamics implementation," *Computer Physics Communications*, vol. 91, no. 1, pp. 43–56, 1995. [Online]. Available: https://www.sciencedirect.com/science/article/pii/001046559500042E

[26] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi: a software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, Mar. 2019. [Online]. Available: http://link.springer.com/10.1007/s12532-018-0139-4

[27] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, "Libmesh: A c++ library for parallel adaptive mesh refinement/coarsening simulations," *Eng. with Comput.*, vol. 22, no. 3–4, p. 237–254, dec 2006.

[28] TileDB, Inc., *tiledb: Universal Storage Engine for Sparse and Dense Multidimensional Arrays*, 2022, r package version 0.15.0. [Online]. Available: https://github.com/TileDB-Inc/TileDB-R

[29] J. Kim, A. D. Baczewski, T. D. Beaudet, A. Benali, M. C. Bennett, M. A. Berrill, N. S. Blunt, E. J. L. Borda, M. Casula, D. M. Ceperley *et al.*, "Qmcpack: an open source ab initio quantum monte carlo package for the electronic structure of atoms, molecules and solids," *Journal of Physics: Condensed Matter*, vol. 30, no. 19, p. 195901, 2018.

[30] G. Chourdakis, K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H. Bungartz, L. Cheung Yau, I. Desai, K. Eder, R. Hertrich, F. Lindner, A. Rusch, D. Sashko, D. Schneider, A. Totounferoush, D. Volland, P. Vollmer, and O. Koseomur, "preCICE v2: A sustainable and user-friendly coupling library [version 1; peer review: 2 approved]," *Open Research Europe*, vol. 2, no. 51, 2022. [Online]. Available: https://doi.org/10.12688/openreseurope.14445.1

[31] R. D. Falgout and U. M. Yang, "hypre: A Library of High Performance Preconditioners," in *Computational Science — ICCS 2002*, G. Goos, J. Hartmanis, J. van Leeuwen, P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, vol. 2331, pp. 632–641, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/3-540-47789-6_66

[32] M. A. Clark, R. Babich, K. Barros, R. C. Brower, and C. Rebbi, "Solving Lattice QCD systems of equations using mixed precision solvers on GPUs," *Computer Physics Communications*, vol. 181, no. 9, pp. 1517–1528, Sep. 2010, arXiv:0911.3191 [hep-lat]. [Online]. Available: http://arxiv.org/abs/0911.3191

[33] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing,"

ACM Transactions on Mathematical Software, vol. 48, no. 1, pp. 2:1–2:33, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3480935

[34] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, May 2006. [Online]. Available: http://journals.sagepub.com/doi/10.1177/1094342006064503

[35] Y. Nejahi, M. S. Barhaghi, J. Mick, B. Jackman, K. Rushaidat, Y. Li, L. Schwiebert, and J. Potoff, "GOMC: GPU Optimized Monte Carlo for the simulation of phase equilibria and physical properties of complex fluids," *SoftwareX*, vol. 9, pp. 20–27, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711018301171

[36] F. Gygi, "Architecture of qbox: A scalable first-principles molecular dynamics code," *IBM J. Res. Dev.*, vol. 52, no. 1/2, p. 137–144, jan 2008.

[37] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A New Framework for Distributed Memory Dense Matrix Computations," *ACM Transactions on Mathematical Software*, vol. 39, no. 2, pp. 1–24, Feb. 2013. [Online]. Available: https://dl.acm.org/doi/10.1145/2427023.2427030

[38] "java implementation of the git version control system. "www.eclipse.org/jgit"," 2022, [Online; accessed 14-July-2022].

[39] Y. Chen, S. Winter, and N. Suri, "Inferring Performance Bug Patterns from Developer Commits," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2019, pp. 70–81, iSSN: 2332-6549.

[40] F. Nusrat, F. Hassan, H. Zhong, and X. Wang, "How Developers Optimize Virtual Reality Applications: A Study of Optimization Commits in Open Source Unity Projects," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 473–485, iSSN: 1558-1225.

[41] S. Garg, R. Z. Moghaddam, N. Sundaresan, and C. Wu, "PerfLens: a data-driven performance bug detection and fix platform," in *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. Virtual Canada: ACM, Jun. 2021, pp. 19–24. [Online]. Available: https://dl.acm.org/doi/10.1145/3460946.3464318

[42] X. Han and T. Yu, "An Empirical Study on Performance Bugs for Highly Configurable Software Systems," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Ciudad Real Spain: ACM, Sep. 2016, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/2961111.2962602

[43] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad India: ACM, May 2014, pp. 1013–1024. [Online]. Available: https://dl.acm.org/doi/10.1145/2568225.2568229

[44] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR '13. IEEE Press, 2013, p. 121–130.

[45] "Cuda toolkit," Aug 2022. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[46] "C++ Standard Template Library." [Online]. Available: https://www.cppreference.com/Cpp_STL_ReferenceManual.pdf

[47] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 428–439.

[48] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The annals of mathematical statistics*, pp. 50–60, 1947.

[49] T. Dey, A. Karnauch, and A. Mockus, "Representation of developer expertise in open source software," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 995–1007.

[50] J. E. Montandon, L. Lourdes Silva, and M. T. Valente, "Identifying Experts in Software Libraries and Frameworks Among GitHub Users," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. Montreal, QC, Canada: IEEE, May 2019, pp. 276–287. [Online]. Available: https://ieeexplore.ieee.org/document/8816776/

[51] C. Hauff and G. Gousios, "Matching GitHub Developer Profiles to Job Advertisements," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Florence, Italy: IEEE, May 2015, pp. 362–366. [Online]. Available: http://ieeexplore.ieee.org/document/7180095/

[52] M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in Android apps," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 352–361.

[53] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," p. 14.

[54] M. Selakovic and M. Pradel, "Performance Issues and Optimizations in JavaScript: An Empirical Study," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, 2015.

[55] Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "Fixing performance bugs: An empirical study of open-source gpgpu programs," in *2012 41st International Conference on Parallel Processing*. IEEE, 2012, pp. 329–339.

[56] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and a. Q. A. Chen, "A comprehensive study of autonomous vehicle bugs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 385–396. [Online]. Available: https://dl.acm.org/doi/10.1145/3377811.3380397

[57] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug Characteristics in Blockchain Systems: A Large-Scale Empirical Study," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. Buenos Aires, Argentina: IEEE, May 2017, pp. 413–424. [Online]. Available: http://ieeexplore.ieee.org/document/7962390/

[58] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A Comprehensive Study on Deep Learning Bug Characteristics," arXiv, Tech. Rep. arXiv:1906.01388, Jun. 2019, arXiv:1906.01388 [cs] type: article. [Online]. Available: http://arxiv.org/abs/1906.01388

[59] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 1110–1121. [Online]. Available: https://dl.acm.org/doi/10.1145/3377811.3380395

[60] J. Cao, B. Chen, C. Sun, L. Hu, and X. Peng, "Characterizing Performance Bugs in Deep Learning Systems," arXiv, Tech. Rep. arXiv:2112.01771, Dec. 2021, arXiv:2112.01771 [cs] type: article. [Online]. Available: http://arxiv.org/abs/2112.01771

[61] G. Long and T. Chen, "On Reporting Performance and Accuracy Bugs for Deep Learning Frameworks: An Exploratory Study from GitHub," *arXiv:2204.07893 [cs]*, Apr. 2022, arXiv: 2204.07893. [Online]. Available: http://arxiv.org/abs/2204.07893

[62] M. Chabbi, K. Murthy, M. Fagan, and J. Mellor-Crummey, "Effective sampling-driven performance tools for gpu-accelerated supercomputers," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[63] K. Zhou, M. W. Krentel, and J. Mellor-Crummey, "Tools for top-down performance analysis of gpu-accelerated applications," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3392717.3392752

[64] B. Welton and B. Miller, "Exposing hidden performance opportunities in high performance gpu applications," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 301–310.

[65] F. Schmitt, R. Dietrich, and G. Juckeland, "Scalable critical-path analysis and optimization guidance for hybrid mpi-cuda applications," *The International Journal of High Performance Computing Applications*, vol. 31, no. 6, pp. 485–498, 2017.

[66] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb, "Parallel performance measurement of heterogeneous parallel systems with gpus," in *2011 international conference on parallel processing*. IEEE, 2011, pp. 176–185.