# Combining Phase Identification and Statistic Modeling for Automated Parallel Benchmark Generation


## ABSTRACT

Parallel application benchmarks are indispensable for evaluating/optimizing HPC software and hardware. However, it is very challenging and costly to obtain high-fidelity benchmarks reflecting the scale and complexity of state-of-the-art parallel applications. Hand-extracted synthetic benchmarks are time- and labor-intensive to create. Real applications themselves, while offering most accurate performance evaluation, are expensive to compile, port, reconfigure, and often plainly inaccessible due to security or ownership concerns.

This work contributes APPRIME, a novel tool for trace-based automatic parallel benchmark generation. Taking as input standard communication-I/O traces of an application's execution, it couples accurate automatic phase identification with statistical regeneration of event parameters to create compact, portable, and to some degree reconfigurable parallel application benchmarks. Experiments with four NAS Parallel Benchmarks (NPB) and three real scientific simulation codes confirm the fidelity of APPRIME benchmarks. They retain the original applications' performance characteristics, in particular the relative performance across platforms. Also, the result benchmarks, already released online, are much more compact and easy-to-port compared to the original applications.


## 1. INTRODUCTION

Benchmarks play a critical role in evaluating hardware and software systems. Compared to CPU, database, and mobile test workloads, supercomputing benchmarks are especially challenging and costly to construct or acquire. With both the scale (in terms of problem size and parallelism) and the complexity of applications growing alongside machine sizes, kernel-based benchmarks such as the NPB suite [20] fail to portrait state-of-the-art applications (e.g., multi-physics codes). Meanwhile, hand extracted benchmarks based on real-world, large-scale applications (such as FLASHIO [18] and GTCBench [13]) are highly labor-intensive to create and cannot easily keep up with the evolution of their long-lived base applications.

To this end, recent research developed tools for automatic generation of communication benchmarks based on trace compression and replay [36, 40]. The automatically generated codes can keep up with the original application's evolution rather easily. However, replay-based benchmarks have several intrinsic drawbacks. They require the use of a specialized, compression-enabled tracing library and cannot leverage other formated existing traces or standard tracing libraries. Also, they are driven by timestamp information collected in the original traces, making them more suitable for reproducing hard coded communication patterns rather than recreating comprehensive, platform-dependent parallel workloads encompassing the interplay among computation, communication, and I/O.

Meanwhile, there are also recent projects investigating the creation of reconfigurable benchmarks, to be discussed in more details in Section 5. However, such tools possess significant restrictions. The more general-purpose benchmark tools [11, 15] require users to "assemble" a synthetic benchmark from a limited number of key workload characteristics such as instruction mix and instruction-level parallelism. Skel [19], a parallel I/O benchmark creation tool, also requires users to clearly identify the begin and end points of periodic I/O phases. In addition, they are not designed to include computation or communication activities.

In this paper, we propose APPRIME, an automatic benchmark generation tool based on off-line statistical trace profile extraction. Given an iterative parallel simulation (the most common type of large-scale HPC applications), APPRIME takes as input the set of traces generated by parallel processes in one execution of the original application. and automatically generates as output *benchmark source code* with similar computation, communication, and I/O behavior. This result benchmark comes with a concise configuration file for users to set execution parameters, such as the number of timesteps and checkpoint frequency. Rather than only aiming at future parallel trace replay, APPRIME strives to "understand" (to some extent) an application and create a stand-alone benchmark that imitates its behavior.

Unlike trace replay-based benchmark creation, APPRIME obtains information from traces but takes a "statistical view" of applications, where a benchmark should reproduce the distributions (in many aspects of parallel program behavior, from instruction composition, to relative ranks of communication partners, to file read/write sizes), rather than line-by-line repetition of traced events. On the other hand, it recognizes that parallel programs are usually tightly coupled codes, whose executions do not build on random events: activities across processes and across timestep iterations are highly correlated (if not identical). Therefore APPRIME takes a hybrid approach, with (1) automatic trace-based phases (timesteps) identification through string analysis, (2) Markov-Chain-based timestep behavior model to enable reconfigurable execution length (number of timesteps), and (3) statistical regeneration of event parameters. This way, APPRIME retains the accurate event ordering for communication and I/O calls within each timestep, as well as the transitions between different types of timesteps (a phenomenon observed in our real application study). Meanwhile, it regenerates event parameters, plus the time gaps as *computation intervals* between each adjacent events, according to their value *distributions* observed from the traces. By distilling

patterns (for both loop structure and communication operations), the end product APPRIME benchmarks are more flexible, portable, and human-comprehensible.

| Name | Project domain | Typical prod. run scale (# of cores) | Open source | Current status |
|---|---|---|---|---|
| XGC* | Gyrokinetic | 225,280 | No | Done |
| GTS* | Gyrokinetic | 262,144 | No | Done |
| BEC2 | Unitary qubit | 110,592 | No | Done |
| QMC-Pack* | Electronic molecular | 256 - 16,000 | No | Ongoing confirmed applicable |
| S3D* | Molecular physics | 96,000 - 180,000 | No | Confirmed applicable |
| AWP-ODC* | Wave propagation | 223,074 | No | Confirmed applicable |
| NAMD* | Molecular dynamics | 1,000 - 20,000 | No | Confirmed applicable |
| HFODD* | Nuclear | 299,008 | Yes | Confirmed applicable |
| Lammps* | Molecular dynamics | 12,500 - 130,000 | Yes | Confirmed applicable |
| SPEC-FEM3D* | Wave propagation | 150 - 600 | Yes | Confirmed applicable |
| NAS-BT | Tri-diagonal solver | N/A | Yes | Done |
| NAS-LU | Gauss-seidel solver | N/A | Yes | Done |
| NAS-CG | Conjugate gradient | N/A | Yes | Done |
| NAS-SP | SP-diagonal solver | N/A | Yes | Done |

*: Applications with * are the ones awarded with large allocations[1] through DOE INCITE [2]. All APPRIME generated benchmarks are released online at [1].

**Table 1: Applications evaluated, or examined as candidates, for APPrime benchmark generation**

Note that APPRIME is intended to emulate the base application's behavior in *all three* major dimensions: computation, communication, and I/O. This paper presents our first step, a proof-of-concept prototype focusing on *re-producing communication and I/O activities*, with computation emulated with a rather simplistic manner (by sleep intervals whose durations are generated statistically). The emulation of computation activities, planned as future work, is to be plugged in as a building block.

We evaluated our APPRIME prototype with iterative parallel simulations, including three large-scale, closed-source applications and four communication-heavy NAS Parallel Benchmarks. All three real applications are top tier resource consumers, like GTS and XGC running on Titan [3], the world's No.2 supercomputer. In addition, we verified with the owners/users of all other DOE allocation awarded, resource-intensive applications (marked with * in Table 1) that APPRIME can adequately model their computation, communication and I/O behavior. Table 1 summarizes the diverse application domain, execution scale, source availability, and APPRIME benchmark generation status of all applications we examined.

Our evaluation results verify that automatically generated APPRIME benchmarks are compact, portable, and able to accurately represent applications' performance characteristics. All of our generated benchmarks have been released online [1], while we plan to extend the effort to other closed-source applications in Table 1. The initial success also motivates our ongoing work on refining APPRIME with statistical computation workload regeneration, plus other extensions (see future work discussion in Section 6).

## 2. SAMPLE APPRIME USE CASES

For further motivation, we give two sample use cases that highlight the need for realistic, configurable, and compact parallel benchmark codes. While our approach is not library-specific, for convenience we limit our discussion to MPI for the rest of the paper, due to its dominance in parallel scientific applications and its capability of performing both inter-node communication and parallel file I/O. Note that though we gave two specific use cases, a single or a set of APPRIME benchmark(s) can be published and used as general HPC benchmarks.

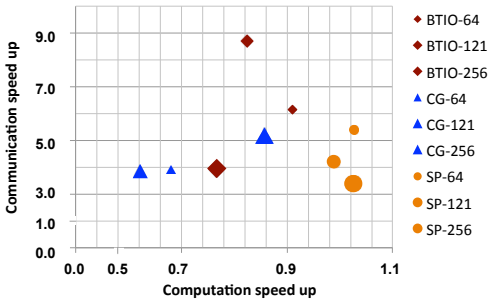



**Figure 1: Sample cross-platform performance**

**Use Case 1: Cross-Platform Performance Estimation** Application users constantly need to adopt new platforms (supercomputers, clusters, and clouds), as their applications outlive machines. Estimating cross-platform performance for real parallel application, however, is highly challenging. Figure 1 illustrates this by plotting the relative performance of three widely used parallel benchmarks across our two tested platforms (to be described in detail later). Not only is the relative performance highly application-dependent, for the same application, the computation and communication dimensions (axes $x$ and $y$) show very different time ratios, each of which further depends on the execution scale. Therefore, it is very difficult to "guess" a given application's performance on a new candidate platform, from either hardware/software parameters or published results from other programs, without actual porting and testing.

Unfortunately, for a scientist assessing candidate platforms for simulation $A$, it costs a significant amount of time *and* labor just to port the code to each candidate machine. For example, scientific applications typically depend on quite a few third-party libraries (such as Petsc and netCDF), which require non-trivial effort to acquire/install on each platform, not to mention adjusting codes or makefiles to compile the entire application. With APPRIME, the scientist can collect traces of one or more typical executions of $A$ on its current platform. APPRIME takes such traces as input, and generate "fake" codes $A'$ as benchmark, with complete MPI source code. $A'$ performs fake computation, communicates and reads/writes junk data, without relying on libraries beyond MPI. However, the computation, communication, and I/O patterns are all generated based on the original appli-

cations' traced behavior. Better, $A'$ can be configured in a similar way as $A$, with users configurable parameters, such as the number of iterations, frequency and API of periodic I/O operations, in the input file or job submission script. $A'$ therefore supports almost effortless porting while maintaining $A$'s essential behavior, allowing cost-effective candidate machine evaluation.

**Use Case 2: I/O Method and Sensitivity Assessment** APPRIME can also help library and middle-ware developers to assess their design or optimizations without involving real applications and their developers. For example, I/O library designers prefer to evaluate their product using real application codes, with given real I/O behaviors (I/O frequency, burstness, size, and access patterns), as well as real interplay between I/O and computation/communication. The latter is particularly important when asynchronous I/O operations are used. However, to obtain and successfully build several state-of-the-art parallel simulations itself is a daunting task, let alone modification of unfamiliar, long source code to enable the use of new I/O libraries or interfaces. APPRIME allows I/O library designers to simply request execution traces from such applications, say $B$, $C$, and $D$, and again generates their fake counterparts $B'$, $C'$, and $D'$, with highlighted (parallel) I/O calls. The designers can then apply their libraries or library updates to these benchmarks, validating their approaches using experiments and checking the impact of user-set configurations or internal parameters.

# 3. APPRIME DESIGN

We design our APPRIME prototype to validate the idea of generating accurate yet reconfigurable benchmark based on statistical summarized *profile* of traces, without retaining or replaying the original/decompressed trace. Figure 2 illustrates the software architecture of APPRIME and its two-phase workflow: *trace profile extraction* (*extractor* for short in the rest of the paper) and *automatic configurable benchmark generation* (*generator*).

To construct a parametric benchmark with APPRIME, users need to provide the following input: (1) a set of traces from one or more prior executions of the target application, (2) the number of timesteps executed in the traced run, and (3) the frequency of each type of periodic I/Os (such as checkpoint and result-snapshot output). Below we give more details on the design of the extractor and generator components, respectively.

## 3.1 Trace Profile Extraction

The APPRIME extractor automatically parses input per-process traces into multiple phases and identifies the main loop as well as other periodic I/O patterns. Its design is based on the observation that most iterative parallel applications have similar execution patterns in the form of $I(C^xW)^*F$ [41]. Here $I$ and $F$ are the one-time initialization and finalization phases, respectively. The iterative computation component contains $C$, the *timestep computation phase* (including communication activities involved in computation), and $W$, the *periodic I/O phase*.

While most parallel iterative simulations share the aforementioned common pattern, there are several issues complicating automatic phase recognition. First, there are often some degree of deviation from this pattern in recorded traces across different timesteps. For example, two of the seven applications we experimented with contain around 0.3% of communication events not fitting into the repetitive pattern. Such variance exists not only with operation event parameters, but also the count and ordering of events themselves. Second, there are often more than one periodic I/O phases, with each occurring in a different frequency: an application may capture one snapshot of intermediate results every 200 timesteps, plus one checkpoint every 1000 timesteps. In addition, real application traces may contain the execution records from a large number of processes (e.g., 100,000$x$ records from Titan at ORNL), with each further containing many timesteps. Identifying the repetitive pattern both within and across processes at such scale requires efficient and scalable trace processing.

As to be discussed in the rest of this section, the first two challenges are addressed by further extending the iterative computation template. Here $C$ is replaced by $C_{[0,\ a]}D^{0|1}C_{[b,\ |C|]}$, allowing rare but possible irregular $D$ phases. Also, a user-supplied number of different I/O phases, $W_i$, are allowed, each occurring at its own *regular* interval. Given the total number of timesteps (number of times $C$ was executed in the trace) and the I/O frequence for each $W_i$, APPRIME will generate benchmark that reproduces the interleaving pattern of $C$ and all $W_i$ phases, while discarding occurrences of the minor $D$ detected. We address the third challenge with a new, fast string-based phase recognition algorithm, leveraging the extra knowledge of user-specified timestep counts and I/O frequencies, given as input to APPRIME.

### 3.1.1 Trace Parsing

APPRIME accepts traces produced by popular tracing libraries such as DUMPI [5] and ScalaTrace [22]. It is also fairly straightforward to extend our support to more tracing libraries. These traces typically contain sequences of library (MPI) calls, each with a list of parameter values, plus begin/end timestamps of the invocation.

APPRIME pre-processes such traces by parsing each record and then building per-process *event tables*. Figure 3 shows a sample event table segment along with the corresponding DUMPI trace. Each table row describes a traced MPI event, with attributes such as start/end time stamps, data type, etc. Note that with event timestamps, the untraced *computation intervals* are implicitly stored for each event. The *Phase ID* and *Phase type* fields, currently marked as N/A, are to be filled by the Phase Identifier (Section 3.1.2).

There are a number of design considerations key to the efficiency of trace parsing: (1) tasks of parsing traces from different processes are mutually independent and therefore can be easily parallelized; (2) trace parsing is done via linear scan, accommodating APPRIME traces or event tables that are larger than memory size, by staging data to/from secondary storage; (3) an event table may appear sparse, with over 20 columns covering all MPI call parameter names, but is implemented using a compact delimited text format. Our experiments with seven workloads generate per-process event tables with sizes always smaller than the original decompressed DUMPI trace, as summarized in Table 4.

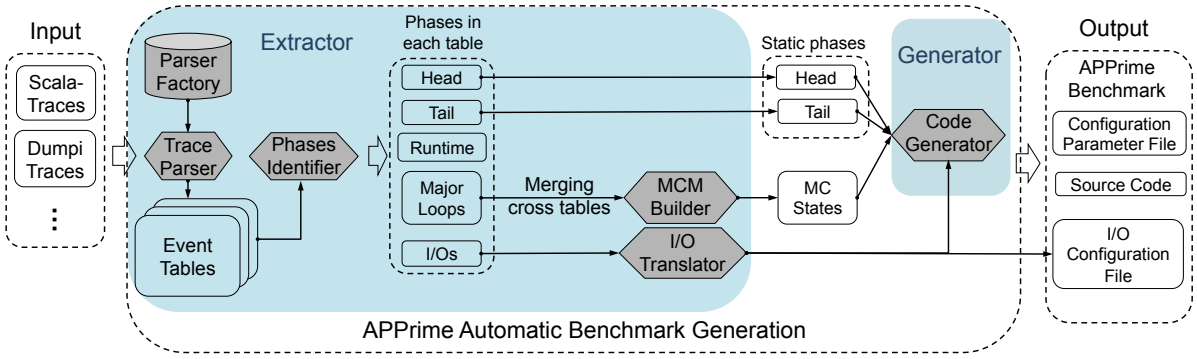



**Figure 2: APPrime overall workflow. The left part is input, right part output. It has two phases, which are trace profile extractor and benchmark generator respectively.**

***Original ASCII DUMPI Trace***

...
MPI_Bcast entering at walltime 102625.244058046, int count=1, MPI_Datatype datatype=4 (MPI_INT), int root=0, MPI_Comm comm=4 (user-defined-comm), MPI_Bcast returning at walltime 102625.2449640.

MPI_Barrier entering at walltime 102625.245683046, MPI_Comm comm=5 (user-defined-comm), MPI_Barrier returning at walltime 102625.2534360.

MPI_File_open entering at walltime 102627.269166046, MPI_Comm comm=5 (user-defined-comm), int amode=0 (CREATE), filename="simple.out", MPI_Info info=0 (MPI_INFO_NULL), MPI_File file=1 (user-file), MPI_File_open returning at walltime 102627.4391070.
...

***Sample Joint Per-process Event Table of Communication and I/O Events***

*Computational intervals*

| MPI function name | Start | End | Data count | Root | Comm. rank | File access mode | Phase ID | Phase type | ... |
|---|---|---|---|---|---|---|---|---|---|
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| MPI_Bcast | ...5.244 | ...5.245 | 1 | 0 | 4 | N/A | N/A | N/A | ... |
| MPI_Barrier | ...5.246 | ...7.253 | N/A | N/A | 5 | N/A | N/A | N/A | ... |
| MPI_File_open | ...7.269 | ...7.439 | N/A | N/A | 5 | CREATE | N/A | N/A | ... |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |

**Figure 3: Example of trace and event table segment**

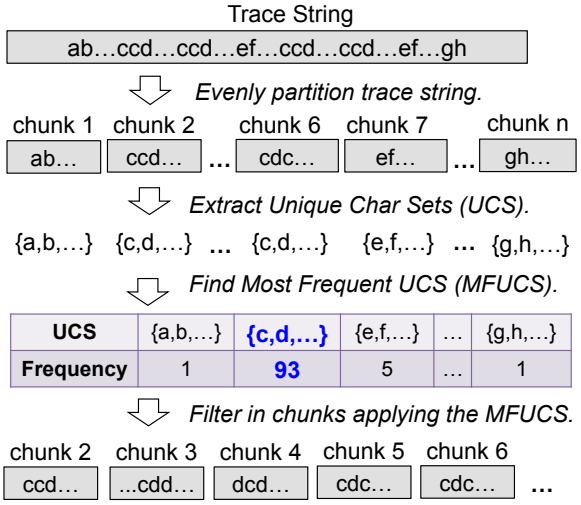



**Figure 4: Using MFUCS to locate desired chunks**

### 3.1.2 *Phase Identification*

The extractor aims at identifying the aforementioned common structure from the input traces. It takes the per-process event tables as input, and fills the missing *Phase ID* and *Phase type'* fields in these tables. At the end of this procedure, each event will be assigned a pair of phase ID and phase type that it is considered belonging to (e.g., "Phase ID of 3 and type of $C$" indicates "the 3rd iteration $C$ phase"). Note that we do not further identify nested loops within the main computation loop, as APPRIME performs summary of communication and I/O operation distributions and regenerates benchmark with a statistical approach. Therefore the trace length, in terms of the number of events within a main iteration, is not a concern, as to be shown in Table 4.

The premise of APPRIME's automatic trace phase identification is that the application behaviors are deemed to be identical or highly similar across iterations. Admittedly, as mentioned earlier, we did observe *data-dependent* activities triggered by dynamic conditions (e.g. error checking/handling). But such deviation from the "regular" iteration behavior is quite small. We observe that the majority of large-scale applications (such as *all* of the top 15 resource consumers in Table 1) satisfy the following assumptions, making them eligible for APPRIME's fast, specialized phase identification algorithm:

1. The iterative application's trace event sequence follows the extended two-level template proposed above.
2. The number of timesteps (i.e., the total count of $C$ phase iterations) and frequencies of all periodic I/O ($W_i$) phases are known in advance.
3. The repeating $C$ phases are identical, as well as all $W_i$ of same frequency, in terms of communication and I/O event sequences (not necessarily parameter values), among different timesteps within each process.
4. $D$ phases, if any, have the lowest occurrences compared to $C$ or $W_i$ phases, and each of them should be shorter than a single $C$ phase and, in its entirety, not a subsequence of the $C$ phase.

These assumptions hold for all real-world applications or benchmarks tested in this work. In particular, later in the paper Table 4 shows that the total length of $D$ phases (in

terms of event counts) is about 0.3% of the total execution. In the case that the relatively strong 4th assumption is indeed found to be invalid for a certain application, APPRIME employs a backup algorithm. It performs more general, yet much slower string pattern recognition, with a time complexity of $O(m^3)$, where $m$ is the total trace length in event count. With either algorithm, there is no constraint on input trace length, allowing APPRIME to handle traces from production runs, whose typical iteration number and I/O frequency settings are given in Table 2 (application descriptions are given in Section 4.1).

| App. | # Timesteps | Snapshot freq. | Checkpoint freq. |
|---|---|---|---|
| BEC2 | 100,000 | 1/500TS | 1/20,000TS |
| XGC | 100,000 | 1/500TS | 1/10,000TS |
| GTS | 30,000 | 1/200TS | 1/4,000TS |

**Table 2: Sample production run settings of real applications**

APPRIME first converts records stored in each *event table* into a compact trace string, retaining only the event names (communication and I/O function names). Each unique event name is mapped to a single character, based on the observation that typical applications use a small subset of MPI library routines [33]. This has been confirmed by our own observation: Table 4 shows that the seven applications tested use from 11 to 38 unique MPI functions. This step transforms the original phase detection to a string pattern matching problem, where each character in the trace string represents a traced event. Our goal is to find the substrings composing $C$ and other recurring phases.

Our solution is inspired by a Coarse-to-Fine approach [24] proposed for efficient object detection in image processing. The main intuition is that (1) repetitions of $C$ compose a major part of the string and (2) we know $n$, the exact number of $C$'s repetition. Consequently, if we simply partition the trace string uniformly into $n$ *chunks*, considering the non-$C$ characters included, each chunk would be (slightly) longer than the actual $C$ phase string. Further, we know that $I$ and $F$ occur only once, while $W$ and $D$ occur at a (much) lower frequency. By converting each chunk into a directed graph and searching for the most frequent one, we can identify *clean chunks* that contain only segments from the $C$ phase.

Candidates for such clean chunks are quickly selected by computing the *Unique Character Set (UCS)* of each chunk, which is the vertex set of the graph, as illustrated in Figure 4. Here the UCS of $\{c, d, ...\}$ is found to be the most frequent (93 times), therefore all the chunks with this UCS are selected to perform more fine-grained analysis, such as direct edge set comparison, to identify the $C$ substring that reappears (with "rotation" effect considered) most frequently.

Once the $C$ phase string is found, APPRIME re-scans the entire trace string, identifying the remaining phases. Intuitively, the "head" preceding the first occurrence of $C$ (first timestep) is marked as $I$, while the "tail" following the last timestep as $F$. Based on the given I/O frequency for each $W_i$, APPRIME examines the "gap strings" between the appropriate pair of adjacent $C$ phase strings (e.g., if the frequency of a $W_i$ is once per 50 timesteps, we will check between the 50th and the 51st timesteps). We apply a simple string matching algorithm here to identify the maximum common substring for each $W_i$. This is an iterative process starting with the most frequent I/O phase, as often such a phase is contained in the same gap string as a less frequent one, and needs to be removed for identifying the latter. All the characters not belonging to any $C$, $I$, $F$, or $W$ phases are considered $D$ phase events and ignored in the rest of APPRIME processing.

Considering the "rotation" effect of $C$ substrings in "clean chunks", the worst-case time complexity of the phase identification step is $O(m^2/n)$, where the trace contains $m$ events in $n$ timesteps. In practice, phase identification makes a very small part of APPRIME's processing overhead, which is dominated by the I/O-heavy trace processing. Table 4 gives the total sequential processing time on a Core-2 Duo with 4GB memory laptop for generating each APPRIME benchmark in column "Time cost".

### 3.1.3 Event and Timestep State Summarization

By now, APPRIME has partitioned the trace into timesteps, with events assigned the appropriate phase type and phase ID information. The next step is to transform the linear sequence of events, with recurrent phases identified, into a compact structure that easily maps to loops, for subsequent benchmark code construction.

Recall that in our phase detection, we only looked at the event names, ignoring the event parameters. Since we made a strong assumption that each iteration of the $C$ phase forms the *identical* series of events, we wondered whether these events also used identical parameter values (such as message sizes, peer ranks in point-to-point communication, and communicators). The answer is negative: from timestep to timestep, the parameter values do vary. This is logical considering that the sequence of events conform to the compile-time code sequence, while the parameter values are mostly assigned dynamically at runtime. An interesting finding is that these values form *clusters*, indicating the existence of a relatively small set of internal "timestep states", which are groups of timesteps having same or highly similar behaviors.

Such program behavior, confirmed across our tested set of applications and benchmarks, lends us more opportunities to summarize and reproduce variances in computation/communication patterns statistically. To create a concise, configurable benchmark that *approximates* the target application's behavior instead of replaying the specific execution traced, APPRIME performs such summarization in three dimensions: (1) detecting distinct timestep states, plus the transition probability between a pair of states, (2) replacing the actual event sequence with the *probability distribution* of events and event parameters within a timestep state, and (3) comparing and grouping the behavior of all processes.

Throughout the three dimensions, APPRIME' design needs to balance between accuracy and code's compactness/reconfigurability. A large number of program states or process groups, while more faithful to the original application, lead to verbose, "replay-style" codes. Below we

present APPRIME' design choices in its multi-dimensional summarization.

**Event Parameter Histogram Construction** For event parameters, including the computational intervals, we take the standard solution of building histograms [38], one histogram per parameter per event, across all phase iterations. We found the majority of parameter fields in our test applications to be either constant or with normal distribution in value, so we apply Sturges's formula [30] to decide the bin number and bin width. With a few exceptions, such as GTS' point-to-point communication buffer sizes with bimodal distribution, we found Doane's formula [10] working well. APPRIME can be easily extended with more sophisticated histogram building techniques if necessary. After constructing such global histograms, we now replace the original values in the table with the associated bin's mean value.

**Cross-process Event Table Merging** Next, APPRIME merges similar event segments with the same phase rank across all per-process event tables. This procedure goes through all event tables, starting from that of process 1, merging them into the process 0 event table ("root table"), one after another. Recall that applications going through such summarization already passed the phase identification step, meeting expectations on matching $C$ phase (as well as the number of $C$ iterations) and matching $W_i$ frequencies.

When merging the corresponding phases (segments in a pair of event tables with matching phase type and phase ID) from different processes, both segments' events may or may not match. In the former case, APPRIME merges corresponding events, for each of which creating a list of all distinct parameter values. In the latter case (e.g., when there are "aggregator" processes which act differently than its "ordinary" peers), APPRIME first merges the "matching events", starting from collective event pairs with matching parameters. Then APPRIME inserts remaining events into the "root table" between each two "matching events" following their original sequence. During this procedure, APPRIME stores each event's preceding *computational intervals* from all applicable processes in a 1D array.

**Markov Chain Model Construction:** Our final dimension in program behavior summarization is the timestep-to-timestep $C$ phase event parameter variance, which we adopt Markov Chain (MC) [23] to model statistically. MC is a technique to model state transitions, where the next state depends only on the current state but not on the sequence of events preceding it. Most parallel simulations are iterative scientific simulations whose $C$ phases possess this property. After further inspection of the attribute values in event tables among different $C$ phases, we choose the following three key parameters as the metrics to group $C$ phases into one *MC state*: (1) message (buffer) size, (2) rank of target communicating process, and (3) the communicator ID. The rest of most events' parameters values are found to be static across timesteps. Then MC Builder assigns the result MC state with an unique state rank, as depicted in Figure 6. Finally, MC Builder statistically computes the transition probability between each pair of states and stores the result in a probabilistic transition matrix.

With timestep states identified, the APPRIME MC Builder statistically calculates a per-process, per-timestep-state 1D histogram for each *computational intervals* ("bubbles" between adjacent traced events), to retain the state-specific computation duration distribution, see Figure 5. As to I/O, since periodic I/O calls are found to have static parameter values, APPRIME is able to summarize each type of $W_i$ phases with only one state, which always transits to itself.

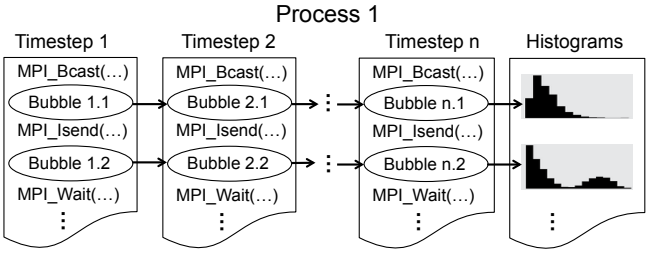



**Figure 5: Profiling computation intervals (bubbles) across timesteps**

Properly concatenated together, these models will guide the code generator to emit output benchmark codes. Note that while alternative tools, such as Hidden Markov Model (HMM) [25], may offer more powerful modeling, we consider MC sufficient based on scientific parallel applications' rather static behavior. In fact, the result MC models we obtained from applications are close to finite state machines.

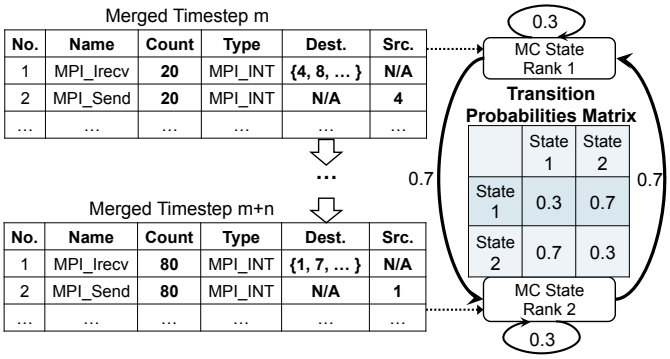



**Figure 6: Converting timesteps ($C$ phases) to MC states**

## 3.2 Configurable Benchmark Generation

Given the event tables and transition matrices produced by the APPRIME extractor, finally the APPRIME generator automatically constructs a source program that resembles the original application. Following the aforementioned template ($I(C^xW)^*F$), the generator creates the one-time static phases $I$ and $F$, and most importantly, the iterative phase between the two, with reconfigurable parameters such as the number of timesteps and periodic I/O frequencies. Within this major loop, each iteration has one $C$ phase recreating the behavior summarized by its corresponding MC state. Periodic I/O phases are inserted at appropriate frequencies.

Rather than replaying the exact sequence of timestep states recorded, the MC model constructed by the extractor allows

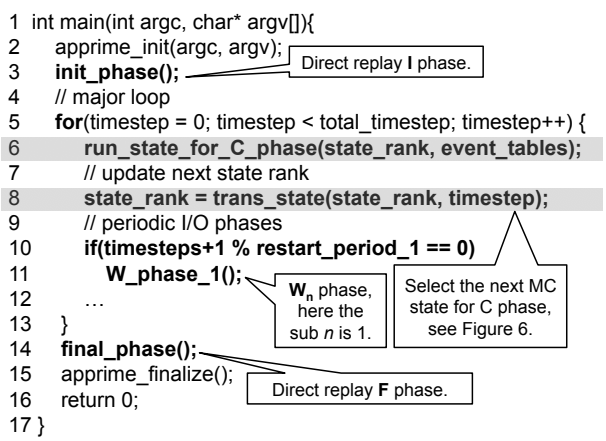



**Figure 7: Code structure in main()**

the generator to dynamically select the next MC state to transit to at the end of each timestep, as shown in Figure 7.

Within each MC state, though, the generator deploys the more faithful direct replay strategy, transforming event table entries to respective communication or I/O calls, with corresponding recorded parameter values for each process. Note that identical values across processes have already been compressed in the merged event tables. Usually, values are represented as one dimensional array and each of its elements is for one process.

Between each pair of adjacent communication or I/O calls, APPRIME inserts a *computation interval*, whose length is generated statistically according to the 1D histogram array (as discussed in Section 3.1.3). This is done by inserting `usleep(duration)` calls, where `duration` is determined by randomly sampling the appropriate histogram. As the histograms capture inter-process latency variances, load imbalance is retained by recreating the "computation bubble" distribution across different processes. This is because: for most iterative parallel applications, every computation gap varies considerably across processes in the same timestep. For example, None Uniform Memory Access (NUMA) platforms such as Titan and Sith at ORNL, bring obvious underlying imbalance across processes. Meanwhile, total execution time of different timesteps for the same state possesses a negligible variation, as showed in Table 4.

The current APPRIME prototype ignores the captured $D$ phases, though it is straightforward to reproduce statistically similar "noises" according to the traced events.

# 4. EVALUATION

In this section, we evaluate our APPRIME prototype, implemented in over 16,000 lines of Java code. Our tests used two platforms of Oak Ridge Leadership Computing Facility (OLCF) supercomputer/clusters. Table 3 lists major configurations.

We validate its major design choices and demonstrate the output benchmarks' effectiveness using the aforementioned use cases. For use case 1 (cross-platform performance assessment), we used the four most communication-intensive members (BTIO, SP, CG, and LU) of the NAS benchmark suite [7]. For use case 2 (I/O configuration evaluation), we started with three large real-world applications (all proprietary), whose developers/users are interested in exploring asynchronous I/O through data staging: the quantum turbulence code BEC2 [31] and two gyro-kinetic particle simulations: XGC [26] and GTS [35].

| Name | # of nodes | Cores per node | Mem. per node | OS | File system |
|---|---|---|---|---|---|
| Titan | 18,688 | 16 | 32GB | Cray xk7 | Lustre |
| Sith | 40 | 32 | 64GB | Linux | Lustre |

**Table 3: Tested platforms**

A major focus of our evaluation is *accuracy*, measuring the similarity in behavior between the each original application $A$ and the APPRIME benchmark $A'$, potentially across multiple platforms. We examined not only the overall performance, but also the breakdown of time spent on computation, communication, and I/O. In addition, we checked whether the inter-processes load imbalance and overall performance variance existing in $A$ is retained by $A'$. Each test is run three times and we report the average, with error bars indicating standard deviation. All traces are collected with the SST DUMPI library [16].

## 4.1 Summary of Application Trace Characteristics

Before presenting results, we first give information on our seven test workloads. Table 4 summarizes key statistics collected in APPRIME's step-by-step trace processing. For each application, we list statistics from two sample executions, with 64 and 256 processes respectively. All NAS benchmarks use the D-class problem size.

We make the following observations. (1) With only 256 processes, parallel application runs produce sizable traces (1.1GB to 12GB). The size of event tables are smaller, but stays quite stable relative to the original trace size (around 70%). (2) APPRIME's compact string representation of event name sequence produces per-process strings under 100KB in most cases, enabling efficient phase recognition. (3) As reported by a previous study [33], each application uses a rather small subset of MPI functions (11 to 38 unique MPI routines). (4) Each NPB benchmark has only one $C$ phase state, creating homogeneous timesteps. In contrast, both of the two gyrokinetic simulations (XGC and GTS) have two states. (5) The same applications also contain small portions of noise ($D$ phases), which is not found in other applications or benchmarks we tested. They account for 0.1% and 0.3% of traced events respectively and appear safe to ignore in our benchmark creation. (6) At the end of its trace processing, APPRIME significantly reduces the traces into compact profiles for code generation (average 4.2MB, median 3.5MB). Note that the profile size only grows with the execution scale but not with the number of timesteps.

## 4.2 Justification of Methodology

Next, we validate APPRIME's design, by answering the following questions:

| App | # of procs | # of TSs | Trace size[§] | Table size[§] | # events in one state | String size | # unique funcs | # of states | D%[⋆] | TSV% [¶] | Profile size |
|---|---|---|---|---|---|---|---|---|---|---|---|
| BTIO | 64 | 250 | 832 MB | 584 MB | 183 | 44.4 KB | 16 | 1 | 0% | 2.1% | 2.2 MB |
| BTIO | 256 | 250 | 7.02 GB | 4.75 GB | 266 | 91.5 KB | 16 | 1 | 0% | 4.3% | 8.1 MB |
| CG | 64 | 100 | 1.42 GB | 1.00 GB | 789 | 77.8 KB | 11 | 1 | 0% | 1.5% | 3.4 MB |
| CG | 256 | 100 | 7.51 GB | 5.50 GB | 1478 | 101.2 KB | 11 | 1 | 0% | 1.8% | 11 MB |
| SP | 64 | 500 | 1.44 GB | 960 MB | 139 | 68.1 KB | 15 | 1 | 0% | 1.4% | 1.4 MB |
| SP | 256 | 500 | 11.7 GB | 7.43 GB | 278 | 138 KB | 15 | 1 | 0% | 3.5% | 6.1 MB |
| LU | 64 | 300 | 18 GB | 12.3 GB | 1604 | 471 KB | 11 | 1 | 0% | 2.3% | 11.3 MB |
| LU | 256 | 300 | 75 GB | 51.3 GB | 1604 | 471 KB | 11 | 1 | 0% | 3.8% | 44.2 MB |
| BEC2 | 64 | 100 | 142 MB | 101 MB | 74 | 7.5 KB | 14 | 1 | 0% | 1.8% | 1.1 MB |
| BEC2 | 256 | 200 | 1.08 GB | 800 MB | 74 | 14.7 KB | 14 | 1 | 0% | 2.7% | 3.6 MB |
| XGC | 64 | 100 | 262 MB | 243 MB | 73 | 11.5 KB | 28 | 2 | 0.1% | 4.3% | 0.98 KB |
| XGC | 256 | 200 | 2.1 GB | 1.64 GB | 103 | 15.3 KB | 28 | 2 | 0.1% | 5.8% | 1.40 MB |
| GTS | 64 | 50 | 213 MB | 137 MB | 391 | 11.6 KB | 38 | 2 | 0.3% | 5.6% | 1.9 MB |
| GTS | 256 | 100 | 1.83 GB | 1.15 GB | 391 | 24.9 KB | 38 | 2 | 0.3% | 5.9% | 7.2 MB |

†: timestep [§]: total [⋆]: percentage of traced events identified as $D$ phases [¶]: average relative time variation among timesteps within one execution on Sith

**Table 4: All seven applications' executions and trace features, their A' benchmarks are released online [1].**

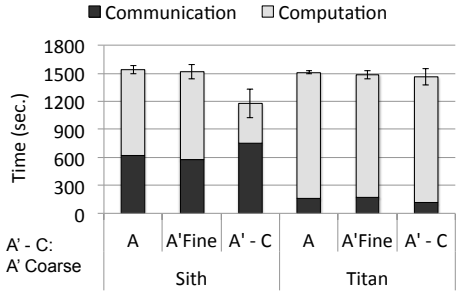



**Figure 8: Performance accuracy: coarse-grained vs. fine-grained benchmark generation of BEC2**

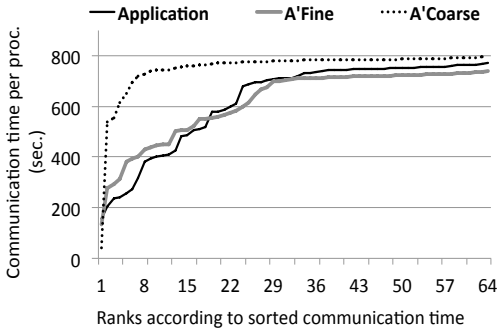



**Figure 9: Captured inter-process latency variances by SP on Sith**

1. Do we really need to identify individual (and potentially heterogeneous) timesteps in parallel simulations? Since we take a statistical approach and the number of timesteps is known, what difference will it make if we simply partition the overall traces by the timestep number?
2. Does the distribution-based code generation introduce variance not belonging to the original application?
3. Can APPRIME produce significantly more compact benchmarks compared to the original applications?
4. For applications with heterogeneous timesteps (in communication behavior), how soon can we recognize these states?

To answer the first question, we constructed a coarser-granularity alternative to the standard APPRIME approach. It statistically summarizes traces using lightweight profiling (our implementation uses mpiP [34]), collecting per process high-level statistical information for each MPI function called in the application. Similarly, the *computational intervals* are profiled at a coarser granule, for each type of (MPI) event, instead of identifying the accurate $C$ phase event sequences and profiling at the per-event level as in the standard approach. The coarse-grained benchmark for application $A$, which we label $A'$ *Coarse* (as opposed to $A'$ *Fine* for the standard output), is created by partitioning the event profiled uniformly into the known timestep counts and generating event sequence and parameter values according to the profiled distribution. Special care is taken to handle calls such as MPI_Isend, MPI_Irecv, and MPI_Waitall as atomic event groups, to avoid deadlocks.

Figure 8 reports the computation-communication time breakdown on both test platforms. $A$ in this case is the NAS SP application, which we find to contain a fair level of load variance across processes. As can be seen in Figure 8, the $A'$ *Coarse* benchmark has significantly lower fidelity than $A'$ *Fine*, which stays close to $A$ in overall computation/communication time on both platforms. In addition, $A'$ *Coarse* produces significantly larger execution time variance on Sith. As to be seen in Figure 10, standard APPRIME's randomized bin selection gives very small room for generating extra variances, while the coarse-grained version allows much more dynamic behavior at runtime.

Figure 9 further zooms into the communication load balance behavior. We sort the total measured communication time among the 64 processes, creating three monotonically growing curves for $A$, $A'$ *Fine*, and $A'$ *Coarse*, respectively. The $A'$ *Coarse* curve closely traces the $A$ curve, demonstrating its capability of recreating not only the overall execution time, but also the communication overhead distribution. The $A'$ *Coarse* curve, on the other hand, misses some of the skewed communication workload assignment and produces a more "balanced" curve. This and the previous set of results illustrate that simply relying on the high-level event

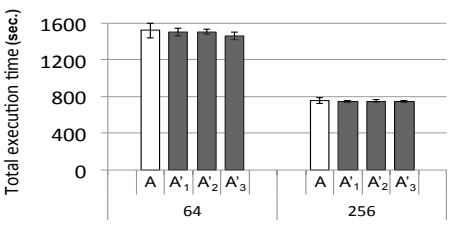



**Figure 10: BEC2's APPrime benchmark performance consistency**

occurrence and parameter distribution is not able to generate high-fidelity benchmarks.

Next, to answer the 2nd question, we examined the impact of APPRIME's randomized bin selection when performing distribution-guided benchmark code generation. Figure 10 illustrates the result using the BEC2 application, where we show the performance (plus performance variance) of the original application $A$, and three versions of APPRIME benchmarks ($A'_1$, $A'_2$, and $A'_3$) using three different random number generation seeds. The results confirm that the benchmarks do not incur additional variances. Actually, the $A'$ benchmarks show smaller performance variance compared to $A$, due to that they are all derived from the same instance of traced executions. Hopefully this limitation can be eliminated when we add real computation workload generation into APPRIME.

| App | Lines of code A | Lines of code A' | Max # of TS tested | Min # of TS required |
|---|---|---|---|---|
| BEC2 | 1.5K | 856 | 1000 | 1 |
| XGC | 93.7K | 7.7K | 1000 | 36 |
| GTS | 178.4K | 13.7K | 200 | 2 |

**Table 5: Statistics regarding timestep state identification, including the minimum number of timesteps required to capture all states**

Table 5 answers the last two questions, by giving the size (in lines of code) of the three real applications and their APPRIME counterparts, plus information on timestep state recognition. While the code size reduction is modest (less than half) for the already compact BEC2, there is a more than 10-fold reduction for both XGC and GTS. The generated code is in standard C and does not rely on libraries other than MPI or ADIOS I/O. As a result, the APPRIME benchmarks are very easy to port. In comparison, the original applications take a graduate student author experienced with parallel programs $8-24$ working hours to port even to a library-rich platform.

For XGC and GTS, each with two timestep states identified, the state set "convergence" happens rather early. According to Table 5, their second timestep states are discovered at the 36th and 2nd timestep respectively. We also executed all three real applications using their production run length given in Table 2: 1000 timesteps for BEC and XGC, and 200 timesteps for GTS, with no new timestep states discovered. This confirms our finding from interacting with scientists that the set of timestep behaviors is rather small and can be identified by running a short "prefix" of the application. To further verify the "predictability" of small APPRIME tracing runs, Figure 11 shows the effectiveness of using a relatively short tracing run (100 timesteps for BEC2 and XGC, 50 for GTS) to generate APPRIME benchmarks. The APPRIME benchmarks are shown to stay consistent with the original applications, even with runs much longer than traced ones.

## 4.3 Use Case Evaluation

### *4.3.1 Results: Cross-Platform Relative Performance*

We assessed APPRIME benchmarks' capability of retaining the original applications' relative performance across our two test platforms, Sith and Titan. Figure 12 gives all three phases (computation, communication and I/O) time cost and the total execution time's variation of the four NAS benchmarks tested, along with that of the corresponding APPRIME benchmarks.

Considering their different compute capability, we collected input traces for the original applications *on each platform* and generated the platform-specific *computational interval* distributions. Note that this handling does not live up to our use case objective (to avoid porting applications) and is meant to be fixed by adding real compute activity regeneration, as discussed in Section 6. Therefore, the assessment focus is on the cross-platform fidelity of the synthetic communication and I/O workloads. APPRIME benchmarks are shown in Figure 12 to closely match the original applications' relative behavior across applications, job scales, and platforms.

In addition, we verified that all the APPRIME benchmarks after the four NAS benchmarks produce *identical* event name sequences as the original applications do. By automatically creating benchmarks highly faithful to the original application, yet without disclosing original programs (either source or executable), APPRIME can help users quickly assess the capability of different platform choices for their target workloads.

### *4.3.2 Results: Asynchronous I/O Configuration Assessment*

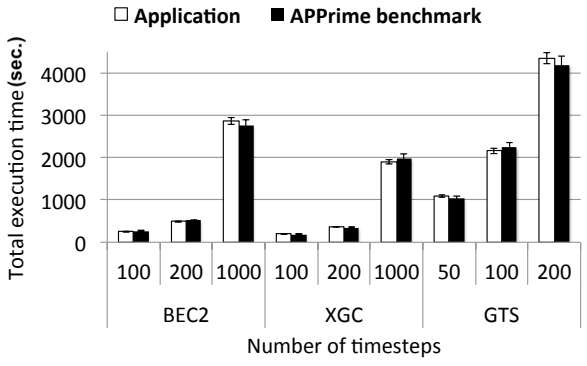
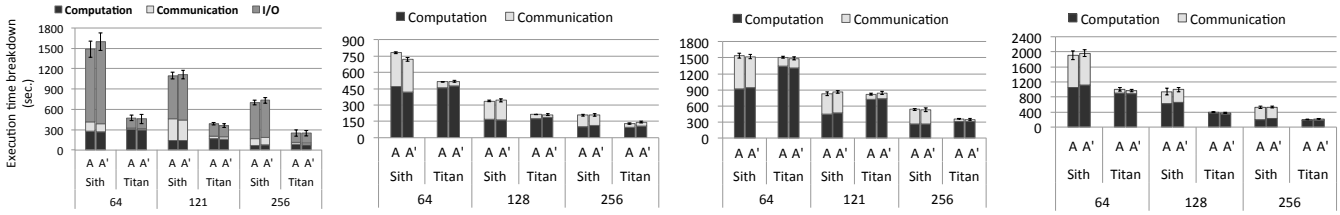
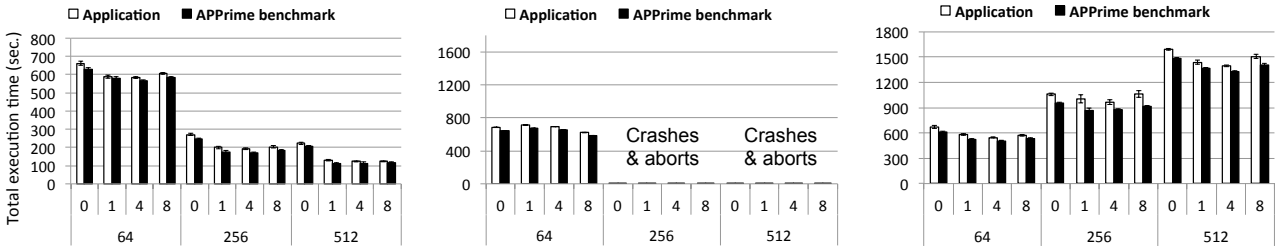



**Figure 11: Performance accuracy of APPrime benchmarks in long executions, with short traced runs for training**




(a) BTIO (b) CG (c) SP (d) LU

**Figure 12: Computation, communication and I/O phases time of selected NAS benchmarks on two platforms**




(a) BEC2 (b) GTS (c) XGC

**Figure 13: Total execution time with different numbers of asynchronous I/O staging processes**

In this section, we demonstrate that APPRIME benchmarks are able to help in examine the impact of different I/O methods, or even specific I/O settings. Our experiments compared the behavior of our three real-world applications along with their APPRIME counterparts, by enabling asynchronous I/O (data staging) via the ADIOS parallel I/O library [4]. In such async-I/O is adopted, the periodic output content will be collected by a small group of *staging nodes*, who write the data to the parallel file system while the compute processes resume their computation.

Figure 13 gives the execution time of BEC2, GTS, and XGC applications, with three different execution scales and four staging process settings. "0 staging process" indicates that synchronous I/O is used. As APPRIME retains the original configuration file of the application and reproduce the automatically identified I/O phases, the I/O settings can be configured in the same way as for the original applications. The results show that (1) the total execution time of APPRIME benchmarks remain faithful to the original applications, and (2) the relative impact of using different number of staging processes is corrected reflected by the APPRIME benchmarks. For example, the APPRIME benchmark runs correctly reveal that for BEC2, asynchronous I/O brings significant saving to the overall execution time, especially with larger-scale runs, while deploying more than one staging process does not bring much incremental benefit. As another example, for XGC APPRIME runs properly indicate that adopting 4 staging processes appear to be the configuration sweet point.

Note that GTS crashes when performing I/O with ADIOS in 256- and 512-process runs. After consulting the ADIOS team, we suspect the reason to be a problem with the application's I/O call arguments. To verify that APPRIME is able to reproduce such crashes, we generated the APPRIME benchmark with traces from an execution with I/O turned off, and inserted the I/O phase using the Skel tool [19] using the same GTS I/O configuration file. The benchmark, at both 256 and 512 scales, also aborted with errors during checkpoint output.

Finally, we also verified that the APPRIME benchmarks' reconfigurability in adjusting the periodic I/O frequency. Again they accurately reflect the original application's performance behavior (result chart omitted due to space limit).

# 5. RELATED WORK

**Tracing and Profiling** Communication trace collection and analysis tools (*e.g.*, [5, 29, 17, 21, 22]), plus post-mortem trace analysis and replay tools (*e.g.*, [8, 14]), have been widely used to understand/optimize large-scale HPC applications. To address the large size of parallel execution traces, several compression tools [40, 39, 22, 43] present in-situ lossless or lossy trace compression. Profiling tools [12, 34, 6] take a relatively light-weight approach, by summarizing aggregate or statistical information of parallel job executions. APPRIMEbuilds on trace collection and analysis, with the goal of understanding and simulating the *application* itself (rather than its certain executions) as stand-alone, portable, and to some extent configurable benchmark code.

**Trace-based Application Analysis** ScalaExtrap [37] successfully identifies and extrapolates communication topologies, given traces from executions of different scales, though with several constraints (requiring certain patterns such as stencil/mesh manner point-to-point communication). Currently APPRIME does not support automatic problem size or job size scaling, but can potentially leverage similar approaches to extrapolate application behavior observed at different scales. By taking a statistical approach,

such extrapolation can be applied to communication, computation, and I/O in a consistent way. However, the current strict and accurate event recreating per timestep might need to be relaxed.

There also exist trace analysis work on phase identification, *e.g.*, by applying signal processing techniques [9]. The authors demonstrated that the proposed tool reduces trace size effectively, but was not able to correctly identify HPC applications' iterative structures. Other automatic techniques (such as Stranger for PHP [42]) usually have strict constraints on the input string regarding grammar and language. The lack of fixed alphabetic or consistent iterative structure across different applications, as well as the noise phases found, motivates us to design our own algorithm for trace-based parallel iterative program phase recognition.

**Benchmark Generation** It is widely recognized (and confirmed in this study) that these benchmarks cannot fully keep up with state-of-the-art parallel applications in terms of problem size, execution scale, or behavior complexity. Existing automatic generation techniques, such as Benchmaker [11, 15] and HBench [27], also use statistical models to characterize the original applications using metrics like instruction set, memory access stride, and cache miss rate. With APPrime, we couple statistical modeling (for generating parameter values such as buffer sizes, computation intervals, and communication partners) with accurate event sequence matching and re-creation, to build highly realistic and somewhat reconfigurable parallel benchmarks. At the same time, there is prior work on benchmark extraction based on compressed communication trace, in particular ScalaBenGen [36]. However, it focuses on communication behavior recreation, requires in-house trace format, and generates benchmarks with hardcoded number of timestep and I/O frequencies and methods. Therefore we did not evaluate it in our experiments for comparison. The performance fidelity of APPrime is similar to that reported for ScalaBenGen. Meanwhile, APPrime captures the more high-level timestep phases (rather than loops heuristically identified at runtime), tolerates heterogeneous timestep behaviors, and works with standard trace collection tools as well as existing traces.

Finally, compiler-assisted source code reduction (such as slicing) [28, 44, 32] offers an alternative approach to benchmark creation. As discussed by other researchers [36], such static techniques have significant disadvantages even for creating communication-only benchmarks. For example, they rely on the availability of *all* source codes, including application and its dependent libraries. Skel [19] automatically generates skeletal I/O applications with ADIOS API from an abstraction of simulation I/O parameters. Compared to reduction-based and specialized code skeletons, APPrime aims at generating comprehensive, generic application benchmarks by analyzing and recreating the original applications' dynamic behavior.

# 6. CONCLUSION AND FUTURE WORK

This paper presents APPrime, which intelligently distills the behavior profile of parallel, iterative applications from execution traces and automatically generates compact, portable benchmark codes. APPrime takes a hybrid approach that couples fine-grained communication and I/O event sequence matching (for automatic timestep recognition) and statistical event modeling (for event parameter and computation interval regeneration). Through our experiments with macro-benchmarks and real-world large applications, we have verified that APPrime generates compact, portable benchmarks that retain the original applications' performance characteristics across multiple execution scales and platforms. Our study also indicates that today's applications possess complex behaviors (such as heterogeneous timesteps) not portrayed in popular parallel benchmarks such as the NPB suite.

Based on the validation results reported in this paper, our on-going work is investigating ways to regenerate synthetic computation activities simulating real-application workloads. This can be viewed as a recursive step within APPrime, where we "zooms into" the computation intervals and statistically regenerate integer/floating-point computation instructions (along with memory access patterns) as observed from the original application. Also, the current APPrime prototype focuses on temporal behavior study and has not yet studied creating benchmarks with similar *scalability behavior* when the number of processes is changed. We suspect that this can be done by learning an application's scaling behavior and applying techniques such as communication pattern extrapolation [37]. Finally, APPrime might be made more efficient and flexible by enabling online trace processing (to distill patterns and statistics for benchmark creation).

# 7. REFERENCES


[1] APPrime Website. `http://www.apprimecodes.com/`.

[2] DOE INCITE. `http://www.doeleadershipcomputing.org/awards/2015INCITEFactSheets.pdf`.

[3] OLCF Titan. `https://www.olcf.ornl.gov/titan/`.

[4] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. DataStager: Scalable Data Staging Services for Petascale Applications. In *Cluster Computing*, 2010.

[5] H. Adalsteinsson, S. Cranford, D. A. Evensky, J. P. Kenny, J. Mayo, A. Pinar, and C. L. Janssen. A Simulator for Large-Scale Parallel Computer Architectures. In *IJDST*, 2010.

[6] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *CCPE*, 2010.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. In *IJSA*, 1991.

[8] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In *ICCS*. Springer-Verlag, 2001.

[9] M. Casas, R. M. Badia, and J. Labarta. Automatic Phase Detection and Structure Extraction of MPI Applications. *IJHPCA*, 2010.

[10] D. P. Doane. Aesthetic Frequency Classifications. *The American Statistician*, 1976.

[11] J. Dujmović. Automatic Generation of Benchmark and Test Workloads. In *WOSP/SIPEW*, 2010.

[12] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca Performance Toolset Architecture. In *CCPE*, 2010.

[13] gtc2link. GTC-benchmark in NERSC-8 suite, 2013.

[14] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny. Using Simulation to Design Extremescale Applications and Architectures: Programming Model Exploration. *ACM IGMETRICS PER*, 2011.

[15] A. M. Joshi, L. Eeckhout, and L. K. John. The Return of Synthetic Benchmarks. In *SPEC Benchmark Workshop*, 2008.

[16] J. P. Kenny, G. Hendry, B. Allan, and D. Zhang. Dumpi: The mpi profiler from the sst simulator suite, 2011.

[17] A. Knupfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel. Introducing the Open Trace Format (OTF). Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006.

[18] R. Latham, C. Daley, W. keng Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary. A case study for scientific i/o: improving the flash astrophysics code. *CSD*, 5(1):015001, 2012.

[19] J. Logan, S. Klasky, H. Abbasi, Q. Liu, G. Ostrouchov, M. Parashar, N. Podhorszki, Y. Tian, and M. Wolf. Understanding I/O Performance Using I/O Skeletal Applications. In *Euro-Par*. Springer-Verlag, 2012.

[20] NASA. Nas parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html, 2003.

[21] M. Noeth, F. Mueller, M. Schulz, and B. de Supinski. Scalable compression and replay of communication traces in massively parallel environments. In *IPDPS*, 2007.

[22] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. ScalaTrace: Scalable Compression and Replay of Communication Traces for High-Performance Computing. *J. Parallel Distrib. Comput.*, 2009.

[23] F. Pachet, P. Roy, and G. Barbieri. Finite-length Markov Processes with Constraints. In *IJCAI*, 2011.

[24] M. Pedersoli, A. Vedaldi, and J. Gonzalez. A coarse-to-fine approach for fast deformable object detection. In *IEEE CVPR*, pages 1353–1360, 2011.

[25] L. R. Rabiner and B. H. Juang. An introduction to hidden Markov models. *IEEE ASSP Magazine*, pages 4–15, January 1986.

[26] S. Ku, C. S. Chang, and P. H. Diamond. Full-f Gyrokinetic Particle Simulation of Centrally Heated Global ITG Turbulence from Magnetic Axis to Edge Pedestal Top in A Realistic Tokamak Geometry. *Nuclear Fusion*, 2009.

[27] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang. The Case for Application-Specific Benchmarking. In *IEEE HOTOS*, 1999.

[28] S. Shao, A. K. Jones, and R. Melhem. A Compiler-based Communication Analysis Approach for Multiprocessor Systems. In *IEEE IPDPS*, 2006.

[29] S. Shende and A. D. Malony. TAU: The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2), 2006.

[30] H. A. Sturges. The Choice of a Class Interval. *Journal of the American Statistical Association*, 1926.

[31] G. Vahala, M. Soe, B. Zhang, J. Yepez, L. Vahala, J. Carter, and S. Ziegeler. Unitary Qubit Lattice Simulations of Multiscale Phenomena in Quantum Turbulence. In *SC11*, 2011.

[32] L. Van Ertvelde and L. Eeckhout. Dispersing Proprietary Applications as Benchmarks Through Code Mutation. *ACM SIGOPS OSR*, 2008.

[33] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *IPDPS*, 2002.

[34] J. S. Vetter and M. O. McCracken. Statistical Scalability Analysis of Communication Operations in Distributed Applications. *ACM SIGPLAN*, 2001.

[35] W. X. Wang and Z. Lin and W. M. Tang and W. W. Lee and S. Ethier and J. L. V. Lewandowski and G. Rewoldt and T. S. Hahm and J. Manickam. Gyro-kinetic Simulation of Global Turbulent Transport Properties in Tokamak Experiments. *Physics of Plasmas*, 2006.

[36] X. Wu, V. Deshpande, and F. Mueller. ScalaBenchGen: Auto-Generation of Communication Benchmarks Traces. In *IEEE IPDPS*, 2012.

[37] X. Wu and F. Mueller. ScalaExtrap: Trace-based Communication Extrapolation for SPMD Programs. In *ACM PPoPP*, 2011.

[38] X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. Roth. Probabilistic communication and i/o tracing with deterministic replay at scale. In *ICPP*, 2011.

[39] Q. Xu and J. Subhlok. Construction and Evaluation of Coordinated Performance Skeletons. In *HiPC*. Springer-Verlag, 2008.

[40] Q. Xu, J. Subhlok, R. Zheng, and S. Voss. Logicalization of Communication Traces from Parallel Execution. In *IEEE IISWC*, 2009.

[41] L. T. Yang, X. Ma, and F. Mueller. Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution. In *SC05*, 2005.

[42] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An Automata-Based String Analysis Tool for PHP. In J. Esparza and R. Majumdar, editors, *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[43] J. Zhai, J. Hu, X. Tang, X. Ma, and W. Chen. Cypress: Combining static and dynamic analysis for top-down communication trace compression. In *SC14*, 2014.

[44] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng. FACT: Fast Communication Trace Collection for Parallel Applications Through Program Slicing. In *SC09*, 2009.