# Chameleon: Online Clustering of MPI Program Traces

Amir Bahmani Frank Mueller
*Department of Computer Science, North Carolina State University, Raleigh, NC, Email: mueller@ncsu.edu*

***Abstract*—The data explosion in scientific computing applications continues to fuel increasing demand for computational power. Understanding application behavior in this context becomes essential to determine shortcomings, e.g., by collecting detailed information with tracing toolsets. This work considers parallel applications using the SPMD (single program multiple data) paradigm that relies on iterative kernels. This characteristic provides an opportunity to empower tracing toolsets with effective machine learning algorithms. One solution is to cluster processes with the same behavior into a group. Instead of collecting performance information from each individual process, this information can be collected from just a set of lead processes, i.e., one lead process per group. This work, called Chameleon, contributes an $online$, $fast$, and $scalable$ signature-based clustering algorithm. Unlike related work, namely ScalaTrace V2, that generates the compressed $global$ trace within $MPI\_Finalize$, Chameleon creates this trace incrementally during the execution of applications and only for lead processes. Chameleon also identifies different program phases, clusters processes exhibiting different execution behavior, and creates a compressed global trace file on-the-fly, all incrementally at interim execution points of applications. The resulting system combines low overhead at the clustering level a lower time complexity of $log(P)$ than prior work.**

***Keywords*-High-Performance Computing, Message Passing, Tracing, Clustering Algorithms**

## I. Introduction

The data explosion in scientific computing applications continues to fuel an ever increasing demand for computational power. Computer scientists have started working to subjugate this power hungry dragon by understanding the behavior of parallel applications. To better understand the behavior of MPI applications, communication traces often provide the insight to detect inefficiencies and help in problem tuning [26], [6].

Today's tracing tools either obtain lossless trace information at the price of limited scalability (e.g., Vampir [7], Tau [23], Intel's toolsets [13], and Scalasca [10]) or preserve only aggregated statistical trace information to conserve the size of trace files (e.g., mpiP [25]). As a result, trace file sizes can easily exceed multiple gigabytes, even for regular single-program multiple-data (SPMD) codes, e.g., 5TB for SMG2k for moderately small input sizes [34]. In response, a number of communication compression techniques have been designed, including run-length compression [35] and structural compression [16], [22], [30].

At extreme scale, tracing tools, linked with applications, could severely affect the efficiency and scalability of the system. The tracing background workload may compete with the application for resources, which can perturb the application's behavior. Moreover, due to the large I/O requirement of tracing data required for applications on top-end HPC platforms, collecting detailed performance information comprehensively may not be feasible from a scalability perspective. Hence, tool designers need new strategies to address these problems.

Bahmani and Mueller [1], [2], [3] showed that at scale, large groups of processes even in task parallel applications behave similarly. Moreover, the iterative nature of parallel applications provides an opportunity to use clustering algorithms to group processes with the same application behavior. Hence, instead of collecting performance information from all individual processes, such information can be collected from just a single node (or a set of lead processes) per cluster group.

This paper proposes an $online$, $fast$, and $scalable$ signature-based clustering framework called Chameleon. Our Chameleon framework, similar to real-life Chameleons, rapidly adapts to changing contexts (i.e., program phase changes in our case). Chameleon clusters processes exhibiting similar execution behavior on-the-fly. We apply our clustering algorithm on trace files created by the public release of ScalaTrace V2 [30], a state-of-the-art MPI message passing tracing toolset that we enhanced based on the public release to create Chameleon. ScalaTrace V2 provides orders of magnitude smaller if not near-constant sized communication traces regardless of the number of nodes while preserving structural information. Henceforth, we refer to ScalaTrace V2 simply as ScalaTrace.

ScalaTrace employs a two-stage trace compression technique, namely intra-node and inter-node compression [22], [31]). It utilizes Regular Section Descriptors (RSDs) to capture the loop structures of one or multiple communication events. Power-RSDs (PRSDs) are utilized to recursively specify RSDs in nested loops (see Section II). After each node has created its own compressed trace file and the program is completing, ScalaTrace performs an inter-node compression over a radix tree rooted in rank 0. During this reduction, internal nodes combine their traces with other task-level traces that they receive from child nodes. While intra-compression is fast and efficient, inter-node compression is a costly operation with O($n^2 \log P$) time complexity, where $n$ is the number of MPI events in PRSD compressed notation and $P$ is the number of processes. Our clustering addresses the high overhead when scaling out by

significantly reducing $P$ to almost a constant value, which effectively eliminates this bottleneck.

Chameleon considers special MPI collective calls as a marker at interim execution points, e.g., at timestep boundaries of scientific codes. The marker helps to engage in clustering during the execution of the program. The proposed system has two main components. The first employs a *transition graph* that keeps track of the status of the system, and identifies when to call clustering. This component works based on the $Call - Path$ signature of MPI events. We use the stack signature to distinguish events originating from different call sequences with associated call paths. The $Call - Path$ signature is the aggregated composition of stack signatures of different events. Chameleon calculates $Call-Path$ signatures of events added between two marker calls. Then, the transition graph assists Chameleon to wisely decide on re-clustering or skipping clustering based on the changes of $Call - Path$.

The second component of Chameleon is merging and creating the global trace on-the-fly. Unlike ScalaTrace, which uses intra-compression during the execution of the program and then inter-compression at $MPI\_Finalize$ to create the global trace, Chameleon runs both compression techniques on-the-fly by calling inter-node compression at interim execution points. In the inter-node compression in ScalaTrace, all $P$ processes participate over a radix tree to create the global trace. On the other hand, Chameleon identifies program phase changes, clusters processes, and only considers $K$ lead traces in an online inter-node compression step. These $K$ traces are created on-the-fly by the intra-compression step between two consecutive marker calls.

Chameleon merges the output of each $online$ inter-compression with a trace called $online$ trace in the root (node 0). The $online$ trace incrementally expands to an equivalent output of $MPI\_Finalize$ in the original ScalaTrace. These algorithms are discussed in detail in the following sections. The objective of clustering in Chameleon is to drastically reduce the ScalaTrace overhead (both in execution time and space), and to select representative processes resembling the overall execution time of the original application without clustering.

**Contributions:**

- We design Chameleon, a low-overhead clustering framework that utilizes signature-based clustering algorithms to cluster processes and creates a singular trace file by combining intra-node and inter-node compression on-the-fly.
- We describe a transition graph that assists Chameleon to identify phase changes to subsequently engage in re-clustering.
- We evaluate Chameleon for a set of HPC benchmarks showing its effectiveness at capturing representative application behavior for communication events.
- We compare the accuracy of traces generated by Chameleon with ScalaTrace.
- We explain how Chameleon opens a new horizon in terms of space complexity with a potential for increased energy efficiency for tracing MPI applications.

## II. Background

Our work builds on ScalaTrace as an MPI tracing toolset. Here, we briefly introduce several of the key ideas and techniques used in ScalaTrace.

ScalaTrace captures MPI events in the innermost loop as Regular Section Descriptors (RSD), while power-RSDs capture RSDs (PRSDs) of higher-level loop nests represented as a constant sized data structure [22]. Consider the example in the following code snippet:

```
for i = 0 → 1000 do
  for k = 0 → 100 do
    MPI_Send(...);
    MPI_Recv(...);
  end for
  MPI_Barrier(...)
end for
```

Trace compression with PRSDs results in the following tuples: RSD1:<100, MPI_Send1, MPI_Recv1> denotes a loop with 100 iterations of alternating send/receive calls with identical parameters (omitted here), and PRSD1:<1000, RSD1, MPI_Barrier1> denotes 1000 invocations of the former loop (RSD1) followed by a barrier.

ScalaTrace has the following three main properties: (1) It provides location-independent encodings: Communication end-points (task IDs) in SPMD programs often differ from one node to another. However, their position relative to the MPI task ID often remains constant. Therefore, ScalaTrace leverages relative encodings of communication end-points, i.e., an end-point is denoted as $\pm$c for a constant $c$ relative to the current MPI task ID [22].

(2) ScalaTrace features calling sequence identification: MPI calls, such as a Send, may be scattered over various locations in a program. To distinguish between events from different locations, just recording the MPI event type itself is insufficient. ScalaTrace captures the calling context by recording the calling sequence that leads to the MPI event, which is obtained from the stack backtrace of an MPI event. Each location is represented as a unique signature of the stack trace called the stack signature [22].

(3) ScalaTrace provides communication group encoding: it leverages a special data structure called ranklist to represent a communication group. Using EBNF notation, a rank list is represented as ⟨dimension, start_rank, iteration_length, stride, iteration_length, stride⟩, which denotes the dimension of the group, the rank of the starting node, and the iteration and stride of the corresponding dimension, respectively [28].

ScalaTrace employs a two-stage trace compression technique, namely intra-node (loop level) and inter-node compression. The latter is consolidating traces in a reduction step over a radix tree in the $MPI\_Finalize$ PMPI wrapper. While intra-compression is fast and efficient, inter-

compression is costly as it depends on the number of tasks. Bahmani and Mueller [1], [2], [3] developed signature-based clustering algorithms, which lowered this overhead. Figure 1 depicts the main components of ScalaTrace with clustering. The left two components are executed for each process, and the right two components are executed over a radix tree among a group of processes. Clustering components are the top two (green) components. Clustering procedures are also called in the $MPI\_Finalize$ PMPI wrapper. After calling clustering, the inter-node compression step only considers a small group of lead traces.

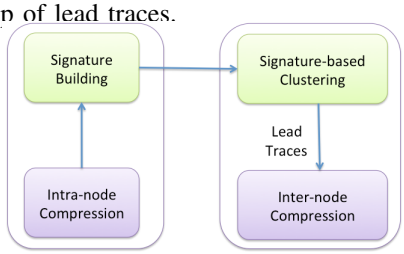



Figure 1: A Schematic of Signature-based Clustering

## III. The Proposed Clustering Algorithm

In the design of Chameleon, we considered two main steps: (1) an online clustering algorithm helps to group processes with the same execution behavior, and (2) partial representative traces are merged to create the global trace on the fly.

To implement the first step of Chameleon, we need to identify the times and locations in the program where the clustering algorithm is called. We refer to these specific calls as $Markers$. We assume markers are special MPI collective events, which trigger clustering during the execution of the program at interim execution points under special conditions. In Chameleon, we consider $MPI\_Barrier$ as the marker. To distinguish the marker with other $MPI\_Barrier$ calls in MPI programs, Chameleon assigns a unique value to the communicator field.

Many parallel codes report progress at the end of kernel loops or timesteps (e.g., NAS parallel benchmarks, POP, LULESH, Sweep3D). We insert our marker in this progress reporting point. Since clustering can have significant execution cost, we need a mechanism to track phase changes in execution behavior. To this end, Chameleon considers the calling context using its $Call-Path$ signature and identifies new phases by previously unseen signatures.

To capture the calling context, ScalaTrace uses the stack signature consisting of a number of backtrace addresses of the program counter (return addresses), one for each stack frame. After creating each frame's stack signature, in order to create the 64-bit $Call-Path$ signature, Chameleon computes the exclusive or ($XOR$) of all 64-bit stack signatures. Moreover, to order events, it multiplies the modulo 10 plus 1 of the sequence number of each event by the 64-bit stack signature and then uses this value in the $Call-Path$ signature. This ensures that signatures cannot cancel out each other due to permutations on call sequences and recursion.

Chameleon keeps track of MPI events between two consecutive calls of the marker. Every time the program calls the marker, Chameleon creates the $Call-Path$ signature of added events between two calls.

Figure 2 depicts the transition graph in Chameleon. There are two MPI calls to the clustering algorithm. One is the marker and the other is $MPI\_Finalize$. The blue ellipse in the transition graph shows the status of marker calls. The marker could be in four different states, **All Tracing** ($AT$), **Clustering** ($C$), **Lead** ($L$) or **Final** ($F$). For every marker, each process independently creates its $Call-Path$ signature. Then, all processes participate in a collective voting procedure. If all processes observed a repetitive behavior (i.e., their current $Call-Path$ matched the last one), then they will enter the $C$ state. Otherwise, they will stay in $AT$. Clustering only happens in state $C$, and for each cluster of processes, a single process is declared as the "**lead**". Once in $C$, a mismatch in the next $Call-Path$ changes the state back to $AT$, while a second matching $Call-Path$ transitions to state $L$. In $L$, only lead processes continue to trace ("lead" flag is true) while non-lead processes in $L$ state set the "lead" flag to false and discontinue tracing temporarily (but remain in $L$). Should a lead process (with **lead** flag set) detect that its own $Call-Path$ has changed from the repetitive sequence, then the traces of lead processes up to this point are merged and all processes enter the $AT$ state. Should the trace end at any time (no more events), state $F$ is entered.

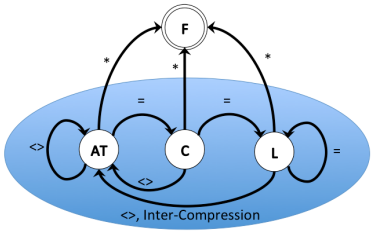



Figure 2: Transition Graph

Figure 3 presents sample code (left side) and the corresponding state transitions (right side). Four nodes start in the all tracing $AT$ state, they then transition to the clustering $C$ state since they have the matching $Call-Path$ signatures (i.e., a repetitive sequence of events in the 1st loop). They then enter the leader $L$ state, node 1 keeps tracing while the other non-representative processes turn off tracing (depicted as hollow rectangles). Nodes remain in $L$ state until $Call-Path$ signatures change (2nd loop), at which time Chameleon flushes all partial traces, merges the output with the global trace, and enters $AT$ for all nodes. After observing two distinct patterns, state $C$ and then $L$ are entered, but now with two lead nodes, 1 and 3 (representing if then and else branches). Upon calling MPI_Finalize, the final inter-compression step is run.

Algorithm 1 presents the pseudocode for the transitions of the graph. Creating a signature has $O(n)$ complexity, where

**ALGORITHM 1:** Phase Recognition of Online Clustering

**Input** : A Sequence of Compressed MPI Events (PRSDs)
**Output**: Clustering Status
**Initialization**: OldCallPath=0; Re-Clustering Flag = $true$; Lead Flag = $false$;

```
1  CurrentCallPath = Create signatures of input sequence of MPI events;
2  if OldCallPath == 0 then
3  |   //First time hitting the marker
4  |   Set OldCallPath = CurrentCallPath;
5  |   return AT;
6  end
7  Set tempReduceVal = 0;
8  Set globReduceVal = 0;
9  if OldCallPath != CurrentCallPath then
10 |   tempReduceVal = 1;
11 end
12 globValReduce = Sum all tempReduceVals using MPI_Reduce;
13 MPI_Bcast globValReduce by rank root;
14 Set OldCallPath = CurrentCallPath;
15 if globValReduce == 0 then
16 |   if Re-Clustering Flag == true then
17 |   |   //Clustering Phase
18 |   |   Re-Clustering Flag = false;
19 |   |   return C;
20 |   end
21 |   else
22 |   |   //Lead Phase w/o inter-compression
23 |   |   Lead Flag = true;
24 |   |   return AT;
25 |   end
26 end
27 if Lead Flag == true then
28 |   //Lead Phase w/ inter-compression
29 |   Lead Flag == false;
30 |   return L;
31 end
32 Re-Clustering Flag = true;
33 return AT;
```

$n$ is the number of MPI events in PRSD-compressed notation generated by intra-node compression (loop-compression). $n$ is equal to the number of disjoint stack signatures over all processes (e.g., 3 disjoint instructions in the sample example from the background section create at most 3 disjoint events over 1000 iterations). $n$ is much smaller than the number of processes at large scale ($N << P$). $MPI\_Reduce$ and $MPI\_Bcast$ are $O(log(P))$. Therefore, the complexity of this operation is $O(nlog(P))$.

Bahmani and Mueller [1], [2], [3] used K-Farthest, K-Medoid and multi-level hierarchical clustering algorithms. As previously mentioned, clustering happens over signatures, not traces. There are two phases. First, clustering using signatures, and then lead traces are merged.

Algorithm 3 depicts the main body of Chameleon. Once processes reach a marker, they invoke this algorithm. In the Chameleon implementation, we only consider $Call-Path$, $SRC$ and $DEST$ signatures. By analyzing related work [1], [2], [3], we found that these three 64-bit signatures are the most important ones, especially $Call-Path$. These signatures often cover other parameters as well (e.g., count, loop size, tag). To create $SRC$ and $DEST$ signatures, Chameleon averages parameter signatures composing $SRC$ and $DEST$ parameters of the MPI call events. Because aggregating event values and then taking the average could result in an overflow, we utilized an estimation function.

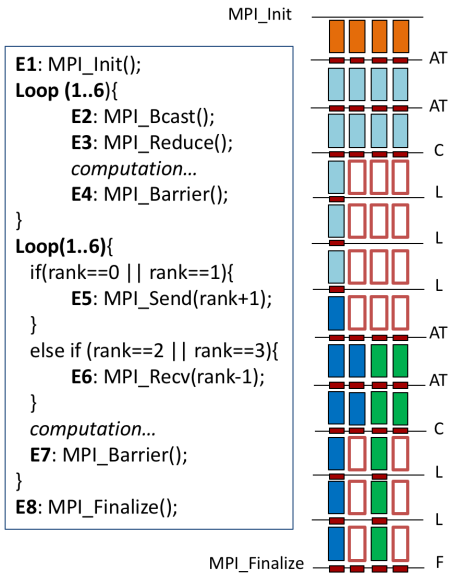


Figure 3: Sample Transitions

Chameleon ensures that after each process creates three signatures in Algorithm 1, it participates in clustering ("Clustering" state). If a process has any child, it receives the signatures from left and right children, and merges them with its own map of signatures (i.e., the data structure is a hashmap of <signature, ranklist>). Then, to cover all the events, it picks $\frac{K}{Num_{Call-Path}}$ lead processes from each $Call-Path$ cluster. Bahmani and Mueller [1], [2], [3] showed that the number of $Call-Path$ usually is below 9, which is sufficient to cover stencil codes. They studied the impact of increasing the number of clusters on the accuracy of traces. They observed that the key element with respect to trace accuracy is the number of Call-Path clusters. Covering all distinct events over all traces results in acceptable accuracy. Chameleon does not miss any MPI event by selecting at least one representative from each call-path cluster. It dynamically increases the value of $K$ should the number of different $Call-Path$ signatures exceed $K$.

Algorithm 2 helps to find the top $\frac{K}{Num_{Call-Path}}$ lead processes based on $SRC$ and $DEST$ signatures for each $Call-Path$ signature. Users could select any clustering algorithm (e.g., K-Medoid, K-Furthest, K-Random selection). Bahmani and Mueller in [3] compared K-Medoid and K-Furthest clustering and observed that the accuracy of traces is very close for these clustering algorithms. After picking lead processes, other non-selected clusters are merged with the closest clusters. Finally, if the process has any parent, it will send the information about clusters and lead processes with their signatures to the parent (i.e., cluster hashmap is a tuple <lead rank, ranklist>).

We highlighted multiple lines in Algorithm 3:
**(1)** The complexity of creating signatures is $O(n)$. We

**ALGORITHM 2:** Find Top K

**Input** : $K$ and $SRC$/$DEST$ signatures
**Output**: $TopK$ list

```
1  Calculate distance matrix for Top K list based on SRC and DEST;
2  TopK list = { } ;
3  while Size of TopK list < K do
4  |  Find farthest cluster to TopK list;
5  end
6  foreach cluster ∈ AllNode list - TopK list do
7  |  Find closest cluster;
8  |  Assign cluster to closest one;
9  end
```

introduced an input parameter called $Call\_Frequency$. This parameter gives users the option to control the number of times Chameleon creates signatures and calls the transition graph. We discuss this in the experimental section.

**(2)** In $AT$ state, only Algorithm 1 is executed at the marker.

**(3)** Only under "Clustering" state are lines 7-24 executed. The root node then broadcasts the list of top $K$ ranks.

**(4)** Under both "Clustering" and "Lead", only lead processes executes lines 25-35. Before merging starts, each lead process replaces the ranklist of events with the ranklist of its cluster (e.g., if a cluster contains ranks 0, 1, 2, 3, 4, and 5 with ranklist <1 1 0 5 1>, and the lead process is 5 with ranklist <1 1 5 1 0>, then rank 5 must update its trace with the cluster ranklist).

**(5)** After merging the top $K$ lead traces, Chameleon needs to merge the output of inter-node compression with the $online$ trace. As previously mentioned, the $online$ trace is the incremental global trace. Rank 0 is responsible to keep the $online$ trace. At this stage, it is possible that the root of the radix tree differs from rank 0. If so, the root rank sends the partial trace to rank 0, and rank 0 merges the partial trace with the $online$ trace.

**(6)** Under $C$ and $L$, at the end of each marker, all processes start over by removing their partial intra-node trace. Processes only need to keep the stack signature of the last event so that ScalaTrace considers the computation time between the last event and the new event. (7) Because of the synchronization (Broadcast+Reduction) in Algorithm 1, all processes receive the same state value. They could have different execution behavior between calls, but the synchronization step guarantees they are in the same state with respect to clustering.

At the end of the application, in $MPI\_Finalize$, Algorithm 3 is called with a small modification to add the last events to the $online$ trace. The only difference is that there is no need for Algorithm 1 because at least one new event has been added (i.e., $MPI\_Finalize$). Therefore, the $Call - Path$ is definitely different from the previous clustering, so re-clustering must be triggered but the inter-compression part remains the same. Note that communication for clustering occurs within PMPI pre- and post-wrappers of the maker.

As previously noted, ScalaTrace's inter-compression is a

**ALGORITHM 3:** Online Inter-Compression using Clustering

**Input** : A Sequence of Compressed MPI Events (PRSDs), Call_Frequency
**Output**: $Online$ Trace

```
1  Increment Marker_Call_Counter;
2  if Marker_Call_Counter % Call_Frequency != 0 then
3  |  return;
4  end
5  ClusteringState = Call Algorithm 1;
6  if ClusteringState==C OR ClusteringState==L then
7  |  if ClusteringState==C then
8  |  |  if a left/right child exists then
9  |  |  |  Receive list of left_K / right_K clusters;
10 |  |  |  Receive signature of head of top left_K / right_K clusters;
11 |  |  |  Merge left_K / right_K clusters + yourself into AllNode list;
12 |  |  |  if left_K + right_K + 1 > K then
13 |  |  |  |  local K = K / Number of Call-Paths;
14 |  |  |  |  foreach Call-Path signature do
15 |  |  |  |  |  TopK = TopK + findTopK(local K, SRC and DEST signatures);
16 |  |  |  |  end
17 |  |  |  end
18 |  |  end
19 |  |  if a parent exists then
20 |  |  |  Send current list of K clusters to parent;
21 |  |  |  Send signature of head of top K clusters to parent;
22 |  |  end
23 |  |  MPI_Bcast (Top K) by root;
24 |  end
25 |  if my rank ∈ Top K list then
26 |  |  Replace ranklist of collected events with my cluster ranklist;
27 |  |  tempRank = assign a temp rank from Top K;
28 |  |  if a left/right child exists then
29 |  |  |  Receive the left/right child traces;
30 |  |  |  Merge their trace with yours;
31 |  |  end
32 |  |  if a parent exists then
33 |  |  |  Send your trace to parent;
34 |  |  end
35 |  end
36 |  if my rank == root rank of Top K list OR rank == 0 then
37 |  |  if root of Top K list != 0 then
38 |  |  |  if rank == 0 then
39 |  |  |  |  Receive partial global trace from root of top K;
40 |  |  |  end
41 |  |  |  else
42 |  |  |  |  //root of Top K list
43 |  |  |  |  Send partial global trace to rank 0;
44 |  |  |  end
45 |  |  end
46 |  |  if my rank == 0 then
47 |  |  |  Merge partial trace with Online Global Trace;
48 |  |  end
49 |  end
50 |  //All nodes;
51 |  Delete your partial trace;
52 end
```

costly operation with $O(n^2 \log P)$ complexity, where $n$ is the size of the PRSD-compressed intra-node event trace and $P$ is the number of processes. Our logarithmic algorithms find the top $K$ traces and change cost to $O(n^2 \log K)$. Recall that $K$ is usually a constant value (e.g., 9 for stencil code).

The complexity of the K-Medoids algorithm is $K^3$. Each process in Chameleon at most consider $2K + 1$ items for a

constant $K$. Therefore, the clustering part of Algorithm 3 has the time complexity of $O(nlogP)$, once added, the $online$ inter-compression complexity is $O(r \times n^2 \log K)$, where $K$ is the number of lead processes, and $r$ is the number of re-clustering. Experiments in the next section show that both $r$ and $K$ are small numbers for real-world applications.

## IV. Experimental Setup

We utilized a 108 node cluster computer to conduct experiments. All machines were 2-way SMPs with AMD Opteron 6128 processors with 8 cores per socket. Nodes are connected by QDR InfiniBand. This is the largest platform we were able to obtain access to at this time. We tested Chameleon and "without clustering", which is the default version of ScalaTrace, for the $NAS$ Parallel Benchmarks ($NPB$), Sweep3D and the Parallel Ocean Program (POP). Each experiment was run five times, and the average value and standard deviation are reported. The aggregated wall-clock times across all nodes for the mentioned benchmarks is calculated and reported.

We conducted experiments with the NPB suite (version 3.3 for MPI) with class D input size [5] and Sweep3D [17]. Sweep3D is a solver for the 3-D, time-independent, particle transport equation on an orthogonal mesh. It uses a multidimensional wavefront algorithm for "discrete ordinates" in a deterministic particle transport simulation. In our experiments, the problem size is 100×100×1000. The Parallel Ocean Program (POP) [14] is an ocean circulation model developed at Los Alamos National Laboratory. Our experiments exercise a one degree grid resolution in which the problem size is 896×896 blocks and the individual block size is 16×16. ElasticMedFlow (EMF) is a generic framework for representing and executing medical application pipelines in parallel [4] with a master-worker paradigm with mpi4py [8] (MPI for python) atop MPI. We created a sample DNA prepossessing pipeline of 9 stages with problem size of 1000 patient datasets. For each patient, four DNA sequences are read, i.e., 1000×4×9 tasks are spawned. We modified mpi4py to support ScalaTrace and Chameleon.

## V. Results and Analysis

Table I depicts the number of clusters considered for the experiments (determined a priori [2]).

Table I: # of Clusters for the Tested Benchmarks

| Pgm | BT | LU | SP | POP | S3D | LUW | EMF |
|---|---|---|---|---|---|---|---|
| K | 3 | 9 | 3 | 3 | 9 | 9 | 2 |

Table II indicates the number of executed marker calls over the entire program run (with the number of processes indicated as (P) in parentheses). Notice that the number of required clusterings is only one for all the tested benchmarks. The number of times being in "Lead" state accounts for over 70% over the total number of marker calls. This percentage increases by increasing the number of marker calls. Note, LUW denotes LU under weak scaling. (Note that $P$ for EMF represents one master process and $P-1$ worker processes.)

Table II: # of Marker Calls, and # of times being in states Clustering(C), Lead(L) and All Tracing(AT)

| Pgm (P) | # Iters. | #Freq. | #Calls | #C | #L | #AT |
|---|---|---|---|---|---|---|
| BT(1024) | 250 | 25 | 10 | 1 | 8 | 1 |
| LU(1024) | 300 | 20 | 15 | 1 | 11 | 3 |
| SP(1024) | 500 | 20 | 25 | 1 | 21 | 3 |
| POP(1024) | 20 | 1 | 20 | 1 | 16 | 3 |
| S3D(1024) | 10 | 1 | 10 | 1 | 7 | 2 |
| LUW(1024) | 250 | 25 | 10 | 1 | 8 | 1 |
| EMF(126) | 288 | 32 | 9 | 1 | 6 | 2 |
| EMF(251) | 144 | 16 | 9 | 1 | 6 | 2 |
| EMF(501) | 72 | 8 | 9 | 1 | 6 | 2 |
| EMF(1001) | 36 | 4 | 9 | 1 | 6 | 2 |

The results in Tables I and II show that $P-K$ processes were idle for more than 70% of the execution of markers. Moreover, Chameleon did not use any of these $P-K$ traces in creating the $online$ trace. Overall, this may result in a reduction of energy, but still requires the idle processors to wait for those that obtain traces in their place. (The impact of Chameleon on space complexity and energy efficiency is described at the end of this section.)

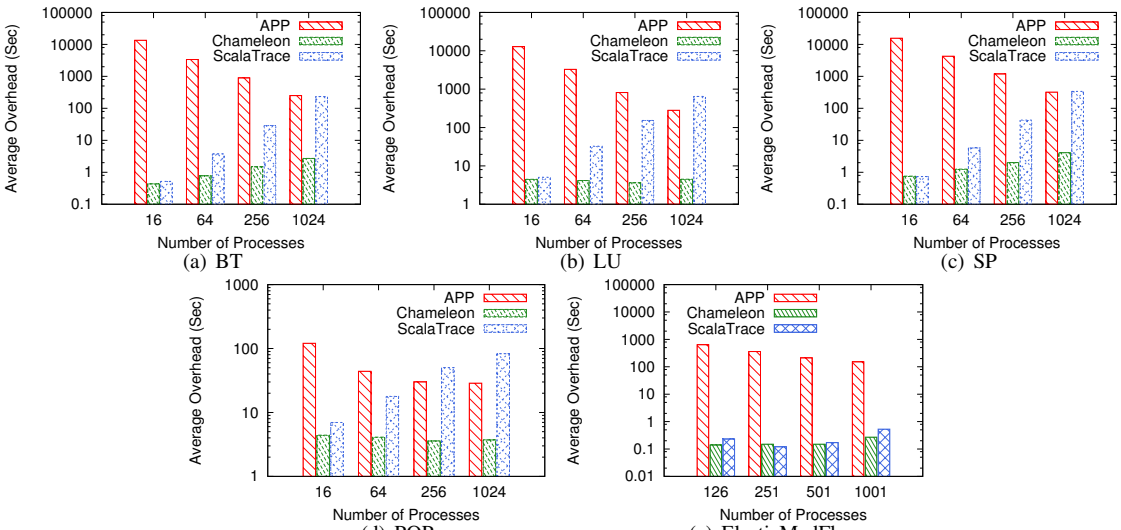
*Observation 1: By obtaining traces only from the lead process of each cluster, nearly no tracing overhead is induced for the majority of processors.*

Two experiments assess the accuracy and the overhead of the proposed system. First, we contract the overhead of "Chameleon" with "ScalaTrace". Second, to verify the accuracy of $online$ traces by replaying Chameleon traces and comparing its wall-clock time with that of ScalaTrace and the application's execution time (APP). All experiments are run five times, and the average is reported in this section. For these two experiments, the standard deviation is less than $1\%$ of the average values. The number of marker calls is based on Table II.

### A. Strong Scaling

Under strong scaling, the number of processes is increased under the same program input. We assessed our clustering algorithm on the NAS benchmarks and POP under strong scaling. Figure 4 depicts three bars: (1) the non-instrumented original application, (2) the execution overhead of Chameleon and (3) the execution overhead of ScalaTrace. The x/y-axes denote the number of processes and execution overhead in seconds, respectively, the latter shown on a logarithmic scale. The execution overhead of ScalaTrace features just regular inter-node compression performed in $MPI\_Finalize$. We observe that the overhead of Chameleon is less than 50% of total program execution time — in contrast to the original inter-node compression of ScalaTrace, which sometimes exceeds the application runtime for larger number of processes.

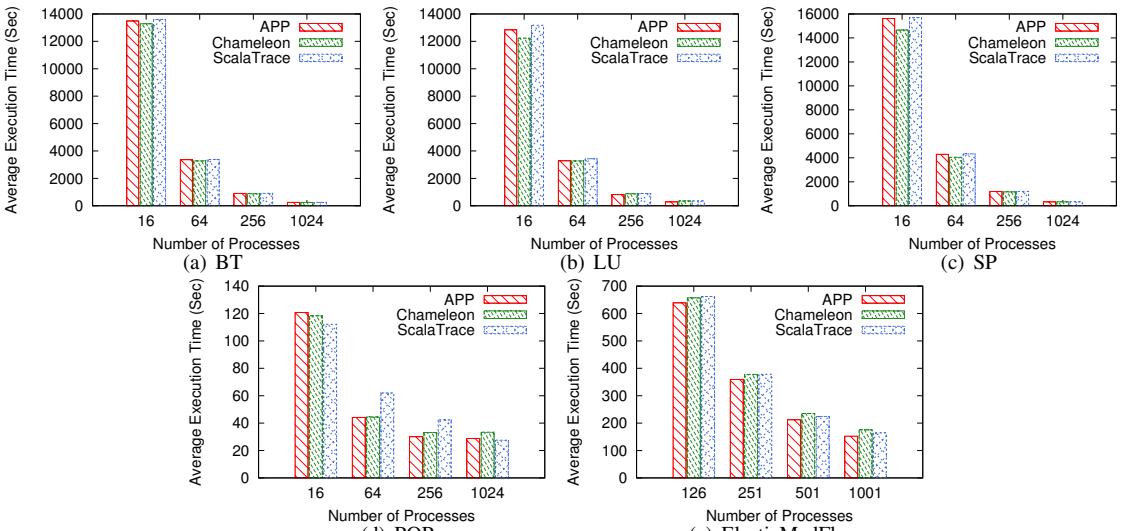
For EMF, intra-compression is extremely effective as it reduces all MPI events to just 6 PRSD events. For such small traces, ScalaTrace outperforms Chameleon for P $<$




(a) BT (b) LU (c) SP

(d) POP (e) ElasticMedFlow

Figure 4: Execution Overhead for NAS benchmarks - Strong Scaling - Nodes/Tasks=1/16




(a) BT (b) LU (c) SP

(d) POP (e) ElasticMedFlow

Figure 5: Replay Time of Traces - Strong Scaling - Nodes/Tasks=1/16

501 slightly, but for larger numbers of processes Chameleon beats ScalaTrace (e.g., Chameleon takes half the time for P=1000) due to higher communication cost at scale.

*Observation 2: Chameleon has 2-3 orders of magnitude lower overhead than ScalaTrace under strong scaling (except for extremely small traces).*

Irregular codes behave as follows: Irregular computations, e.g., sparse matrix vector multiplies (SpMV), do not affect communication and, hence, do not impact clustering (see results for NAS CG, which uses SpMV in CSR format, in [2]). POP experiences different data-dependent convergence points in timestep computation, i.e., the number of inner loop iterations varies across nodes. Hence, communication at this level occurs at irregular iteration points. Chameleon can replay POP with only 3 clusters (i.e., the number of Call-Paths). It utilizes the automatic filter from [2] for call parameters so that the communication patter becomes regular and can be represented by 3 clusters.

To assess the accuracy of the trace files created by the clustering algorithm, we utilized ScalaReplay, a replay engine operating on the application traces generated by ScalaTrace. It interprets the compressed application traces on-the-fly, issues MPI communication calls accordingly, and simulates computational overhead as sleeps [29]. We enhanced this replay capability so that the trace of a single node representing a cluster is also replayed by *all other nodes* in the same cluster. These other nodes re-interpret the single node trace and transpose any parameters relative to their task ID automatically because ScalaTrace utilizes relative encodings of end-points, while all other parameters are taken verbatim from the lead process of the cluster.

The accuracy of the replay time for traces is defined as

$$ACC = 1 - \frac{|t - t'|}{t}$$

where $t$ is the replay time without clustering and $t'$ is the replay time for clustered traces.

Figure 5 depicts the overall execution time, depicted in seconds on a linear y-axis for (1) the non-instrumented original application, (2) replay of a Chameleon trace and (3) replay of a ScalaTrace trace for the same x/y axes as before, but on a linear scale for the latter. Replay under Chameleon for BT, SP, LU, POP and EMF is 97.75%, 95.5%, 91%, 89.75% and 87% accurate, respectively, relative to the application runtime for all configurations, which also closely resembles ScalaTrace's behavior.

*Observation 3: Chameleon's clustered traces of lead processes represent application execution time as accurately as per-node traces with ScalaTrace under strong scaling.*

### B. Weak Scaling

Weak scaling typically involves scaling the problem size and the number of processors at the same rate such that the problem size per processor is fixed. (Weak scaling may sometimes also refer to scaling the number of nodes at the same rate as the memory footprint or computational complexity of some algorithm, which we consider as well in the following.). Figure 6 depicts execution overhead in seconds on a logarithmic scale (y-axis) of LU and Sweep3D for different numbers of processors (x-axis). Notice that other benchmarks lack weak scaling inputs.

*Observation 4: Chameleon's clustering results in 1-3 orders of magnitude shorter execution time than ScalaTrace under weak scaling.*

Figure 7 depicts the overall trace-file replay time in seconds on a linear scale (y-axis) for different numbers of processors (x-axis). Similar to strong scaling, this shows that the overall trace-file replay time under Chameleon for LU and Sweep3D is 90.75% and 98.32%, respectively, relative to application runtime over all configurations. Sweep3D exhibits load imbalance, but this irregularity does not affect clustering since delta times [27] are represented in histograms for repetitive signatures.

*Observation 5: Chameleon's clustered traces represent application execution time as accurately as ScalaTrace under weak scaling.*

### C. The impact of Online Clustering

To better assess the behavior of *online* clustering, we conducted four more experiments. First, we assessed the fraction of execution time per clustering state relative to overall tracing cost (i.e., inter-compression and marker calls). Figure 8 depicts the amount of time Chameleon spent in each state in seconds (linear y-axis) for different benchmarks (x-axis) for both Chameleon (CH) and ScalaTrace (ST). In this experiment, the number of marker calls is equal to the number of timesteps (e.g., 300 for LU, see Table II). The standard deviation over the same number of experiments as before is less than 6% of the reported average execution time.

For EMF with P=1000, the tuple costs (clustering, inter-compression) are (0.46%, 0.11%) for Chameleon and (0%, 0.53%) for ScalaTrace. We only report it in the text since it would not be visible on the scale of the figure. We observe that even for small compressed traces, inter-compression is reduced significantly in Chameleon compared to ScalaTrace under the maximum number of marker calls.

*Observation 6: The overhead of Chameleon under the maximum number of marker calls is an order of magnitude smaller than that of ScalaTrace.*

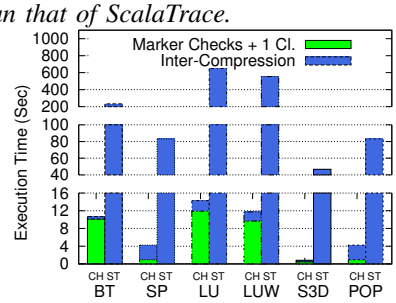
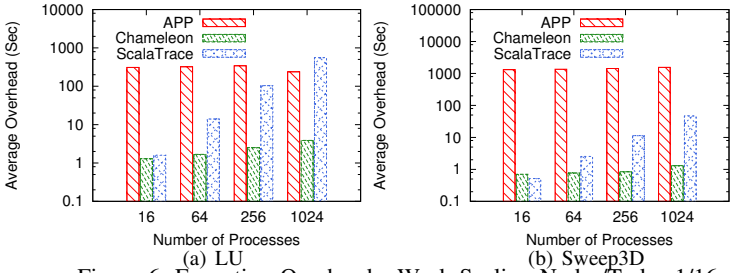



Figure 8: Overhead of Chameleon vs. ScalaTrace - Max. # of Marker Calls and P=1024

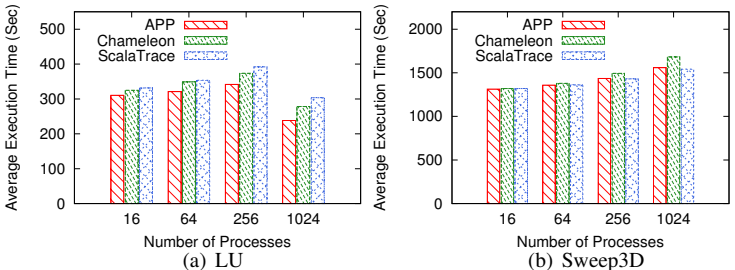
Next, we assessed the impact of phase changes on the overhead of Chameleon. The impact of marker calls on the overhead of Chameleon is depicted in Figure 9 with Chameleon's overhead (linear y-axis) for different numbers of marker calls (x-axis) for P=1024. The overhead maxes out at 300, where Chameleon creates signatures at each timestep. This overhead is still an order of magnitude less than ScalaTrace's.

According to the transition graph, if in every marker call, there is a different $Call - Path$, then there would be no clustering, and Chameleon stays in state "AT". To maximize the number of re-clusterings, the we can force state between "AT" and "C", i.e., such that at every other marker call a phase change is simulated. To observe the overhead of re-clustering, we modified LU such that for every 10 timesteps,




(a) LU (b) Sweep3D

Figure 6: Execution Overhead - Weak Scaling Nodes/Tasks=1/16




(a) LU (b) Sweep3D

Figure 7: Replay Time of Traces - Weak Scaling - Nodes/Tasks=1/16

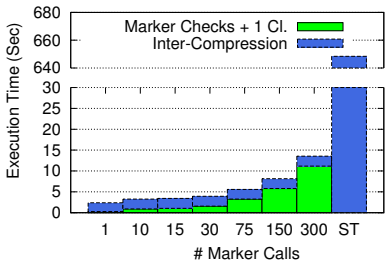



Figure 9: Overhead vs. # Clustering Calls: LU Class D, P=1024

processes call a new $MPI\_Barrier$. This indicates a new $Call - Path$ and changes the program phase. Figure 10 depicts the overhead of re-clustering (linear y-axis) under varying numbers of re-clusterings (x-axis) for P=1024. The second solid bar represents ScalaTrace's overhead. For LU class D with 300 timesteps, the maximum number of re-clusterings is 30 here (every tenth timestep). We increase the number of re-clusterings gradually from 1 (i.e., the original application) to 30. The standard deviation is less than $10\%$ of the average execution time. For the modified LU with 30 re-clusterings and phase changes, the overhead of Chameleon still is an order of magnitude less than ScalaTrace.

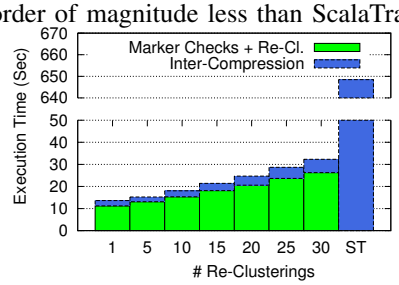



Figure 10: Re-Clustering Cost: LU Class D, 300 Markers, P=1024

*Observation 7: In contrast to ScalaTrace, phase changes are transparently captured by Chameleon while retaining an order of magnitude lower overhead.*

The slight increase in the inter-compression overhead for Chameleon is due to the new MPI events added to the original sequence. New events create a new $Call - Path$, which prevent perfect matches and increase the size of the trace, but only insignificantly in size.

We next assessed the impact of problem sizes on the overhead of each state. Figure 11 depicts the overhead of each state (linear (y-axis) for different problem size (input classes A/B/C/D) and numbers of timesteps (x-axis) for both Chameleon (CH) and ScalaTrace (ST) with P=256. We chose P=256 because the problem size of class A is too small for a larger P. The standard deviation is less than $8\%$ of the reported average time. the overhead of Chameleon increases with the number of timesteps, where every timestep results in a marker call. However, this overhead is still an order of magnitude smaller than ScalaTrace's across problem sizes.

*Observation 8: Chameleon retains an order of magnitude lower overheads irrespective of input problem sizes.*

ACURDION clusters traces dynamically online. In contrast, Chameleon clusters traces only at MPI_Finalize. Table III shows the execution overhead under ACURDION

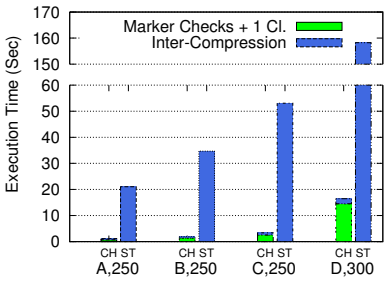



Figure 11: Overhead per Method vs. Input Sizes, LU, P=256

vs. Chameleon. This experiment constrains the maximum number of calls for Chameleon (to 250 calls for BT Class=D). The results show that the execution overhead of Chameleon is almost twice that of ACURDION even under this maximum threshold for the number of calls.

Table III: Overhead[secs]: BT Class D

| Pgm (P) | 16 | 64 | 256 | 1024 |
|---|---|---|---|---|
| ACURDION | 0.08 | 0.14 | 0.32 | 0.74 |
| Chameleon | 0.13 | 0.33 | 0.57 | 1.76 |

### D. Space Complexity

Our algorithm follows the same space complexity as the $Call - Path + Parameter$ clustering [1] at the inter-node compression step. It reduces the space by 2-3 orders of magnitudes on average compared to ScalaTrace. This is because the number of processes involved in inter-node compression is limited to a small (constant) number, $K$, while without clustering imposes $P$ participants.

*Observation 9: Chameleon reduces intra-node compression space in contrast to related work, namely $Call-Path+Parameter$ clustering.*

This benefit is due to the fact that non-lead processes do not need to be traced in the "L" (Lead) state. Therefore, non-lead processes do not need to generate traces in first place, they simply rely on their cluster lead processes to do so.

Table IV indicates the allocated space of traces under different states for BT Class D with P=256. Processes 0, 1, and 15 are lead processes. For the remaining 253 non-lead processes, *no extra space* (0 bytes) is required for 197 calls as the follow the leader in their cluster. The average space allocation for all processes under ACURDION as well as non-clustering is $77,687$ bytes, i.e., in ACURDION, all processes need to allocate memory for their traces because clustering only occurs late at MPI_Finalize. This shows that only process 0 required more space (a $49\%$ increase for the global online trace), processes 1 and 15 required about half the space (a decrease of over $54\%$), and all 253 non-leads have significantly reduced space ($\%99$ smaller than before).

## VI. Related Work

Bahmani and Mueller [1], [3], [2] proposed signature-based clustering algorithms for ScalaTraceV2. Chameleon significantly improves over this prior work by developing a context-aware clustering framework that supports *online* inter-node compression.

Table IV: Memory Allocation for Traces in Bytes for BT Class D with P=256 - Three Lead and 253 Non-Lead Processes

| BT(Class D) | # Calls | 0∗ | 1 | 15 | Non-Lead |
|---|---|---|---|---|---|
| All Tracing (AT) | 2 | 96,140 | 96,140 | 96,140 | 96,140 |
| Clustering(C) | 1 | 144,244 | 43,336 | 43,336 | 43,336 |
| Lead (L) | 247 | 144,364 | 43,456 | 43,456 | 0 |
| Finalize (F) | 1 | 204,572 | 103,664 | 103,664 | 0 |
| Avg. Per Call | 251 | 115,461 | 35,459 | 35,459 | 939 |

∗ Node 0 allocates space for own trace + global online trace.

CYPRESS [34] is a communication trace compression framework that combines static program analysis with dynamic runtime trace compression. It extracts the program structure at compile time to identify critical loop/branch control structures. They compared their result with ScalaTraceV2. One of the main problems of CYPRESS is the overhead of static and dynamic analysis while the combination of Chameleon does not have any overhead during application compile time.

A density-based clustering analysis, proposed by Gonzalez et al. [20], [12], [11], can use an arbitrary number of performance metrics to characterize the application (e.g., "instructions" combined with "cache misses" to reflect the impact of memory access patterns on performance). Using $K$-means clustering to select representative data for migration of objects in $CHARM++$ is an approach utilized by Lee et al. [18] and [19]. Their clustering algorithms are expensive in terms of time complexity, especially for large-scale sizes. On the other hand, our work contributes a low overhead clustering algorithm with $O(logP)$ complexity.

Phantom [33], a performance prediction framework, uses deterministic replay techniques to execute any process of a parallel application on a single node of the target system. To reduce the measurement time, Phantom leverages a hierarchical clustering algorithm to cluster processes based on the degree of computational similarity. First, the computational complexity for most hierarchical clustering algorithms is at least quadratic in time, and this high cost limits their application in large-scale data sets [32]. Second, because the paper focuses on performance prediction, it emphasizes computational similarity and does not sufficiently cover communication behavior.

HPCTOOLKIT [24] utilized statistical sampling to measure performance. HPCTOOLKIT provides and visualizes per process traces of sampled call paths. A parallel clustering algorithm based on CLARA [15] was proposed in CAPEK [9] that enables in-situ analysis of performance data at runtime. Even though the algorithm is logarithmic, the process of clustering and creating the global trace file is based on trace sampling. Sampling cannot produce accurate data but rather represents a statistical and lossy method. For instance, if the sampling frequency is too low, results may not be representative. Conversely, if it is too high, measurement overhead can significantly perturb the application. In HPCTOOLKIT and CAPEK, the process of finding an appropriate rate of sampling is also complicated.

In contrast to the above existing frameworks, including its predecessors, Chameleon provides a full trace file without resorting to sampling, and it does so at very low cost by leveraging 64-bit stack signatures. Chameleon reduces space allocation drastically for all $P - 9$ non-lead processes by creating an online trace using only 9 lead processes and via turning off tracing for non-lead processes.

## VII. Discussion

The proposed system has the following weaknesses that we will consider in our future work, but are mostly engineering:

(1) Marker insertion requires source code a modification. For applications where the source code is not available, one could insert the marker into binary files using binary instrumentation, e.g., via Pin [21].

(2) Finding of a good location for inserting marker and choosing an appropriate frequency call are open problems. Even though the execution overhead under the maximum number of marker calls is at least an order of magnitude smaller compared to ScalaTrace, Chameleon puts the burden of adding the marker and choosing the number of marker calls on programmer. This could be automated in some cases.

For iterative scientific applications (most scientific codes), the main loop gets executed by all processes (and marker insertion can be automated), i.e., processes might diverge but only at finer granularity. For execution via asynchronous tasks, a task-based method would be required and could also be automated at that scope. Of course, today, few applications exist that use asynchronous tasks in HPC.

## VIII. Conclusion and Future Work

Scalability is one of the main challenges of scientific applications in HPC. This paper contributes an online clustering algorithm with $logP$ time complexity and low overhead. The approach relies on grouping together processes with the same execution behavior at interim execution points, e.g., at timestep boundaries of scientific codes. ScalaTraceV2's inter-node compression is performed online. The results of our experiments indicate that our clustering algorithm provides significant reductions in performance over ScalaTraceV2 making it suitable for extreme-scale computing. Our clustering algorithm is applicable to both strong and weak scaling applications. We currently plan to leverage the idle time for non representative processes at interim execution points by utilizing dynamic voltage frequency scaling (DVFS). This would reduce energy consumption and make clustered tracing energy efficient as well.

## Acknowledgment

This research was supported by the National Science Foundation, award numbers 1217748, 1525609.

## REFERENCES


[1] Amir Bahmani and Frank Mueller. Scalable performance analysis of exascale mpi programs through signature-based clustering algorithms. In *International Conference on Supercomputing*, pages 155–164. ACM, 2014.

[2] Amir Bahmani and Frank Mueller. Efficient clustering for ultra-scale application tracing. *Journal of Parallel and Distributed Computing*, 98:25–39, 2016.

[3] Amir Bahmani and Frank Mueller. Scalable communication event tracing via clustering. *Journal of Parallel and Distributed Computing*, 109:230–244, 2017.

[4] Bahmani, Amir and Baucom, Sarah and Khan, Monis and Koprly, Hussein and Moore, Spencer and Mueller, Frank and Niethammer, Marc and Owzar, Kouros and Parsian, Mahmoud and Sibley, Alexander and Styner,Martin. Elasticmedflow, 2016.

[5] David H Bailey, Eric Barszcz, Leonardo Dagum, and Horst D Simon. Nas parallel benchmark results. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):43–51, 1993.

[6] Daniel Becker, Felix Wolf, Wolfgang Frings, Markus Geimer, Brian J. N. Wylie, and Bernd Mohr. Automatic trace-based performance analysis of metacomputing applications. In *International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

[7] Holger Brunst, Manuela Winkler, Wolfgang E Nagel, and Hans-Christian Hoppe. Performance optimization for large scale computing: The scalable vampir approach. In *Computational Science-ICCS 2001*, pages 751–760. Springer, 2001.

[8] Lisandro Dalcin. mpi4py, 2007.

[9] Todd Gamblin, Bronis R De Supinski, Martin Schulz, Rob Fowler, and Daniel A Reed. Clustering performance data efficiently at massive scales. In *International Conferernce on Supercomputing*, pages 243–252. ACM, 2010.

[10] M. Geimer, F. Wolf, B. J. N. Wylie, E. Abraham, D. Becker, and B. Mohr. The scalasca performance toolset architecture. In *International Workshop on Scalable Tools for High-End Computing*, June 2008.

[11] Juan Gonzalez, Judit Gimenez, and Jesus Labarta. Automatic detection of parallel applications computation phases. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2009.*, pages 1–11. IEEE, 2009.

[12] Juan Gonzalez, Kevin Huck, Judit Gimenez, and Jesus Labarta. Automatic refinement of parallel applications structure detection. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1680–1687. IEEE, 2012.

[13] Intel. Intel trace analyzer and collector, 2015. https://software.intel.com/en-us/intel-trace-analyzer.

[14] Philip W Jones, Patrick H Worley, Yoshikatsu Yoshida, JB White, and John Levesque. Practical performance portability in the parallel ocean program (pop). *Concurrency and Computation: Practice and Experience*, 17(10):1317–1327, 2005.

[15] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. Wiley.com, 2009.

[16] Andreas Knupfer. Construction and compression of complete call graphs for post-mortem program trace analysis. In *International Conference on Parallel Processing*, pages 165–172, 2005.

[17] Kenneth R Koch, Randal S Baker, and Raymond E Alcouffe. Solution of the first-order form of the 3-d discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65(108):198–199, 1992.

[18] Chee Wai Lee and Laxmikant V Kalé. Scalable techniques for performance analysis. *Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana-Champaign, Tech. Rep*, pages 07–06, 2007.

[19] Chee Wai Lee, Celso Mendes, and Laxmikant V Kalé. Towards scalable performance analysis and visualization through data reduction. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS), 2008.*, pages 1–8. IEEE, 2008.

[20] German Llort, Juan Gonzalez, Harald Servat, Judit Gimenez, and Jesús Labarta. On-line detection of large-scale parallel application's structure. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010*, pages 1–10. IEEE, 2010.

[21] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[22] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R de Supinski. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710, 2009.

[23] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[24] Nathan R Tallent, John Mellor-Crummey, Michael Franco, Reed Landrum, and Laksono Adhianto. Scalable fine-grained call path tracing. In *International Conferernce on Supercomputing*, pages 63–74. ACM, 2011.

[25] J. Vetter and M. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001.

[26] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Journal of Parallel Distributed Computing*, 63(9):853–865, September 2003.

[27] X. Wu, K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. Probabilistic communication and i/o tracing with deterministic replay at scale. In *International Conference on Parallel Processing*, pages 196–205, September 2011.

[28] Xing Wu and Frank Mueller. Scalaextrap: Trace-based communication extrapolation for spmd programs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 113–122. ACM, 2011.

[29] Xing Wu and Frank Mueller. Scalaextrap: Trace-based communication extrapolation for spmd programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(1):5, 2012.

[30] Xing Wu and Frank Mueller. Elastic and scalable tracing and accurate replay of non-deterministic events. In *Conference on International Conference on Supercomputing*, ICS '13, pages 59–68. ACM, 2013.

[31] Xing Wu, Frank Mueller, and Scott Pakin. Automatic generation of executable communication specifications from parallel applications. In *Proceedings of the international conference on Supercomputing*, pages 12–21. ACM, 2011.

[32] Rui Xu, Donald Wunsch, et al. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[33] Jidong Zhai, Wenguang Chen, and Weimin Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. *ACM Sigplan Notices*, pages 305–314, 2010.

[34] Jidong Zhai, Jianfei Hu, Xiongchao Tang, Xiaosong Ma, and Wenguang Chen. Cypress: Combining static and dynamic analysis for top-down communication trace compression. In *Supercomputing*. To appear, 2014.

[35] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.