Scalable Communication Event Tracing via Clustering [⋆]

Amir Bahmani [a] Frank Mueller [a,∗]

[a] *North Carolina State University Department of Computer Science Raleigh, NC 27695-7534*

Abstract

Communication traces help developers of high-performance computing (HPC) applications understand and improve their codes. When run on large-scale HPC facilities, the scalability of tracing tools becomes a challenge. To address this problem, traces can be clustered into groups of processes that exhibit similar behavior. Instead of collecting trace information of each individual node, it then suffices to collect a trace of a small set of representative nodes, namely one per cluster. However, clustering algorithms themselves need to have low overhead, be scalable, and adapt to application characteristics. We devised an adaptive clustering algorithm for large-scale applications called ACURDION that traces the MPI communication of code with O(log P) time complexity. First, ACURDION identifies the parameters that differ across processes by using a logarithmic algorithm called Adaptive Signature Building. Second, it clusters the processes based on those parameters. Experiments show that collecting traces of just nine nodes/clusters suffices to capture the communication behavior of all nodes for a wide set of HPC benchmarks codes while retaining sufficient accuracy of trace events and parameters. In summary, ACURDION improves trace scalability and automation over prior approaches.

Key words: Clustering Algorithms, Programming Techniques, Concurrent Programming, Performance Measurement
PACS: 07.05.Bx

[⋆] An earlier version of this paper appeared at IEEE BigData'15. This journal version extends the earlier paper by implementing K-Medoid during intra-clustering in addition to K-Farthest and compares the performance of the two clustering approaches. Furthermore, a new clustering method, dynamic clustering, has been contributed. Paper extensions further include heatmap comparisons for 6 benchmarks, and execution overhead plus replay time graphs for strong and weak scaling besides discussions of additional related work. This work was supported in part by NSF grants CCF-1217748 and CNS-0958311.
[∗] Corresponding author.
Email addresses: `abahman@ncsu.edu` (Amir Bahmani), `mueller@cs.ncsu.edu` (Frank Mueller).

1 Introduction

The increasing size of HPC systems often requires applications to be carefully designed for scalability. One of the key challenges is to utilize communication efficiently. Programmers generally understand the semantics of MPI communication routines, but they may not know trade-offs and limitations in a concrete implementation of MPI, particularly when a problem surfaces only at a larger scale but not in constrained testing with small inputs. In such scenarios, communication traces often provide the insight to detect inefficiencies and help in problem tuning [29,3]. Traces are also utilized to drive HPC simulations to determine the effect of interconnect changes for future procurements [24,33,20,26,34].

Today's tracing tools either obtain lossless trace information at the price of limited scalability (e.g., Vampir [5], Tau [23], Intel's toolsets [11], and Scalasca [8]) or preserve only aggregated statistical trace information to conserve the size of trace files (e.g., mpiP [28]). 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 [36]. In response, a number of communication compression techniques have been designed, including run-length compression [38] and structural compression [14,21,31].

Any of these trace compression techniques, except for run-length compression, provide the means to analyze and replay traces without decompression. But the original trace is still obtained across all nodes and cores of an HPC application. In general, the scalability of tracing tools becomes a challenge when an application is run on large-scale HPC facilities. For instance, ScalaTraceV2 [31] first performs intra-node trace compression and later (at program termination) consolidates compressed traces from each node into a single representation (during inter-node compression). This latter step reportedly has limited scalability, a problem that has been addressed in two ways. (1) CYPRESS [36] uses a hybrid static/dynamic compiler-aided compression technique [36] with speedups of 10x for inter-node compression, but traces are still obtained across all nodes, which constrains speedups to a constant factor. (2) Traces can be clustered into groups of processes that exhibit similar behavior [1]. Once clustered, traces no longer need to be collected per node but just for a single node per cluster, which means clustering overhead is incurred for a small number of nodes irrespective of the total number of nodes. In fact, this node number is a small constant for all practical purposes [1]. The behavior of this single node representing a cluster is later interpreted as the behavior of all nodes in the clusters, but in a manner that preserves unique properties of each node. For example, message endpoints are still interpreted relative to a subject node and not simply copied from the representative node of a cluster. In practice, the number of clusters needed tends to be small (up to nine in our experiments) without sacrificing accuracy.

Bahmani and Mueller [1] employed a hierarchical, *signature-based* clustering algorithm using two 64-bit signatures. The first level of clustering is call-path clustering where processes with different numbers or sequences of events are discovered. During the second phase, this algorithm applies parameter clustering using a 64-bit signature. This signature is composed of the parameters of an MPI event, such as its count, type, source, destination, etc.

Even though this previous work is suitable for large-scale applications, it has several limitations. The size of the signature is the first problem. The algorithm is very space efficient due to the 64-bit signature for all of the MPI parameters, but compressing the parameters could result in loss of information. For instance, 16 bits for message sizes (*COUNT*) is not enough. On the other hand, expanding signatures blindly to a larger scale could increase space complexity significantly and cause scalability problems.

The second limitation is observed for benchmarks such as CG [2]. The unique communication behavior of the processes causes the number of parameter clusters to increase linearly with the number of processes, which is not scalable. To tackle the problem, user input plug-in functions were utilized to specify the communication pattern. The problem with the user plug-in functions is that finding user plug-ins for complex benchmarks is not easy.

The third problem is a scalability challenge due to hierarchical clustering, which already improves on ScalaTrace. 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. While intra-compression is fast and efficient, inter-compression is costly as it depends on the number of tasks. Hierarchical clustering lowered this overhead so that it depends on the number of clusters, but some application codes still require a number of clusters linear to the number of tasks in order to retain trace accuracy.

This paper describes ACURDION, a scalable clustering algorithm for large scale applications. Figure 1 depicts the schematic of ACURDION with respect to ScalaTrace.

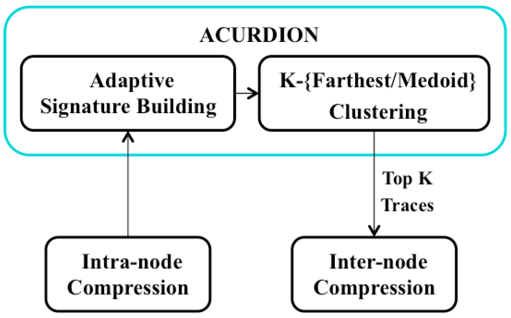


Fig. 1. A Schematic of ScalaTrace with ACURDION

ACURDION first finds parameter differences between processes through Adaptive

Signature Building, a $O(logP)$ algorithm. Then, ACURDION applies either the K-Farthest or the K-Medoid algorithms on the selected signatures to group processes into K clusters. This lowers the complexity of inter-node compression to just be dependent on K, which is a constant in practice ($K = 9$ for the programs studied). Consequently, user plug-ins are no longer needed for complex communication patterns, i.e., ACURDION advances trace automation as well.

Contributions:
• We introduce ACURDION, a $O(logP)$ clustering algorithm that supports both K-Farthest and K-Medoid clustering with low time and space overheads.
• We describe novel signature finding algorithms that help prune unnecessary metrics and only consider parameters covering differences among the traces.
• We evaluate ACURDION for a set of HPC benchmarks showing its effectiveness at capturing representative application behavior for communication events. The resulting number of clusters is a constant for all benchmarks.
• We compare the accuracy of traces generated by ACURDION with prior work on signature-based trace clustering.
• We introduce "dynamic clustering", and we compare the clustering overhead and accuracy of traces generated of ACURDION and dynamic clustering.

2 Background

In this section, we briefly introduce several of the key ideas and techniques relevant to MPI tracing because our work builds on ScalaTrace as an MPI tracing toolset [31].

ScalaTrace uses the following two main data structures: Regular Section Descriptors (RSD) that capture MPI events in the innermost loop, and Power-RSDs (PRSDs) that capture RSDs from higher-level loop nests. Consider the example in the following code segment:

```
for $i = 0 \rightarrow 1000$ do
  for $k = 0 \rightarrow 100$ do
    $MPI\_Irecv(...)$;
    $MPI\_Send(...)$;
    $MPI\_Wait(...)$;
  end for
  $MPI\_Barrier(...)$
end for
```

RSDs and PRSD of the above code segment are RSD1:<100, MPI_Irecv1, MPI_Send1, MPI_Wait>, and PRSD1:<1000, RSD1, MPI_Barrier1>, respectively.

The main three properties of ScalaTrace are as follows: (1) Location-independent encodings: ScalaTrace leverages relative encodings of communication end-points, i.e., an end-point is denoted as ±c for a constant c relative to the current MPI task ID [21]. Fig. 2 depicts the relative encoding of nodes 7 and 10 in terms of communication end-points, namely -4, -1, $+1$ and $+4$, i.e., these nodes have identical *relative* communication end-points.

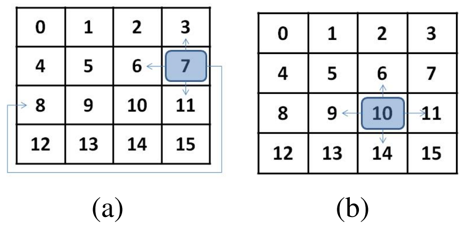


Fig. 2. Communication End-point Encoding

(2) Call sequence identification: To distinguish between MPI calls from different locations, ScalaTrace captures the calling context by recording the calling sequence that leads to the MPI event. The calling context is obtained from the stack backtrace of an MPI event where each location is a unique signature of the stack trace called the stack signature [21].

(3) Communication group encoding: To store communication groups in a scalable way in traces, ScalaTrace leverages a special data structure called ranklist. A ranklist is represented as ⟨dimension, start_rank, iteration_length, stride, iteration_length, stride⟩, where dimension is the dimension of the group, start rank is the rank of the starting node, and the iteration length stride pair is the iteration and stride of the corresponding dimension [30].

A ranklist covers a group of processes with any number of dimensions. For instance, in Fig. 3(a), the shaded nodes are presented as ranklist ⟨*2 0 4 4 4 1*⟩, and in Fig. 3(b), they are presented as ranklist ⟨*2 0 4 4 2 1*⟩. The second ranklist reads as a 2D ranklist starting at task 0 with four entries in the first dimension and a stride of 4 (implying tasks 0, 4, 8 and 12) with two entries in the second dimension with a stride of 1 (implying tasks 1, 5, 9 and 13). The ACURDION algorithm introduced in the next section is working on top of the intra-node compression step.

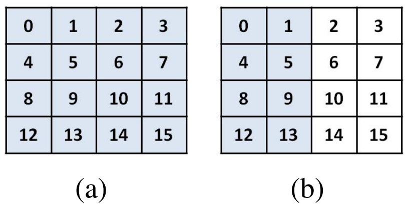


Fig. 3. Ranklist of a Communication Group

3 A Novel Clustering Algorithm

ACURDION has two main phases. First, the adaptive signature building phase discovers signatures suitable for the clustering. Second, the clustering phase clusters traces based on their suitable signatures. This phase involves a single-step (where all signatures are considered in clustering), or a double-step where during the first step it applies clustering on the *Call-Path* signature and then in the second step it applies clustering on all other signatures within the *Call-Path* clusters. This section details design and implementation of the single- and double-step algorithms of ACURDION and the reference clustering algorithm.

3.1 Signature Building

In order to generate global traces with high accuracy, we found (based on our prior work [1]) that the clustering algorithm needs to consider the calling context, and several important MPI parameters. Therefore, the clustering algorithms by default consider eleven signatures. The first one, the *Call-Path* signature, helps to cluster processes with similar sequences of MPI calls.

As noted previously, to represent 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. The *Call-Path* signature, a 64-bit signature, is the XOR of all 64-bit stack signatures. In order to create the *Call-Path* signature, capturing the calling context is sufficient for distinguishing MPI events from each other in most benchmarks. Moreover, to order events, we multiply the modulo 10 plus 1 of the sequence number of each event by the 64-bit stack signature and then use this value in the *Call-Path* signature. This ensures that signatures cannot cancel out each other due to permutations on call sequences and recursion, which could otherwise happen in rare cases (e.g., NAS MG code).

All other signatures are averaged parameter signatures composing parameters of the MPI call event (*COUNT*, *SRC*, *DEST*, *KEY*, *COLOR*, *TAG*, *Computation time*, *Communication time*, *LOOP iteration* and *Data+Operation+Communicator type*). *KEY* and *COLOR* are arguments of MPI_COMM_SPLIT, which splits an existing communicator into multiple communicators using these arguments. For the first eight above-mentioned signatures, aggregating their values and then taking the average could result in an overflow. Thus, we calculate the average incrementally.

Here, *input* is a vector of data, and the algorithm is called when the size of the input is larger than two. For the *Loop* signature, considering the importance of nested loops, we know that multiplying the bounds of nested loops could cause overflow as well. To avoid this, we divide 64 bits into four sections. The least significant 16 bits are assigned to the average of the inner most loop sizes (or the least important loop), then the second 16 bits are assigned to loop above the first section, etc. Anything above three levels is considered part of the fourth section (most significant bits). As a *Data+Operation+Communicator Type* signature, we assign a bit such as 0:MPI CHAR, 32:MPI MAX, and 55:MPI COMM SELF, etc., per MPI data type, MPI operation type, and MPI communicator types. After creating signatures at the node level, the first step of ACURDION is to enter an adaptive signature building phase. Algorithm 1 presents the pseudocode of this phase.

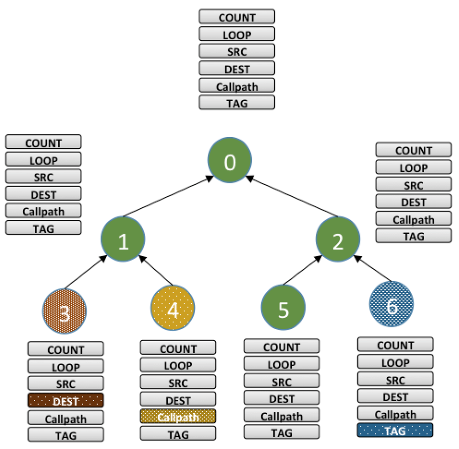
Algorithm 1: Adaptive Signature Building

```
1  Set your signature format to 0;
2  if a left/right child exists then
3      Receive its signatures;
4      if your signature $\neq$ child signature then
5          update the signature format;
6      Receive the child's signature format;
7      signature format = signature format OR child's signature format;
8  if a parent exists then
9      Send your signatures to your parent;
10     Send your signature format to your parent;
11 Broadcast signature format by rank root;
```

The time complexity of the algorithm is $log(P)$. Fig. 4 and Fig. 5 show an example of signature building. First, each process creates its signatures. Then they send their signatures and signature format to their parents in a bottom-up procedure over a radix tree.

This procedure uses a set in which there is a bit (per signature) indicating whether or not the signature should be stored. In this example, due to the space limitation, we only consider 6 signatures. The signature format consists of 6 bits, where the right-most bit represents the TAG signature, and the left-most bit represents the $COUNT$ signature. At the beginning, all bits are zero. When two different clusters encounter each other (i.e., child's and parent's signatures differ), the bits corresponding to the different signatures are set to 1. As shown, leaf nodes (such as 5 and 6) send their signatures to the parent node 2. Then, node 2 compares its own signatures with the received ones. If it finds any difference, it flips the corresponding bit in the signature format to 1. Here, nodes 6 and 2 differ (captured at node 2).

This process continues up to the root of the tree. The root then broadcasts the bitmap to all nodes. At the end of this stage, all nodes know which signatures are subjected to clustering. In this example, the format signature is 000111 (i.e., TAG, $Call-Path$, and $DEST$ signatures are different), so the clustering algorithm only considers three signatures.



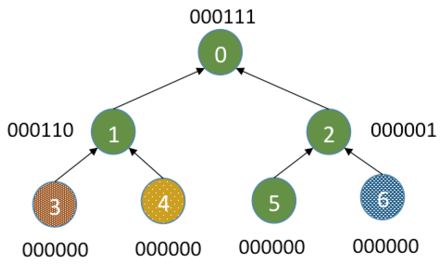


Fig. 5. A Sample of Building Signature Format $O(log(P))$

Fig. 4. A Sample of Creating Signatures $O(1)$

3.2 Single-Step and Double-Step Clustering

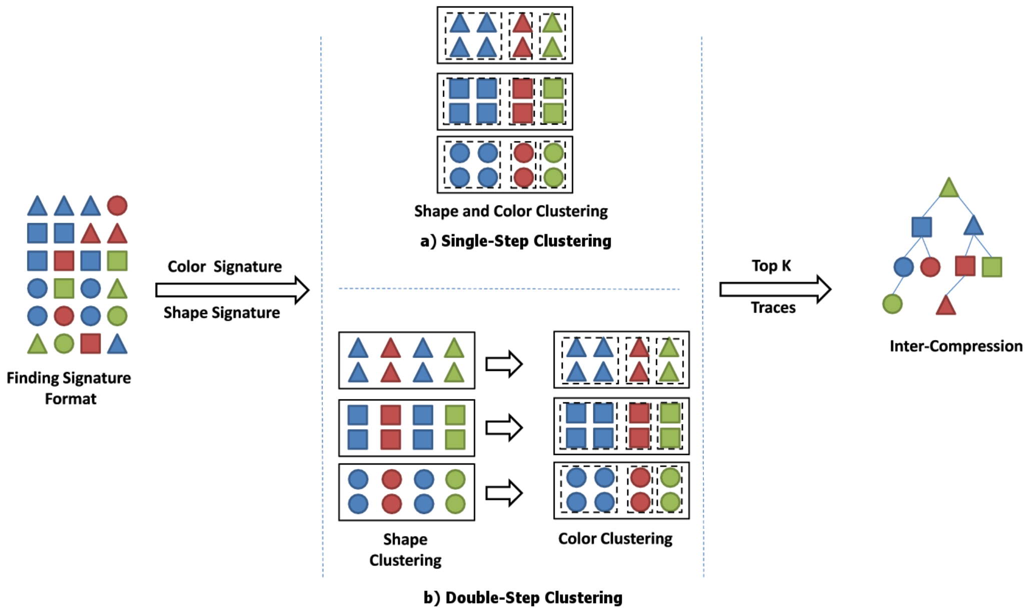


Fig. 6. Overview of Proposed Clustering Algorithm

Fig. 6 depicts a simplified illustration of how ACURDION works. In this figure, we assumed there are only two hypothetical signatures, $shape$ and $color$. ACURDION first finds these two signatures using the adaptive signature building. It then either follows single-step clustering (all signatures are considered in clustering) or the double-step clustering (first, only *Call-Path* or shape in this figure, and second, all other signatures are considered for the clusters created in the first step). In this figure, $color$ is the second dimension. After finding the top K clusters and selecting the top K representatives (nodes) to create the global trace file, inter-compression on the selected traces is applied.

In our design and implementation, we considered both Euclidean and Manhattan distances. Before calculating the distances, ACURDION dynamically normalizes them, i.e., it groups signatures based on their importance. The importance ordering is as follows: Group1={LOOP, COUNT, COMM Time and COMP Time}, Group2={SRC, DEST, KEY, COLOR, TAG} and Group3={Call-Path, Data+Operation+Communicator Type}. Compromising on group 1 does not perturb the application time significantly, as experiments show. Group 3 is the most important group because we do not want to lose any events by compromising on *Call-Path*. We further observed that *Call-Path* also covers the $Data+Operation+Communicator$ signature in practice.

Since $COUNT$ is a 32-bit integer, its average is also 32-bits. In our implementation, we assumed there is a boundary on the maximum value of group 2 which is also 32. With these assumptions, we can shift group 2 based on group 1, and then group 3 based on the smaller groups in such a way that the value of the larger group becomes larger than the smaller ones.

We tested ACURDION on all benchmarks with both the Manhattan and Euclidean distance functions. We then calculated the distance between the output of clustering to the non-clustering version. According to our experiments, both metrics are giving close distances, so we chose the Manhattan distance for our experiments.

Algorithm 2: ACURDION K-Farthest clustering algorithm

```
1  if a left/right child exists then
2      Receive list of left_K / right_K clusters;
3      Receive signature of head of top left_K / right_K clusters;
4      Merge left_K / right_K clusters + yourself into $AllNode$ list;
5      if left_K + right_K + 1 > K then
6          Calculate the distance matrix for Top $K$ list;
7          $TopK$ list = { } ;
8          while Size of $TopK$ list < K do
9              Find the farthest cluster to the $TopK$ list;
10         foreach cluster $\in$ $AllNode$ list - $TopK$ list do
11             Find the closest cluster;
12             Assign the cluster to the closest one;
13 if a parent exists then
14     Send your list of $K$ clusters to your parent;
15     Send signature of head of top $K$ clusters to your parent;
16 Broadcast Top $K$ by root;
```

Algorithm 2 depicts ACURDION's K-Farthest clustering algorithm. Fig. 7 shows how the proposed ACURDION K-Farthest algorithm operates over a radix tree. At each node, if it has a child, it first receives the K selected clusters and their signatures. Each node at most receives $2K$ clusters. After receiving K clusters from right child and K clusters from the left child, it then adds itself (i.e., its own cluster) to the list of all nodes $2K + 1$, and determines the top K clusters.

First, it calculates the distance matrix between all potential clusters based on the signature format. Second, it selects the top K clusters farthest from all $2K + 1$ clusters. Third, it distributes the rest of the clusters (which have not been selected) to their nearest cluster. At the end, after finding the top K clusters, if the current node has a parent, it will send the top K clusters and their signatures to its parent. This procedure is similar for K-Medoid clustering. The only difference is instead of finding the top K farthest clusters, we implemented the Partitioning Around Medoids (PAM) algorithm that randomly selects K medoids and iterates as long as the cost decreases until a fixed point is reached [22].

In single-step ACURDION, clustering happens over the entire signature format. For example, if $COLOR$ and *Call-Path* are parts of a signature format then the algorithm calculates the normalized distances from the signatures. The double-step version, in contrast, first clusters over *Call-Path* and then (at the second level) within the clusters created at the first level over other dimensions such as $COLOR$.

Note that the computational cost of our clustering algorithm is $O(logP)$, where P is the number of processes. K can be any constant value. In our experiments, $K = 9$ was shown to preserve sufficient accuracy (discussed in Section 5).

By the end of this stage, the algorithm has clustered all processes with disjoint behavior. Then, the algorithm creates the complete trace based on the cluster information.

3.3 Aggregating the Traces

As mentioned in the introduction, ScalaTraceV2 first performs intra-node trace compression and later (at program termination) consolidates compressed traces from each node into a single representation (during inter-node compression). The time complexity of inter-node compression of ScalaTrace is O(n^2) , where n is the size of the PRSD-compressed intra-node event trace. Since for ScalaTrace without clustering, all processes are participating in this operation over a radix tree, the time complexity is $O(n^2 logP)$. On the other hand, for ScalaTrace with the clustering algorithm, only a set of representative K nodes with different signatures are participating in this operation. During the last phase of Fig. 6, K different nodes are merged.

The cost of inter-compression with clustering is $O(n^2 logK)$, where $K = 9$ in our experiments and the cost of the clustering algorithm is $O(logP)$.

Before merging, the full trace is created from the clustered trace by updating the the trace files of the K selected representatives considering all members of the cluster. As an example, consider Fig. 7. P_{13} represents a group of processes: { P_1, P_2, P_8, P_9, P_{10}, P_{13} }. Therefore, P_{13} reflects the participation of all members of the cluster in its own trace file. This operation is linear, i.e., representatives linearly traverse their trace and replace a cluster ranklist (representing all members' IDs) with an event ranklist (compatible with the original ScalaTraceV2 format).

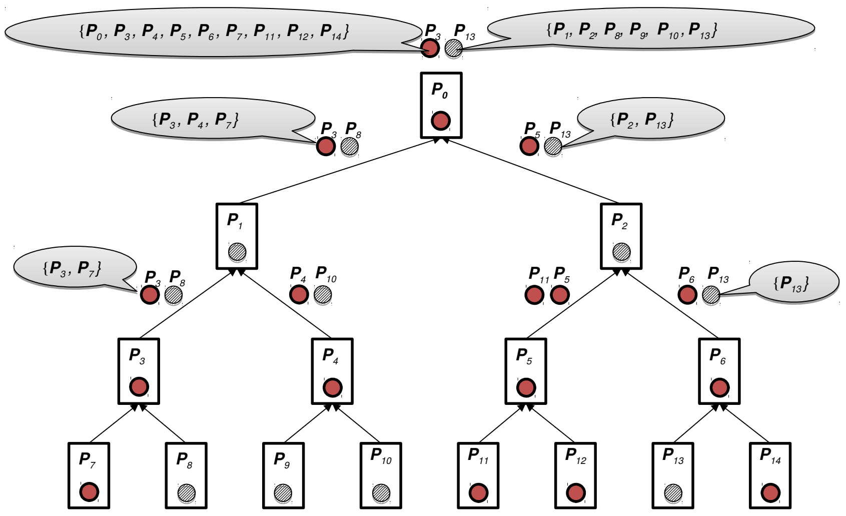


Fig. 7. An illustration of K-Farthest or K-Medoid Clusterings ($K = 2$)

3.4 Reference Signature

We use a reference clustering approach to evaluate the accuracy and scalability of our algorithm. The reference signature utilized by the reference clustering is a sequence of events. It concatenates the *Call-Path* signatures by adding a sequence number to each MPI event, and features parameter clustering by keeping the parameters of each MPI event uncompressed.

In Section 5, we provide the results of the experiments conducted with different benchmarks to compare the space complexity of the ACURDION and the reference signature algorithms.

4 Experimental Setup

TACC's Stampede [25], a state-of-the-art HPC cluster, is utilized to conduct experiments. It consists of a total of 6400 nodes, each with two Intel Xeon E5 processors and one Intel Xeon Phi coprocessor. The compute nodes are interconnected with Mellanox FDR InfiniBand technology (56 Gb/s) in a two-level fat-tree topology.

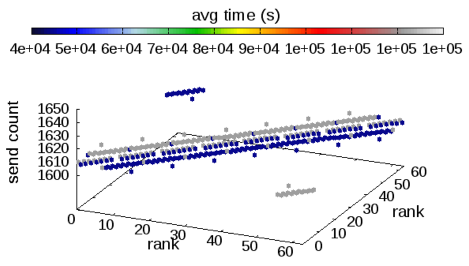
Each experiment was run five times, and their averages are reported. The aggregate wall-clock times across all nodes for these benchmarks are reported. We conducted experiments with a variety of codes: (1) the NPB suite (version 3.3 for MPI) with class D input size [2]; (2) Sweep3D [15], a solver for the 3-D, time-independent, particle transport equation on an orthogonal mesh, which uses a multidimensional wavefront algorithm for discrete ordinates deterministic particle transport simulation with a problem size of $100\times100\times1000$; (3) LULESH, which approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements [12]. Results of ACURDION are compared to reference clustering and related work on signature-based clustering [1] of which we obtained a copy.

5 Results and Analysis

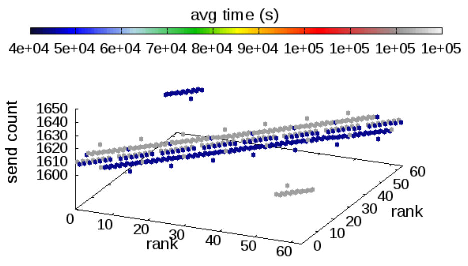
To assess the accuracy of the proposed clustering algorithm, we conducted two types of experiments. First, we tested the accuracy of point-to-point communication through heatmaps to make sure ACURDION captures communication patterns. Second, to verify the accuracy of collective and point-to-point operations, we replayed the traces and compared the wall-clock time of the clustered and non-clustered versions.

Our first experiment assesses the effect of ACURDION on point-to-point communication. Fig. 8 and Fig. 9 depict heatmaps of different benchmarks for 64 processes each. Original versions and ACURDION versions of the benchmarks are shown as pairs (original first, e.g., BT, ACURDION next marked as "starred", e.g., BT*). The x- and z-axes denote mutual communication end-points, and the $number\ of\ sends$ is depicted on the y-axis. The average time in seconds is depicted as heatmaps (dark=low to white=high). The ACURDION heatmaps are a perfect match to the non-clustering ones for BT, Sweep3D, LU and SP. The heatmaps for Lulesh differ slightly before (Fig. 9(e)) and after (Fig. 9(f)) clustering (same for CG and MG).

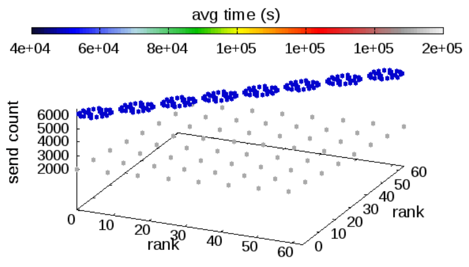
Table 1 indicates varying parameters selected during the signature building phase for these benchmarks. On average, the size of the signature was reduced by 43% across all 11 possible signatures. There were no differences for *DATA+OP*, *KEY*, and *COLOR* for the tested benchmarks. However, benchmarks, such as NAS Fast



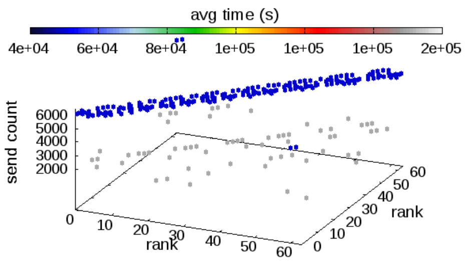
(a) BT



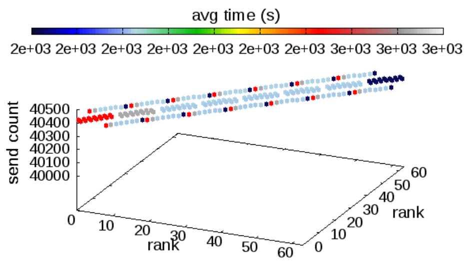
(b) BT*



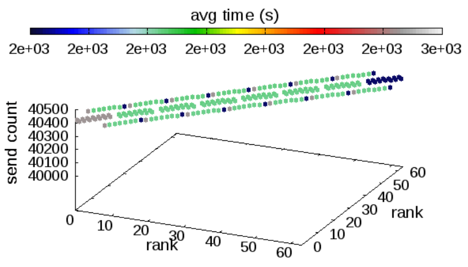
(c) CG



(d) CG*



(e) LU



(f) LU*

Fig. 8. Heatmaps of Point-To-Point Communication for 64 Processes Through ACURDION (K=9)

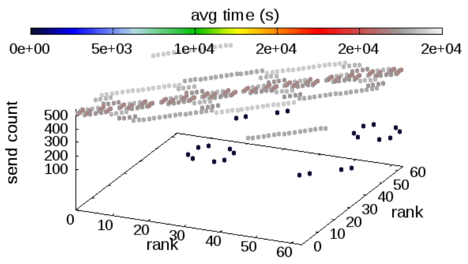
Fourier Transform (FT) tested in our prior work [1], could benefit from *KEY* and *COLOR* signatures.

We next define an accuracy metric of trace replay as

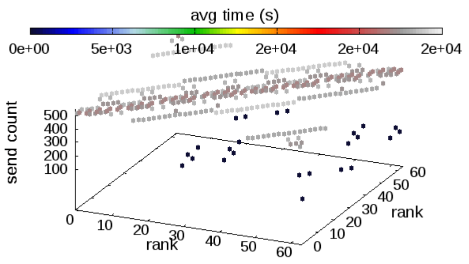
$$ACC = 1 - \frac{|t - t'|}{t}$$

where t and t' are the replay times without and with clustering, respectively.

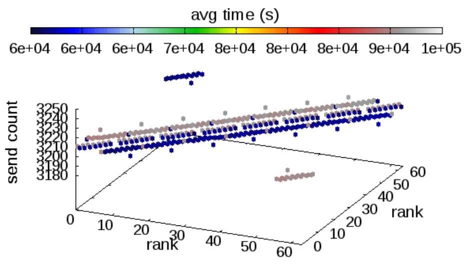
Table 2 covers the percentage of matching clustered parameters relative to non-clustered ones. The similarity of point-to-point events was already depicted in Fig. 8 and Fig. 9. Some codes may experience different endpoints in sends/receives after clustering, but the overall patterns are preserved. In other words, if concrete endpoints differ then only by a slight shift so that the overall behavior remains close to the original program, which is also confirmed in terms of wallclock time later.



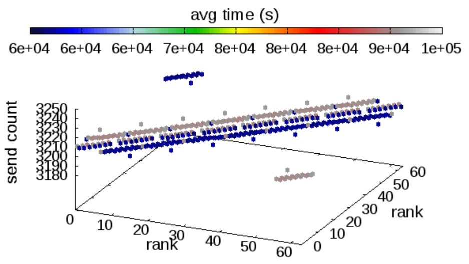
(a) MG



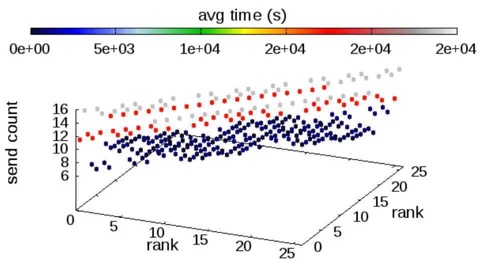
(b) MG*



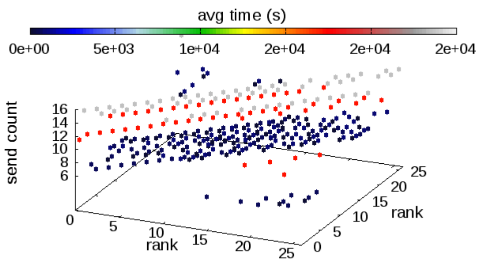
(c) SP



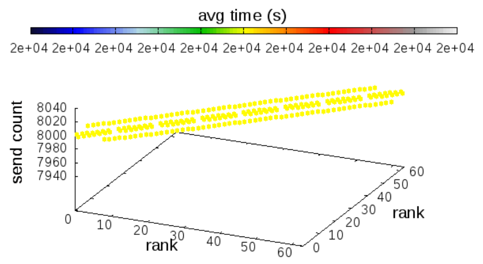
(d) SP*



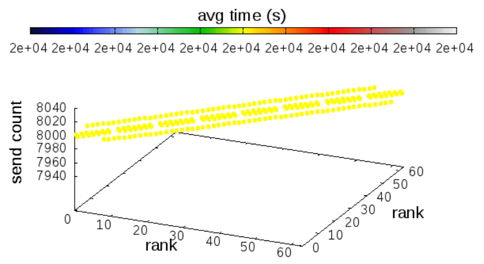
(e) Lulesh



(f) Lulesh*



(g) S3D



(h) S3D*

Fig. 9. Heatmaps of Point-To-Point Communication for 64 Processes Through ACURDION (K=9)

We chose a maximum number of nine clusters ($K = 9$) for ACURDION, which we experimentally determined based on captured communication patterns of related work [1]. This suffices to represent average communication time, send count, and source and destination ranks for point-to-point communication. Table 3 also shows that for CG increasing the number of clusters does not improve the accuracy of

Table 1
Signature Format

Code	Call-Path	COUNT	SRC	DEST	COMP	COMM	TAG	LOOP
BT		✓	✓	✓	✓	✓	✓	
CG			✓	✓	✓	✓		
LU	✓	✓	✓	✓	✓	✓		
MG	✓	✓	✓	✓	✓	✓		✓
SP		✓	✓	✓	✓	✓		
Sweep3D	✓	✓	✓	✓	✓	✓	✓	✓
Lulesh	✓	✓	✓	✓	✓	✓		

Table 2
Matching Percentage

Benchmark	COUNT	LOOP	# EVENTS	SRC	DEST	TAG
BT	99.9%	100%	100%	100%	100%	100%
CG	100%	100%	100%	80.90%	80.90%	100%
LU	97.12%	98.16%	100%	100%	100%	100%
MG	99.7%	100%	100%	96.80%	96.80%	99.03%
SP	100%	100%	100%	100%	100%	100%
Sweep3D	96.86%	87.37%	100%	100%	100%	100%
Lulesh	82.35%	75%	100%	70.37%	70.37%	100%

trace. In fact, we 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. Due to the iterative nature of parallel programs, the number of different *Call-Path* patterns is limited to a small number. For our tested benchmarks, this value was nine, which is sufficient to cover stencil codes. Nonetheless, we can change the value of K *dynamically* should the number of *Call-Path* patterns increase during adaptive signature building.

Table 3
Accuracy and Number of Clusters: CG Class D and P=256

# Clusters	9	27	81	243
Overhead (s)	1.09	3.22	9.54	27.91
Accuracy	98.92%	99.05%	96.88	97.75%

All benchmarks result in the same number of main clusters and sub-clusters under ACURDION, which often reflects the shape of communication patterns in these codes.

But we also observed minor differences. For example, BT and SP have the same communication pattern but slightly different message payloads ($COUNT$ parameter) after clustering, a difference of around 0.1% reported in Table 2, which is not visible in Fig. 8(a). Communication patterns are always retained after clustering, e.g., the stencil pattern of $Sweep3D$ and LU. Here, S3D* retains pattern and send volumes while LU* reflects the communication pattern but diverges slightly in send volume ($COUNT$) under ACURDION.

The Lulesh (Livermore Unstructured Lagrange Explicit Shock Hydrodynamics) proxy application (proxy app) [12] is a shock hydrodynamics code developed at Lawrence Livermore National Laboratory (LLNL). Lulesh is large enough to be more complex than traditional benchmarks, yet compact enough to support a large number of implementations [12]. We observe a slight diffusion in the communication pattern in Fig. 9(f). However, clustering covered all events and retained parameters such as $COUNT$ and $LOOP$ as Table 2 showed.

MG has different clusters as data partitioning (sub-grid creation) depends on input size, number of processes, and two different communication patterns (halo/boundary exchange and cross-grid interpolation). Due to changes in grid resolution per iteration, boundary geometry also changes. As the algorithm moves from coarser to finer, more boundaries are created. Nonetheless, heatmaps show that ACURDION clustering captures this pattern relatively accurately as it closely resembles that without clustering (albeit with some shifts of individual points). Table 2 also indicates that clustering accurately covers the parameters.

CG shows similar shifts that diffuse the regularity of the pattern without clustering, but the overall number of communication exchanges and the send volume are retained. Table 2 indicates that ACURDION clustering captures the parameters accurately. In related work [1], CG's pattern could only be captured via a user-supplied plugin function, which captured unique parameters that otherwise would significantly increase the total number of clusters. It would be preferable to avoid such user plugins where possible as it is difficult for users to provide such functions for complex benchmarks. ACURDION provides the means to retain a concise trace representation without user plugins, but the price is a more diffuse communication matrix. We will later see that this has little effects on replay accuracy, which shows the benefits of using clustering.

5.1 Strong Scaling

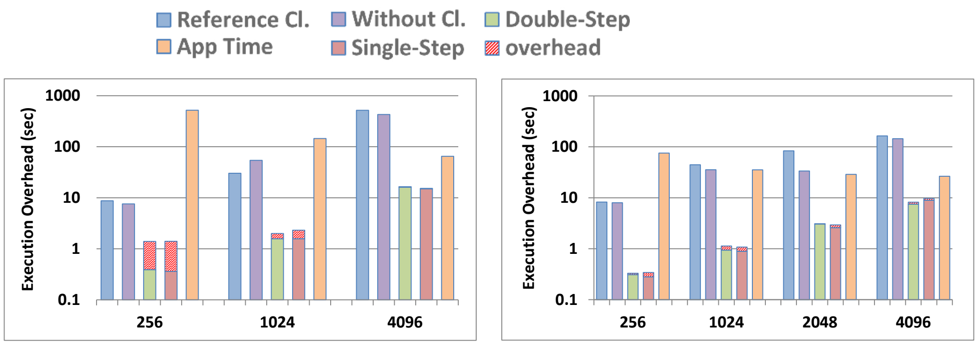
The second set of experiments focuses on the accuracy of wall-clock time comparing replayed traces with ACURDION and without clustering, first under strong and then under weak scaling. Some of the benchmarks only support strong scaling (most NAS codes) while others support weak scaling (Sweep3D, Lulesh) so that different sets of benchmarks are reported in these experiments.

Strong scaling features a set of experiments where the number of tasks is changed while the program input remains the same. This effectively reduces the amount of work per task as the input problem is partitioned into smaller pieces while potentially inflicting more (but smaller) messages as the number of tasks increases. In these experiments, 16 MPI tasks were mapped onto one node (with 16 cores).

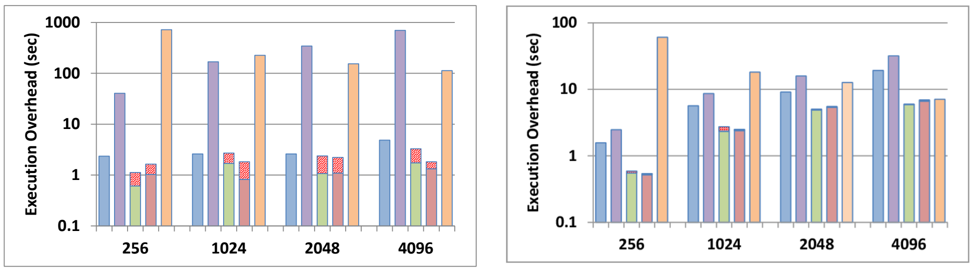
Fig. 10 depicts the wall-clock time on a logarithmic scale (y-axis) for different number of MPI tasks (x-axis) of the respective benchmarks. Per task size, the average execution time over five runs is reported for (a) reference clustering, (b) no clustering (vanilla ScalaTrace V2 with intra- and inter-node reduction), (c) double-step and (d) single-step ACURDION clustering, and (e) base application time without instrumentation. For ACURDION results, bars are stacked to distinguish the base instrumentation overhead (blue/bottom) from the clustering overhead (red/top). This distinction is omitted for reference clustering. The last bar (e) is shown as a reference to get an idea how much time would be spent on tracing compared to base application runtime.

For instance, BT for 256 tasks has about an order of magnitude lower trace overhead with ACURDION clustering than without (or with reference clustering), which is nearly two orders of magnitude smaller than application runtime. Within ACURDION, half the time is spent in clustering. For 4096 tasks, ACURDION incurs an order of magnitude lower overhead than reference/no clustering but results in application overhead of about 20%. The clustering time, however, within ACURDION is negligible. Similar observations were made for CG and SP. LU has low instrumentation overhead under clustering (even reference clustering) while the overhead without clustering is significant and outstrips application runtime at 4096 tasks. ACURDION cuts down overheads to about half or even a quarter of that for reference clustering, which is nearly two orders of magnitude smaller than application runtime regardless of the number of tasks (up to 4096 tasks). Most of the tracing overhead is due to clustering under ACURDION. MG's overhead for ACURDION clustering changes from being two orders of magnitude smaller than application runtime at 256 tasks to match application runtime at 4096 tasks, yet remains about an order of magnitude smaller than reference and no clustering. Overall, single- and double-step clustering perform equally well, and ACURDION outpaces the other techniques due to the lower number of processes involved in inter-node compression after clustering.

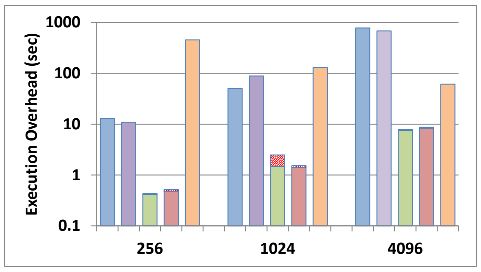
The next set of experiments assess the accuracy of the trace information obtained in the techniques featured so far. To this end, a trace replay tool, ScalaReplay [31], is utilized to issue MPI events in the same order and over the same number of nodes that they were originally recorded during application execution. Yet, instead of computing, the recorded time spent for computation is "replayed" as sleep time to resemble the same distance between communication calls. The communication



(b) BT (c) CG



(d) LU (e) MG

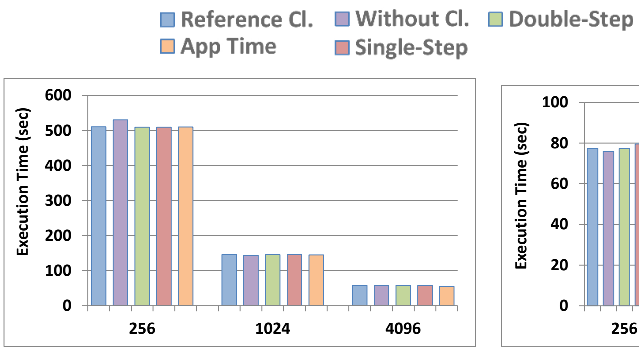


(f) SP

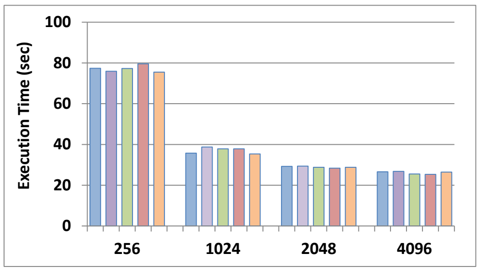
Fig. 10. Execution Overhead for NAS benchmarks: Strong Scaling, Nodes/Tasks=1/16

calls themselves are issued with the same parameters as recorded, except for slight differences in send volume and end-points due to clustering (see previous discussion about communication patterns). The message payload is a buffer of the indicated size (but with some random content as content is not recorded during tracing, nor is it required for correct replay as computation has been replaced by sleep). Nodes interpret the same trace file during replay but transpose MPI communication endpoints relative to their task ID (due to the relative encoding of end-points in ScalaTrace). For clustering, a different event is generated per cluster, which results in up to $K = 9$ different events for subsets of tasks (compared to a single event without clustering). All other parameters are replayed directly from the trace.

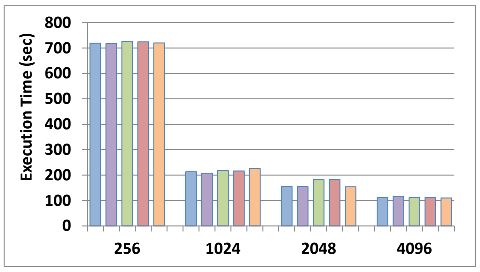
Fig. 11 depicts the replay time in seconds on a linear y-axis for traces resulting from clustering as opposed to *not* using reference clustering, no clustering, single-/double-step ACURDION clustering and also the corresponding application time without instrumentation for comparison. A match to the latter means that traces retain application behavior. We observe that the replay times over all methods, task sizes and applications matches the original application very closely. We observe



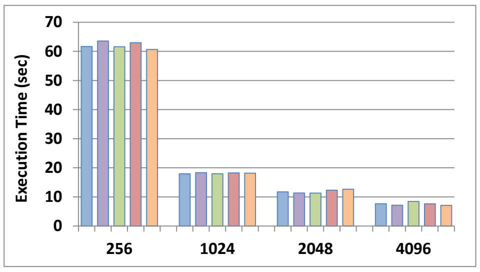
(b) BT



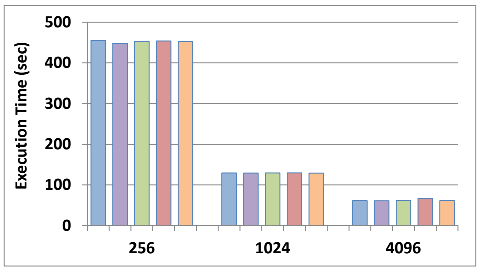
(c) CG



(d) LU



(e) MG



(f) SP

Fig. 11. Replay Time of Traces: Strong Scaling, Nodes/Tasks=1/16

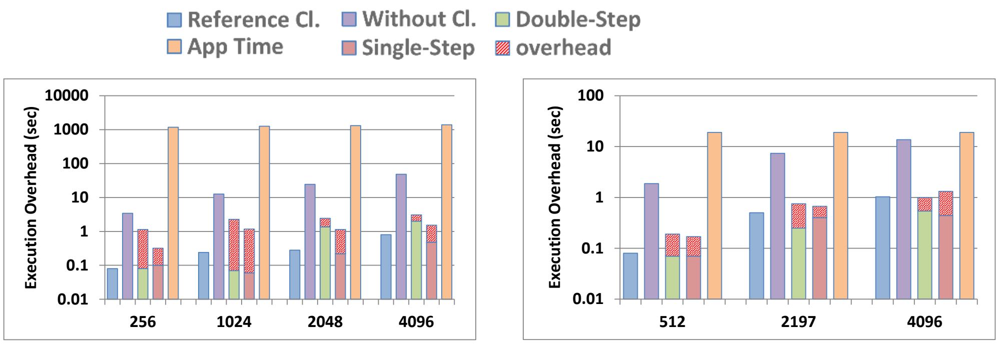
an accuracy level of more than 95% across the set of benchmarks and experimental parameters. This illustrates that ACURDION is competitive with any other scheme, even though it retains only a subset of the trace information of other methods and requires lower overhead.

5.2 *Weak Scaling*

The next experiments cover weak scaling, which features a sequence of experiments where the input size and the number of tasks are increased at about the same rate. The objective is to ensure that the input size per task (after input partitioning) remains constant so that execution times (in the ideal case) also remain constant as we scale up. Of course, changes in communication overhead may influence this behavior. Input constraints on several benchmarks limit the set of experiments that we could conduct for weak scaling to Sweep3D and Lulesh. The input size for Sweep3D is chosen to be $100 \times 100 \times 1000$ per node. For Lulesh, weak scalability tests were run at a problem size of 32^3 per node.

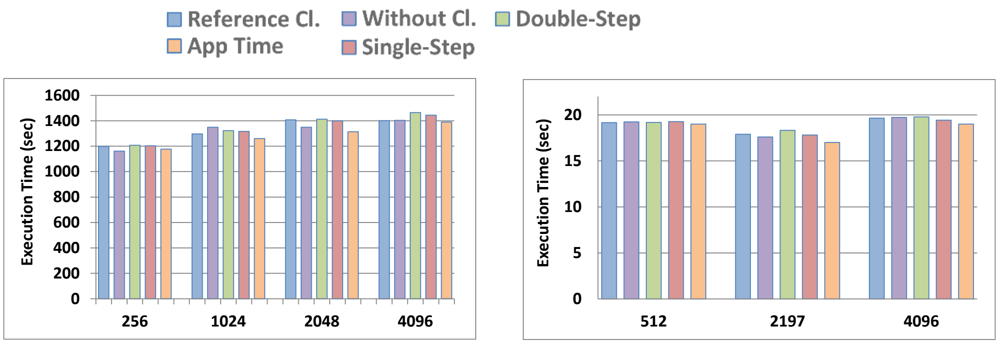
Fig. 12 depicts the overheads in seconds on a logarithmic scale (y-axis) for different number of processors (x-axis) for Lulesh and Sweep3D. Lulesh results in about one order of magnitude lower overhead for any clustering approach than without clustering. Single/double-step ACURDION takes about the same time as reference clustering, even though it has to perform the K-Farthest algorithm. And clustering techniques incur instrumentation overhead 1-2 orders of magnitude smaller than application runtime. Sweep3D results in even lower overheads of 1-2 orders of magnitude for clustering over no clustering. Its instrumentation cost is 3-5 orders of magnitude smaller than the corresponding application runtime. Here, single-step outperforms double-step slightly whereas in all previous experiments no clear winner could be declared between the two. Reference clustering outperforms ACURDION here for the first time. However, we will later show that ACURDION outperforms reference clustering in terms of space complexity, which can have a significant impact for large-scale tracing.

Fig. 13 features the overhead per replay method and in comparison to original application runtime in seconds on a linear scale (y-axis) for a different number of tasks (x-axis). We observe that replay times are uniformly resembling the original application runtime irrespective of which tracing method was used. The overall accuracy of ACURDION is 95%-97%.



(b) Sweep3D (c) Lulesh

Fig. 12. Execution Overhead: Nodes/Tasks=1/16

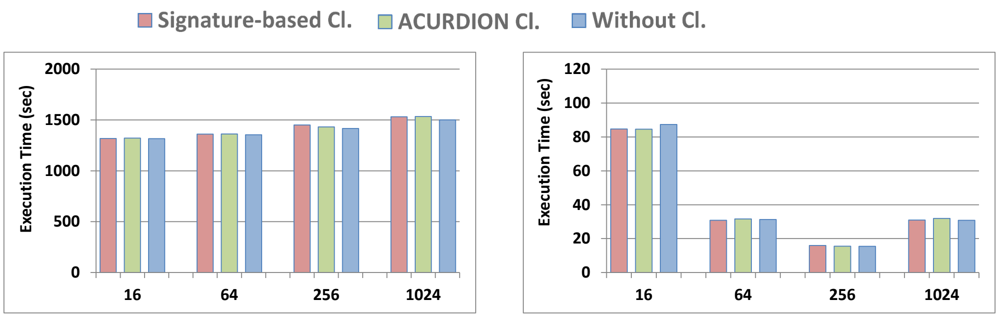


(b) Sweep3D (c) Lulesh

Fig. 13. Replay Time of Traces: Weak Scaling, Nodes/Tasks=1/16

5.3 Signature-based vs. ACURDION

To compare the accuracy of trace files generated by ACURDION to signature-based clustering [1], we conducted two experiments covering both strong and weak scaling on the same platform, where all machines were 2-way SMPs with AMD Opteron 6128 processors with 8 cores per socket. Each node is connected by QDR InfiniBand. In the first experiment, we compared ACURDION and signature-based clustering for CG class C. Figure 14 shows that the accuracy of traces for signature-based and ACURDION are 98.73% and 98.93% compared to non-clustering. Note that for CG class C, the performance deteriorates for P=1024, because there is not enough workload for each process. This results in a high communication to computation ratio. The second experiment covers weak scaling for Sweep3D in Fig. 14. The average accuracy of traces is 97.96% and 97.91% for ACURDION and signature-based, respectively. While ACURDION has a high level of accuracy, it does not have the limitations of the signature-based approach, i.e., no user plug-in is required and it has lower space complexity.



(b) Sweep3D (c) CG

Fig. 14. Replay Time of Traces: Nodes/Tasks=1/16

5.4 K-Farthest vs. K-Medoid

In our implementation, we used K-Farthest clustering at the node level. Overall, ACURDION uses signature-based, hierarchical clustering as the main clustering across the nodes, and K-Farthest or K-Medoid at the node level. We compared the accuracy of intra-clustering for K-Medoids and K-Farthest clusterings. Similar to the previous experiment, we conducted two experiments covering both strong and weak scaling on the same platform. Figure 15 shows that the accuracy of traces are very close to each other. Since the execution overheads depicted in 4 and 6, and standard deviation depicted in 5 and 7 are also very close, we conclude that intra clustering does not play a major role in trace accuracy. The signature-based nature of the clustering is the main factor by which ACURDION covers all MPI events.

Table 4
Execution Overhead: CG Class C - Strong Scaling

# Processes	16	64	256	1024
K-Medoid Overhead (s)	0.41	0.58	0.86	1.94
K-Farthest Overhead (s)	0.42	0.6	0.91	1.99

Table 5
Standard Deviation: CG Class C - Strong Scaling

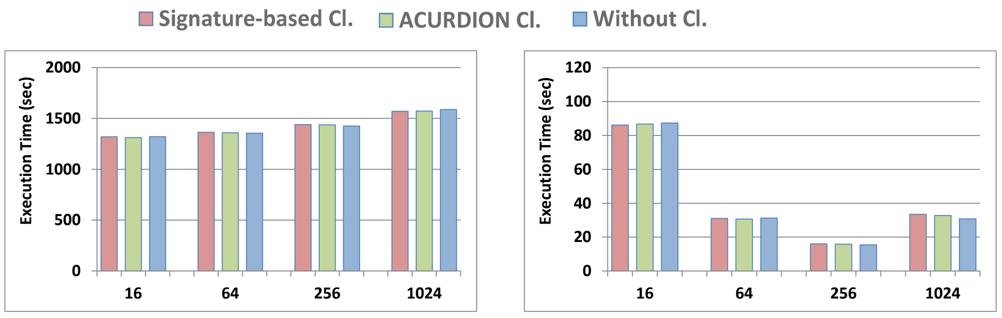
# Processes	16	64	256	1024
K-Medoid Overhead (s)	0.68	0.42	0.45	0.12
K-Farthest Overhead (s)	0.45	0.22	0.34	0.43

Table 6
Execution Overhead: Sweep3D - Weak Scaling

# Processes	16	64	256	1024
K-Medoid Overhead (s)	0.34	0.4	0.36	0.75
K-Farthest Overhead (s)	0.23	0.27	0.29	0.92

Table 7
Standard Deviation: Sweep3D - Weak Scaling

# Processes	16	64	256	1024
K-Medoid Overhead (s)	4.84	2.49	7.01	17.15
K-Farthest Overhead (s)	3.59	3.42	8.85	16.16



(b) Sweep3D (c) CG

Fig. 15. Replay Time of Traces: Nodes/Tasks=1/16

5.5 Space Complexity

Let us consider the space complexity of tracing with and without clustering. The memory space allocated during trace compression with optional clustering differs from method to method. Table 8 depicts the number of clusters and the required memory space in KB per method and per benchmark for 256 tasks. We observe that ACURDION (single/double-step do not differ) requires about an order of magnitude (and up to two orders of magnitude) less space than reference clustering or no clustering. The number of clusters for ACURDION is always nine due to the K-Farthest or K-Medoid methods, which is often significantly smaller than any other technique. (Notice that without clustering, each task is considered a cluster of its own).

Table 8
Average Space Complexity Per Process for P=256 (P=216 for Lulesh)

	ACURDION		Reference Clustering		Without Clustering	
Code	# clusters	avg space	# clusters	avg space	# clusters	avg space
BT	9	2.73KB	41	108.49KB	256	71.71KB
CG	9	1.70KB	256	376.32KB	256	43.82KB
LU	9	2.73KB	16	25.05KB	256	71.71KB
MG	9	11.8KB	72	733.83KB	256	215.15KB
SP	9	2.50KB	53	133.06KB	256	67.73KB
Sweep3D	9	1.50KB	9	4.86KB	256	27.89KB
Lulesh	9	1.51KB	26	17.56KB	216	27.87KB

The required space of ACURDION is on average one order of magnitude larger compared to signature-based clustering [1], i.e., it ranges from slightly smaller (MG) to two orders of magnitude larger (LU). And the number of clusters of ACURDION is sometimes smaller and sometimes larger compared to signature-based clustering. But more significantly, the case (MG) where the number of clusters grows linearly with the number of tasks for signature-based clustering presents a non-scalable behavior. In contrast, ACURDION always requires only $K = 9$ clusters and still retains similar overheads at a constant trace size. This is a significant advance in terms of scaling behavior.

This scalability result is corroborated by a complexity analysis of the algorithms. Without clustering, all processes contribute to the inter-compression step, so the space complexity is linear to number of processes. But for clustering, only a constant number of representative nodes are involved in this operation (e.g., one node per cluster). Furthermore, the size of signatures is a key player for clustering. Thus, we considered the size of signatures and related algorithms for space complexity analysis (e.g., adaptive signature building for ACURDION).

Prior work established the complexity without clustering (1), reference clustering (2) and signature-based clustering (3), which resulted in the lowest overhead. In contrast, the complexity of K-Farthest or K-Medoid clustering (4) is even lower than (3) since it depends on the constant K for ACURDION, and we showed that $K = 9$ suffices in experiments. Specifically, the average trace and signature sizes per node are multiplied by the constant K (instead of the sum of main clusters and sub-clusters in prior work, which is not constant for some programs, such as MG).

5.6 Dynamic Clustering

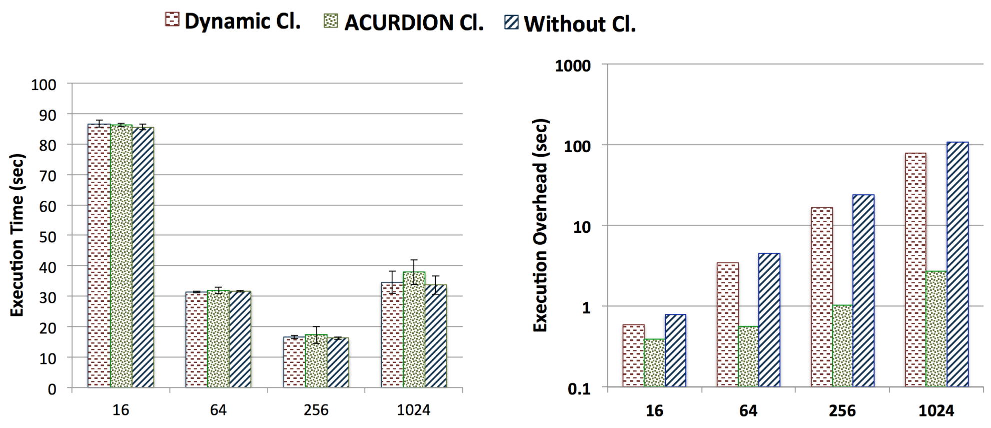
We developed yet another clustering algorithm derived from ACURDION, which only considers Call-Path, SRC, and DEST signatures. It hierarchically clusters at two levels. First, it clusters based on Call-Path signature. Second, it finds new clusters based on SRC and DEST signatures. The hypothesis for designing dynamic clustering was that all events and the full communication pattern can be captured by only considering three signatures and would result in output traces with a very high replay accuracy even though other signatures are ignored.

We tested dynamic clustering against ACURDION with nine clusters and the original ScalaTrace without clustering. Figure 16 and Figure 17 depict replay time and execution overhead for CG class C (strong scaling), and LU class C (weak scaling). The accuracy of ACURDION and dynamic clustering are almost the same. The main difference is the number of clusters. For CG, due to the unique communication patterns for each process, the number of SRC/DEST signatures goes up significantly, which results in much larger overhead for dynamic clustering.

Programmers could use Dynamic clustering to gain a better understanding about the number of clusters. We conducted another experiment in which we chose smaller numbers of clusters (less than 9) for LU to check the accuracy of traces. The replay engine was unable to execute the trace, simply because some of the events are now missing, which creates deadlocks. We conducted the same experiment for BT and reduced the number of clusters to less than three so that the output trace misses the correct communication pattern. However, because the trace still contained all the events, the replay engine executed successfully. Surprisingly, the replay time was accurate. Based on the HPC benchmarks that we tested, it seems the Call-Path is the root cause of accuracy. As long as the HPC tracing toolset captures all the events (Call-Paths), the tracing toolset is accurate in terms of timing.

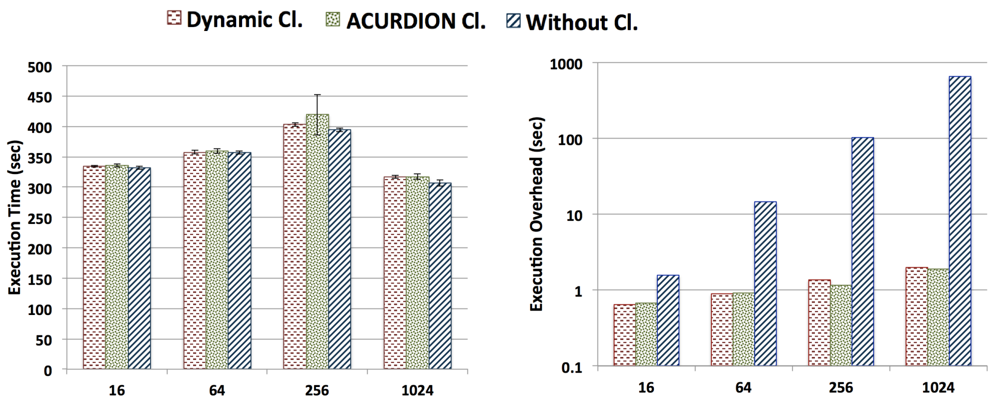
6 Related Literature

Bahmani and Mueller [1] proposed a signature-based clustering algorithm for ScalaTraceV2. ACURDION enhances this work in two directions. First, the parameter signature of [1] is a concatenation of several truncated parameters. At a large scale, 16 bits for representing average COUNT may not be enough to cover all differences in COUNT. The eleven 64-bit signatures that are created based on characteristics of the application by the Adaptive Signature Building phase avoid such deficiencies in ACURDION. Second, the strength of ACURDION lies in its independence of



(b) Replay Time (c) Execution Overhead

Fig. 16. Execution Overhead and Replay Time for CG Class C: Nodes/Tasks=1/16



(b) Replay Time (c) Execution Overhead

Fig. 17. Execution Overhead and Replay Time for LU Weak Scaling: Nodes/Tasks=1/16

user input plug-ins for benchmarks that have unique communication behavior (e.g., CG).

CYPRESS [36] is a communication trace compression framework. CYPRESS combines static program analysis with dynamic runtime trace compression. It extracts the program structure at compile time, obtaining critical loop/branch control structures. They compared their result with ScalaTraceV2. As previously mentioned, the problem of ScalaTraceV2 was at the inter-compression step. In terms of space complexity, [1] and ACURDION have at least an order of magnitude smaller traces than ScalaTraceV2. One of the main problems of CYPRESS is the overhead of static and dynamic analysis while the combination of ScalaTraceV2+Clustering does not have any overhead at the compile time of applications. Unlike CYPRESS, such an approach works even when only binaries/libraries are available for instrumentation.

Scalasca [37] collects profile and trace data and feeds it into an automatic analysis process to detect specific bottleneck patterns. The main limitation of Scalasca is the slow and inefficient analysis report post-processing, which is inconvenient both for experiments with complex calltrees or large numbers of processes/threads.

Vampir [5], which is a tracing tool for MPI communication, uses profiling extensions to MPI and facilitates the analysis of message events of parallel execution, helping to identify bottlenecks and inconsistent run-time behavior. Scalability is the main problem of Vampire where trace complexity increases with the number of MPI events in a non-scalable fashion.

Statistical sampling is a method utilized in HPCTOOLKIT [27] to measure performance. HPCTOOLKIT provides and visualizes per process traces of sampled call paths. All of the call paths are presented for all samples (in a thread) as a calling context tree (CCT). A CCT is a weighted tree whose root is the program entry point and whose leaves represent sample points.

A density-based clustering analysis was proposed in [19,10,9] that 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 in [16] and [17]. The above-mentioned clustering algorithms are expensive in terms of time complexity, especially for large-scale sizes. On the other hand, our work is a low overhead clustering algorithm with $O(logP)$ complexity.

Phantom [35], 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.

A new algorithm for K-means clustering called Yinyang K-means was proposed in [6]. By clustering the centers in the initial stage and leveraging efficiently maintained lower and upper bounds between a point and centers, Yinyang K-means more effectively avoids unnecessary distance calculations than prior algorithms. As we conducted an experiment for both K-Medoid and K-Farthest clusterings, the intra clustering does not play a significant role in our implementation. Therefore, we did not utilize Yinyang K-means in our implementation.

A parallel clustering algorithm based on CLARA [13] was proposed in CAPEK [7] 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, finding an appropriate rate of sampling is complicated, and the cost of having a dense CCT is high. In contrast, ACURDION/ScalaTraceV2 provides a full trace file without resorting to sampling and it does so at very low cost by leveraging 64-bit stack signatures.

The Stack Trace Analysis Tool (STAT) [18] provides scalable detection of task equivalence classes based on the functions that the processes execute. The Probabilistic Calling Context (PCC) approach [4] continuously maintains a probabilistically unique value representing the current calling context in a hash table. STAT and PCC only consider stack traces. Therefore, if two processes can exhibit the same stack trace despite having very divergent timing characteristics, these tools cannot distinguish the difference. On the other hand, ACURDION not only covers different stack traces, but also captures other characteristics of processes (e.g., timing characteristics) by considering 11 signatures.

7 Conclusion and Future Work

This work contributes ACURDION, a novel signature-based clustering algorithm with a low time complexity of $O(logP)$ and low space overheads. A signature finding algorithm prunes unnecessary metrics and only considers parameters representing differences among the traces of nodes.

We evaluated ACURDION in comparison to other clustering algorithms for a set of HPC benchmarks showing its effectiveness at capturing representative application behavior for communication events. ACURDION is superior to past work because it is more scalable in terms of space and time complexities at sustained accuracy. And in contrast to other work, it does not rely on user plugins, which may be hard to construct. Experiments showed that without loss of accuracy, only nine clusters suffice to represent the behavior of a wide set of HPC benchmarks codes.

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] 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.

[3] 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.

[4] Michael D Bond and Kathryn S McKinley. Probabilistic calling context. In *ACM SIGPLAN Notices*, volume 42, pages 97–112. ACM, 2007.

[5] 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.

[6] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 579–587, 2015.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] I. Karlin, A. Bhatele, J. Keasler, B.L. Chamberlain, J. Cohen, Z. DeVito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C.H. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *Parallel Distributed Processing (IPDPS)*, pages 919–932, May 2013.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Gregory L Lee, Dong H Ahn, Dorian C Arnold, Bronis R De Supinski, Barton P Miller, and Martin Schulz. Benchmarking the stack trace analysis tool for bluegene/l. In *PARCO*, pages 621–628, 2007.

[19] 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.

[20] G. Marin and J. Mellor-Crummey. Cross architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2004.

[21] 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.

[22] Hae-Sang Park and Chi-Hyuck Jun. A simple and fast algorithm for k-medoids clustering. *Expert Systems with Applications*, 36(2, Part 2):3336 – 3341, 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] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing*, November 2002.

[25] Texas Advanced Computing Center Stampede. http://www.tacc.utexas.edu/resources/hpc/stampede, 2014.

[26] Ryutaro Susukita, Hisashige Ando, Mutsumi Aoyagi, Hiroaki Honda, Yuichi Inadomi, Koji Inoue, Shigeru Ishizuki, Yasunori Kimura, Hidemi Komatsu, Motoyoshi Kurokawa, Kazuaki J. Murakami, Hidetomo Shibamura, Shuji Yamamura, and Yunqing Yu. Performance prediction of large-scale parallell system and application using macro-level simulation. In *Supercomputing*, 2008.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Supercomputing*, November 2005.

[34] Jidong Zhai, Wenguang Chen, and Weimin Zheng. Phantom: Predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 305–314, 2010.

[35] 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.

[36] 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.

[37] Ilya Zhukov and Brian J. N. Wylie. Assessing measurement and analysis performance and scalability of scalasca 2.0. In *Proc. of the Euro-Par 2013: Parallel Processing Workshops*, volume 8374 of *Lecture Notes in Computer Science*, pages 627–636. Springer, 2014.

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