ABSTRACT

BAHMANI, AMIR HASSAN. Scalable Communication Tracing via Clustering. (Under the direction of Rainer Frank Mueller.)

Scientific computing applications continue to push the envelope to meet the ever increasing demand for computational power. Extreme-scale computing poses a number of challenges to application performance. Developers need to study application behavior by collecting detailed information with the help of tracing toolsets to determine shortcomings. Today's tracing tools either obtain lossless trace information at the price of limited scalability, or preserve only aggregated statistical trace information to conserve the size of trace files.

One effective solution to address these challenges is to empower tracing toolsets with effective machine learning algorithms such as clustering processes that exhibit similar execution behavior. Instead of collecting performance information from each individual node, this information can be collected from just a set of lead nodes, one per cluster. This document proposes three logarithmic clustering approaches:

1. CallPath+Parameter Clustering contributes a fast, scalable, signature-based clustering algorithm. Instead of prior work based on statistical clustering, CallPath+Parameter clustering produces precise results without significant loss of events or accuracy.

2. An adaptive clustering algorithm for large-scale applications, called ACURDION, identifies the parameters that differ across processes by using a logarithmic algorithm called Adaptive Signature Building. Furthermore, 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 while retaining sufficient accuracy of trace events and parameters.

3. Chameleon contributes an *online*, *fast*, and *scalable* signature-based clustering algorithm. Unlike related work, Chameleon creates this trace incrementally during the execution of applications and only for lead processes.

Overall, this work is centered around scalable tracing of parallel applications. Built upon prior research, it contributes novel clustering approaches on communication trace compression.

Scalable Communication Tracing via Clustering

by
Amir Hassan Bahmani

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2017

APPROVED BY:

Yan Solihin	Xipeng Shen
Guoliang Jin	Rainer Frank Mueller Chair of Advisory Committee

DEDICATION

I dedicate this work to my deceased wife, Someyra, my parents and siblings, and my academic advisor Prof. Frank Mueller.

BIOGRAPHY

Amir Bahmani attended NC State University from 2012 to 2017.

ACKNOWLEDGEMENTS

I have always liked what Rocky Balboa (via Sylvester Stallone) said:

"Let me tell you something you already know. The world ain't all sunshine and rainbows. It's a very mean and nasty place and I don't care how tough you are it will beat you to your knees and keep you there permanently if you let it. You, me, or nobody is gonna hit as hard as life. But it ain't about how hard ya hit. It's about how hard you can get hit and keep moving forward. How much you can take and keep moving forward. That's how winning is done!"

I've gone through a lot during my doctoral studies (lost my wife to cancer, my dad had a severe car accident and he's paralyzed, dealing with many restrictions due to my nationality (e.g., travel restriction, collaboration), but I never gave up. I want to keep moving forward through using the skills and knowledge from my computer science training to advance medical research.

I have to thank so many people, but the first of them is my advisor, Frank Mueller. I cannot explain how wonderful he has been to me. He is the Rocky Balboa of Supercomputing.

I want to thank my committee members, Dr. Yan Solihin, Dr. Xipeng Shen, and Dr. Guoliang Jin, and also those who helped me developing applications: Dr. Xing Wu, Dr. James L. Morrison, Alexander B. Sibley, Dr. Kouros Owzar, Dr. Marc Niethammer, Dr. Mahmoud Parsian, Dr. Martin Styner, Dr. Lisandro Dalcin, Sarah Baucom, Monis Khan, Hussein Koprly, and Spencer Moore.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Scientific computing applications continue to push the envelope to meet the ever increasing demand for computational power. This trend is driven by a need to increase model resolution by orders of magnitude combined with multi-level simulation combining models at different granularity. High-performance computing (HPC) hardware platforms are struggling to keep pace with these demands as a number of challenges are posed in terms of hardware and software advances for next-generation HPC at exascale Flop (Floating-point operations per second) rates. To effectively utilize such extreme-scale HPC platforms, developers need to observe and tune application behavior to ensure their algorithms still scale to a larger number of nodes and cores, a process that is typically repeated for each order-of-magnitude increase in compute capability (Flops). This process is generally aided by collecting detailed information with tracing toolsets to determine algorithmic, software and hardware resource shortcomings. This allows developers to study application behavior of such performance information utilizing performance analysis tools. While applications may be considered "scalability challenged" when exposed to yet another larger platform, current tracing toolsets also fall short of exascale requirements: They can no longer ensure a low background overhead of their tool workload since trace collection for each execution entity is becoming infeasible at extreme scales for hundreds of thousands of cores and beyond. Most tools either obtain lossless trace information at the price of limited scalability, such as Vamipir [11], or preserve only aggregated statistical trace information to conserve the size of trace files, as in mpiP [56].

At extreme scale, tracing tools, linked with applications, could severely affect the efficiency and scalability of the system. The tracing background subsystem 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. Therefore, tool designers need to develop new strategies to address these problems.

1.1 Hypothesis

As we are approaching Exa-Scale computing, application tracing may become an essential step in program tuning. Yet, tracing frameworks themselves have to be efficient and scalable to minimize application perturbance and I/O requirements for the traces. In fact, collecting detailed performance information comprehensively for all nodes may not be feasible from a scalability perspective. Therefore, tool designers need to develop new strategies to address these problems. We attempt to address this limitation in this dissertation. The hypothesis of this dissertation is:

The HPC-prevalent SPMD programming style and the iterative nature of scientific computing algorithms provide an opportunity to use machine learning algorithms in communication tracing. These machine learning algorithms group processes with similar application behavior together, thereby providing an opportunity to reduce tracing costs to few representative nodes with the potential to enable scalable performance analysis, much in contrast to past approaches.

1.2 Contributions

1.2.1 Contributions

This work contributes the following fundamentally new approaches to improve communication tracing techniques:

1. One effective solution is to empower tracing toolsets with effective machine learning algorithms such as clustering processes with the different behavior into groups. Instead of collecting performance information from each individual node, this information is collected from just a set of lead nodes. *CallPath+Parameter* clustering contributes a fast, scalable, signature-based clustering algorithm that clusters processes exhibiting similar execution behavior. Instead of prior work based on statistical clustering, our approach produces precise results nearly without loss of events or accuracy. The proposed algorithm combines low overhead at the clustering level with $log(P)$ time complexity, and it splits the merge process to make tracing suitable for extreme-scale computing. Overall, this multi-level precise clustering approach based on signatures generalizes to a novel multi-metric clustering technique with unprecedented low overhead.

2. Our second solution is an adaptive clustering algorithm with eleven 64-bit signatures called *ACURDION* that traces the MPI communication of MPI codes with O(log P) time

complexity. ACURDION has the following features: 1) It avoids expanding signatures blindly by identifying the parameters that differ across processes using a logarithmic algorithm called Adaptive Signature Building. 2) ACURDION supports top K clustering algorithms (e.g., top K-medoid, K-furthest) to cluster processes based on the pruned parameters. 3) Our 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.

3. We developed Chameleon, which contributes an *online* signature-based clustering algorithm. Unlike related work, Chameleon creates a global trace incrementally during the execution of applications and only for lead processes. It considers parallel applications using the SPMD (single program multiple data) paradigm that relies on iterative kernels. Chameleon also identifies different program phases, clusters processes exhibiting different execution behavior, and creates a compressed global trace file on-the-fly. This global trace is created incrementally at interim execution points of the application execution. The overall system combines low overhead at the clustering level with a time complexity of $C \times log(P)$, where C is the number of re-clusterings and P is the number of processes. For the set of tested HPC benchmarks and applications, re-clustering is only needed once after the initial clustering and results in similar time complexity for the *Chameleon*, *CallPath+Parameter* and *ACURDION* algorithms.

Experiments were performed to evaluate these approaches for trace-based performance analysis. Results demonstrate that 1) all of the proposed clustering algorithms reduce the overhead of the inter-compression phase by at least 2-3 orders of magnitude, and for all tested benchmarks the overhead is less than 50% of total program execution time, 2) all of these algorithms provide high accuracy traces (over 87% accuracy relative to the actual application execution time), 3) the proposed algorithms are logarithmic and scalable, 4) all of the algorithms significantly reduce space complexity, 5) *Call-path+Parameter* clustering is solely based on two 64-bit signatures and is very efficient in terms of space complexity, 6) *ACURDION* and *Chameleon* decide the format of signatures involved in clustering on-the-fly (on average, the size of signatures is reduced by at least 43% across all 11 signatures) and 7) Chameleon, unlike all prior work, supports online inter-compression by detecting application phase changes and merges intermediate traces.

Overall, this work is centered around scalable tracing of parallel applications. It contributes novel clustering approaches on communication trace compression.

1.3 Organization

The document is structured as follows.

Chapter 2 introduces prior research on ScalaTrace that inspired and enabled the work presented in this dissertation. Chapter 3 presents a CallPath+Parameter clustering algorithms that leverages two 64-bit signatures and a hierarchical clustering algorithm for quick clustering analysis at large scale. Chapter 4 describes ACURDION, which is an adaptive signature-based clustering algorithm. ACURDION first builds the size of the signature dynamically, and then leverages K-Medoid or K-Furthest hierarchical algorithms to quickly cluster traces.

Chapter 5 describes Chameleon, a signature-based online clustering algorithm that finds application phase changes and cluster traces at the at interim execution points, e.g., at timestep boundaries of scientific codes. Chapter 7 discusses future work and summarizes the contributions of this research and concludes this dissertation.

Chapter 2

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 [41]. Consider the example in the following code snippet:

ALGORITHM 1: Example MPI Code

```
1 for i = 0 → 1000 do
2 |   for k = 0 → 100 do
3 |   |   MPI_Send(...);
4 |   |   MPI_Recv(...);
5 |   end
6 |   MPI_Barrier(...)
7 end
```

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 [41]. Consider Figure 2.1 with a relative encoding of nodes 5 and 9 in terms of communication end-points of -4, -1, $+1$, and $+4$, i.e., these nodes have identical *relative* communication end-points.

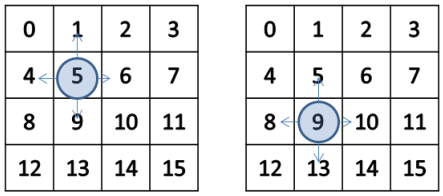


Figure 2.1: Communication End-point Encoding

(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 [41].

(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 [57]. In Figure 2.2(a), the shaded nodes are presented as ranklist ⟨*2 5 2 4 2 1*⟩, and in Figure 2.2(b), they are presented as ranklist ⟨*2 0 4 4 4 1*⟩. The former reads as a 2D ranklist starting at task 5, two entries in the first dimension with a stride of 4 (implying tasks 5 and 9) and two entries in the second dimension with stride 1 (implying tasks 6 and 10).

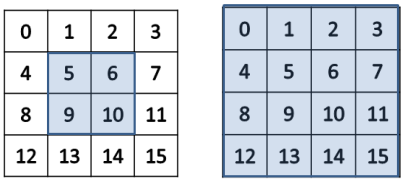


Figure 2.2: Ranklist for Two Communication Groups

ScalaTrace employs a two-stage trace compression technique, namely intra-node (loop level) and inter-node compression.

2.1 Intra-node Compression

Intra-node compression uses the calling context of events to match repetitious behavior. This ensures that identical MPI functions originating from different call paths of the application are not compressed together. Since trace events from different nodes are collected and merged in a single output trace file, the *task rank* information of nodes participating in an event is also compressed and encoded concisely in the compressed trace. This participant node information is represented in a tuple containing starting rank, total number of participants and an offset value separating ranks. Even multi-dimensional information is captured in this encoding format.

2.2 Inter-node Compression

ScalaTrace consolidates traces in a reduction step over a radix tree in the $MPI\_Finalize$ PMPI wrapper. It leverages Longest Common Sequence (LCS) algorithms to find common sequences between traces. The complexity of Inter-Compression is O($n^2 log(P)$); where, n is the number of PRSD events and P is the number of processes.

While intra-compression is fast and efficient, inter-compression is costly as it depends on the number of tasks.

2.3 Time Preservation

Another important feature of ScalaTrace is the time preservation of captured traces. Instead of recording absolute time stamps, the tool records delta time of computation durations between adjacent communication calls. During RSD formation, instead of accumulating exact delta timestamps, statistical histogram bins are utilized to concisely represent timing details across the loop. These bins are comprised of statistical timing data (minimum, maximum, average and standard deviation).

2.4 Timed Replay

ScalaTrace not only supports scalable tracing, it also supports a scalable replay engine. Given a single, compressed trace file, the replay engine allows all I/O and communication calls to be reissued without trace decompression while preserving event ordering. The replay engine runs as an MPI job with the same number of tasks as its original application. It replays I/O and communication events in each node with their original parameters except for actual file content/message payloads. Instead, a random buffer of the same size as the original file/message

buffer is used. Additionally, computation time on each node is simulated by a delay between traced events based on recorded delta times.

Chapter 3

Scalable Performance Analysis of ExaScale MPI Programs through Signature-Based Clustering Algorithms

Scientific computing applications continue to push the envelope on ever increasing demand for computational power. This trend is driven by a need to increase model resolution by orders of magnitude combined with multi-level simulation combining models at different granularity. High-performance computing (HPC) hardware platforms are struggling to keep pace with these demands as a number of challenges are posed in terms of hardware and software advances for next-generation HPC at exascale Flops (Floating-point operations per second) rates. To effectively utilize such extreme-scale HPC platforms, developers need to observe and tune application behavior to ensure their algorithms still scale to a larger number of nodes and cores, a process that is typically repeated for each order-of-magnitude increase in compute capability (Flops). This process is generally aided by collecting detailed information with tracing toolsets to determine algorithmic, software and hardware resource shortcomings. This allows developers to study application behavior of such performance information utilizing performance analysis tools. While applications may be considered "scalability challenged" when exposed to yet another larger platform, current tracing toolsets also fall short of exascale requirements: They can no longer ensure a low background overhead of their tool workload since trace collection for each execution entity is becoming infeasible at extreme scales for hundreds of thousands of cores and beyond. Most tools either obtain lossless trace information at the price of limited scalability, such as Vamipir [11], or preserve only aggregated statistical trace information to conserve the size of trace files, as in mpiP [56].

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. Therefore, tool designers need to develop new strategies to address these problems.

One effective solution is to cluster processes with the same behavior into groups; then, instead of collecting performance information from all individual nodes, such information can be collected from just a single node (or a set of lead processes) per cluster group.

We propose a fast, scalable, signature-based clustering algorithm that clusters processes exhibiting similar execution behavior. We apply our clustering algorithm on trace files created by the public release of ScalaTrace V2 [59], a state-of-the-art MPI message passing tracing toolset. ScalaTrace V2 provides orders of magnitude smaller if not near-constant sized communication traces regardless of the number of nodes while preserving structural information.

ScalaTrace employs a two-stage trace compression technique, namely intra-node and inter-node compression [41, 60]). 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 2). 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 (typically a constant) is the number of MPI events in PRSD compressed notation and P is the number of processes. Our clustering algorithm addresses the high overhead due to scaling out to 100,000+ processor cores by significantly reducing P to a constant for most cases (or a sub-linear term of P for the remaining ones), thereby effectively eliminating this bottleneck.

The proposed clustering algorithm has two levels, the first of which employs *Call-path clustering* based on the stack 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. The first level of clustering distinguishes processes with different execution structures.

Parameter clustering is the second level of clustering. At this level, we use a different signature called the parameter signature. This signature composes parameters of the MPI call event, such as count (number of data elements), type (data type), source, destination, etc., excluding the message content itself. Once the algorithm has clustered processes with different execution structures, with the help of Parameter clustering, we distinguish processes with the same execution structure but different parameters.

The main objective of this work is to generate application traces without perturbing application behavior and with low-overhead that accurately approximate the execution time of the applications. To evaluate the accuracy and scalability of our algorithm, we also designed a *reference clustering* approach based on a reference signature. The reference signature covers Call-path signatures by adding a sequence number to each MPI event as well as Parameter clustering by keeping each MPI event's parameters uncompressed. Detailed implementation information about Call-path+Parameter clustering and reference clustering algorithms are discussed in the following sections.

Contributions:

- We provide a novel multi-level clustering algorithm. By separating aspects in a multi-level approach, the algorithmic complexity of clustering is reduced.

- We develop a unique signature-based clustering methodology. Signatures address the shortcoming of past singular metric approaches to clustering. This allows clustering to be extended to multi-dimensional domains of diverse metrics and equally diverse application scenarios. Signatures again reduce computational clustering overheads since signatures are of constant length.

- We design Call-path clustering of call sequence signatures suitable for program tracing in general. We further compose domain-specific data via parameter signatures and derive clusters capturing common behavior across different execution instances in a highly parallel environment.

- We evaluate the composition of Call-path+Parameter clustering for a set of HPC benchmarks showing that their effectiveness is capturing representative application behavior for communication events. The number of clusters is a constant for most benchmarks and scales sub-linearly in the number of processes for the remaining ones, a significant improvement over linear increases without clustering.

- We demonstrate that application performance is preserved when execution traces composed of a set of just one task per cluster are replayed over the entire original number of processors, where the behavior of other tasks in a cluster is derived from just the singular one.

Overall, a novel technical approach for multi-dimensional clustering is shown to deliver low algorithmic complexity enabling communication tracing at extreme scale in an unprecedented manner.

3.1 Call-Path Clustering

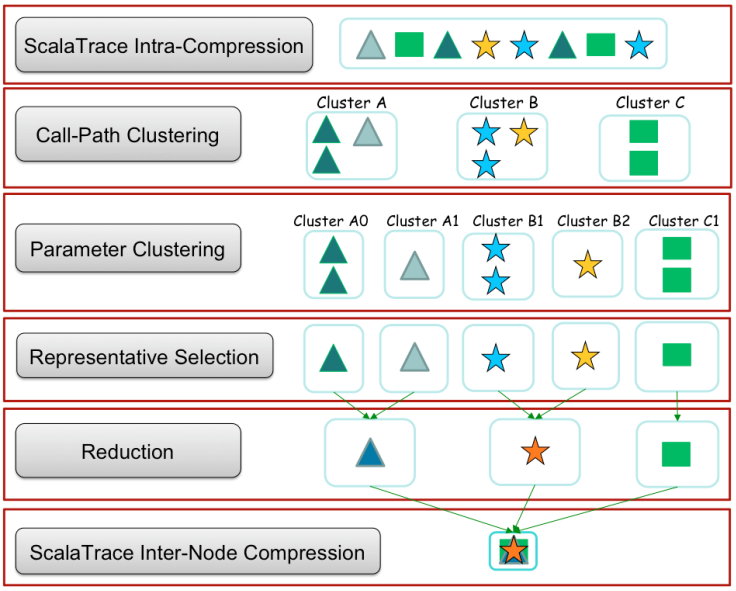


Figure 3.1: Overview of Proposed Clustering Algorithm

Table 3.1: Components of Parameter Signature

Component Descriptions	**Bit Positions**
Average **COUNT** sent or received for MPI events	0-15
DEST: Average of the relative address of destinations of MPI events	16-31
SOURCE: Average of the relative address of sources of MPI events	32-47
MPI Data Types: such as 48:MPI_CHAR, 49:MPI_INTEGER, etc.	48-54
MPI Operation Types: such as 55:MPI_MAX, 56:MPI_MIN, etc.	55-61
MPI Communicator Type: such as 55:MPI_COMM_SELF, etc.	62-63

Figure 3.1 illustrates that Call-path+Parameter clustering has different phases. During the first phase, the algorithm clusters processes with different sequences of MPI calls, which creates so-called "main clusters".

A stack signature consists of a number of backtrace addresses of the program counters (return addresses), one for each stack frame. Our Call-path signature is a 64-bit signature. To represent large stack signatures as 64 bits, we computed the exclusive or (XOR) of each part with the current 64-bit signature value.

After creating the 64-bit version of stack signatures, in order to create the all-path signature, we compute the XOR of all 64-bit stack signatures. In most benchmarks, capturing the calling context is sufficient for distinguishing MPI events from each other. However, the Multi-Grid (MG) benchmark from the NAS Parallel Benchmark (NPB) suite features a case where two processes with same number of events and similar calling contexts experience different orders among their events. Therefore, to capture not only the calling context but also the order of 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.

Notice that any low-overhead hash function can result in conflicts, e.g., when recursion or function pointers are used. However, this is not the case for conditionals resulting in alternating call sequences like a()→b() and b()→a() since the initial calls would originate from different caller program counters (of then/else branch). Most HPC codes neither utilize function pointers nor recursion, in part due to caller overhead and inferior compiler optimization that results from such codes (due to pointer aliasing and data dependence analysis for parallelization, among other problems). Furthermore, the number of different call paths for the set of HPC benchmarks codes that we tested is limited to a small constant (< 10). We observe disjoint call paths for all tested HPC codes with our hash technique in practice so that more expensive hashes are not needed (but we could alert users to collisions or even automatically trigger more complex hashes if needed).

Fig. 3.1 provides a simple illustration of Call-path clustering, where processes of different shapes are grouped into different clusters. This operation occurs on a radix tree, i.e., each node receives the call-path signatures of its children. Then, it compares its own call-path signature with those of its children. Finally, it sends different signatures and corresponding ranklists to its parent. At the top of the tree, node 0 receives all of the different signatures and their ranklists.

Node 0 broadcasts the overall clustering result, so all nodes are informed of their respective cluster membership. In our implementation, we considered the start rank of each cluster ranklist the head of the cluster. The computational cost of these two operations is $\mathrm{O}(\log P)$, where P denotes the number of processes.

During the second phase, our algorithm applies parameter clustering. We use a different signature called the parameter signature, which, similar to the call-path signature, is 64 bits

long. This signature is composed of the parameters of the MPI event, such as its count, type, source, destination, etc., see Table 3.1. Note that we did not include the TAG parameter in the parameter signature. While we could easily add the TAG parameter to the signature, we found few differences in the call-path signatures and observed that SRC/DEST parameters could capture the TAG differences in practice for our benchmark set.

3.2 Parameter Clustering

Parameter clustering is the second phase of the proposed algorithm. Similar to the first phase, this phase was implemented over a radix tree. The main difference was that each cluster had similar operations on parameter signatures over a radix tree of its own members. At the end of this phase, the head of the clusters identified in phase one know all of the different parameter signatures in their own "territory" (cluster). Therefore, with the help of parameter clustering, we were able to distinguish processes with the same execution structure but different parameters. Fig. 3.1 illustrates parameter clustering symbolically, where processes with the same shape but different colors are grouped into different clusters. The computational cost of our clustering algorithm at this phase was also $O(\log P)$.

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 Creating a Complete Trace

The next phase consists of selecting a head of each cluster as the lead rank. We choose the start rank from each different sub-cluster. The reason the algorithms chooses the start rank is after clustering, nodes in each cluster are all identical in terms of parameters (same Call-path and same parameter signatures). Therefore, considering the collected signatures, selecting the first one or any other process will provide the same results. Unlike traditional clustering, which is a top-down process, creating the full trace is a bottom-up process. All similar processes are grouped together after Call-path+Parameter clustering, and each lead process updates the ranklists accordingly to include the members of its own sub-cluster. For instance, if the cluster contains nodes {0,1,2,3,4} and node 4 is selected as the cluster lead, all 1D ranklists in trace 4 convert *<1 4 1 0>* to *<1 0 5 1>*. This change covers all other members of the clusters.

After updating the ranklist, there are different approaches to create the global trace file (also see Table 3.2):
1) *Default version*: the lead traces are merged within each main cluster. Sub-clusters with different parameters, such as A1 and A2, are merged pairwise linearly at a node within a radix tree (facilitating relative encoding matches [41]) so that the overall reduction over the tree is

logarithmic in complexity. For instance, at the reduction phase in Fig. 3.1, two triangles with different colors are merged into a single triangle. The cost of these two operations is $O(n \log P)$, where n denotes the size of the PRSD-compressed intra-node event trace (typically a constant) and P is the number of processes.

The inter-compression reduction of ScalaTrace [41] at each node in the radix tree is a costly operation with $O(n^2)$ complexity, where n is the size of the PRSD-compressed intra-node event trace. When using ScalaTrace without clustering, all processes participate in this operation over a radix tree. The cost of operation is $O(n^2 \log P)$. With the clustering algorithm, on the other hand, only a set of lead processes with different call-path signatures have to participate in this operation. During the last phase of Fig. 3.1, three different shapes are merged.

2) *Function version:* Because some benchmarks have unique parameters per process, the number of sub-clusters could increase linearly with the number of processes. This could result in scalability limitations. To tackle this problem, we devised two strategies: 1) *User Plug-in Functions*: At the level of building a signature, the system only considers non-unique parameters and filters out certain MPI parameters indicated by the user. It then continues clustering based on the created signatures. At replay time, upon encountering MPI events with unique parameters, it leverages user plug-ins to calculate and handle the unique value of MPI parameters correctly. 2) *Filtering Function*: During parameter clustering, the algorithm keeps track of changes of parameters. At the end of clustering, it knows which parameters contributed significantly to the creation of new clusters. Instead of sending its entire trace file to the parent (in an effort to linearly merge it), each lead process sends only different parameters and filters out value-identical parameters. Details of our implementation are provided in Section 3.6.

Table 3.2: Trace Creation

Trace Creation Method	Suitable Benchmarks
Default Version	LU, MG, SP, BT, S3D, IS
User Plug-in Functions	CG, FT
Filtering Function	POP, CG, FT

As previously mentioned, the cost of the clustering algorithm is $O(\log P)$, the cost of the first level of merging is O(n $\log SC$), where SC is the maximum number of sub-clusters within a main cluster, and the cost of the second level is $O(n^2 \log MC)$, where MC is the number of different call-path signatures or main clusters.

Due to the nature of parallel programs, as we expected and observed in most of the parallel benchmarks, the number of processes with different execution structures is very small. Since the

set of different call-path signatures is so small (mostly just a constant), the clustering algorithm reduces the computation time significantly.

Given the space complexity, the best scenario would be to capture application behavior in only one cluster, meaning there is only one execution sequence / parameter set. In this case, at the root node, there will be one signature and one ranklist containing all the node ranks. The exact size will be eight bytes for the signature and ten bytes, or five integer values, for the ranklist.

In the worst case scenario in which each program has its own unique behavior, processes at different levels of the tree have different complexities. At the bottom of the tree, each leaf node has one ranklist and one signature. On the other hand, the root node has P ranklists and P signatures.

3.4 Reference Clustering

A legitimate concern after introducing Call-path+Parameter clustering is to ensure that the proposed clustering algorithm does not lose important information. As noted previously, to evaluate the accuracy and scalability of the algorithm, we create a reference clustering approach that uses a reference signature. The reference signature is a sequence of events, covers call-path signatures by adding a sequence number to each MPI event, and features parameter clustering by keeping each MPI event's parameters uncompressed. The computational complexity of this clustering is $O(n \times m \times s)$, where n is the number of events (i.e., *not* just the size of the PRSD-compressed intra-node event trace), m is the number of disjoint events' parameters and s is the number of disjoint reference signatures. The space complexity is a function of the total number of events.

In Section 3.6, we provide the results of the experiments conducted on different benchmarks to compare the results of space complexity for the multi-level call path+parameter clustering approach and the reference signature.

3.5 Experimental Setup

We utilized a state-of-the-art cluster at our exposure to conduct experiments. All machines were 2-way SMPs with AMD Opteron 6128 processors with 8 cores per socket. Each node is connected by QDR InfiniBand. This is the largest platform we were able to obtain access to at this time. We tested call-path+parameter clustering, reference clustering and no clustering, which is the default version of ScalaTrace for the *NAS* Parallel Benchmarks (NPB) and Sweep3D, the Parallel Ocean Program (POP) Each experiment was run five times, and the average value and standard deviation were 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 C input size [7] and Sweep3D [29]. 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. POP [24] is an ocean circulation model developed at Los Alamos National Laboratory. Our experiments exercise a single degree grid resolution in which the problem size is varied based on number of processes, and the individual block size is 16×16.

3.6 Results and Analysis

As previously noted, ScalaTrace's inter-node compression is a 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. To remove this effective bottleneck, we applied our logarithmic algorithm to find processes that exhibit different behavior. Also, we divided the merge process into two steps: (1) merging sub-clusters into main clusters over a local radix tree with $O(n \log SC)$ complexity, where SC is the maximum number of sub-clusters within a main cluster, and (2) merging main clusters over a radix tree with $O(n^2 \log MC)$ complexity, where MC is the number of main clusters. The second level of merging is the most costly operation. Therefore, our first experiment was to determine MC for different benchmarks.

Fig. 3.2 depicts the topologies of different benchmarks at size 16 (processes). In this figure, main clusters are separated by solid lines, and sub-clusters are separated by dotted lines (e.g., BT has one main cluster and three sub-clusters). Table 3.3 shows the number of main clusters MC and sub-clusters SC for these benchmarks. According to our experiments, for both weak and strong scaling, the reported number of clusters is constant. Also, the number of clusters is constant for the Sweep3D benchmark with different problem sizes. Notice that the total number of clusters is given by $max(MC, SC)$, which indicates how many different traces ultimately have to be collected for communication characterization.

Fig. 3.2 and Table 3.3 indicate the following:

(1) Integer Sort (IS) has three main clusters and no sub-clusters. These three groups of processes display very similar execution behavior, except when each process sends its largest key value to the next process. In this phase of the code, process *zero* does not receive any value, and process $comm\_size - 1$ does not send any value.

(2) The Block Tri-diagonal solver (BT) and the Scalar Penta-diagonal solver (SP) each have only one main cluster, meaning that all processes have the same sequence of MPI events. However, parameter clustering captures three sub-clusters with different communication patterns. Another issue is the COUNT value, which could differ slightly for some events of processes

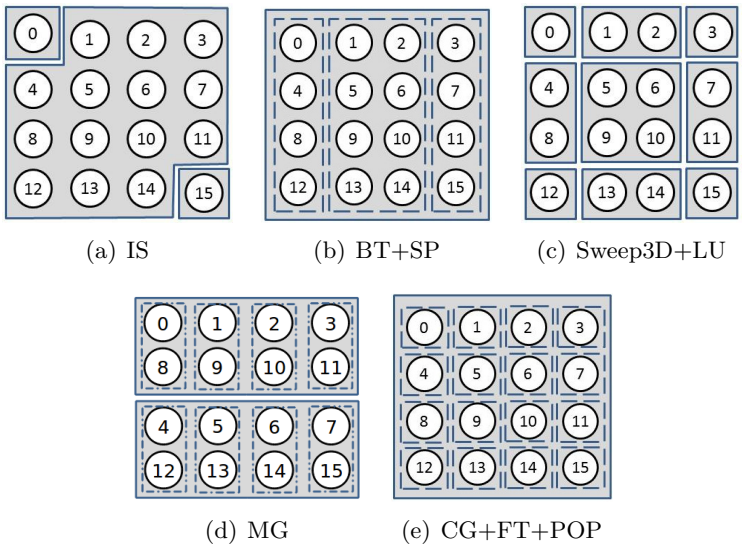


Figure 3.2: Topology of Different Benchmarks for 16 Processes Through Call-path+Parameter Clustering

Table 3.3: Number of Main and Sub Clusters

Benchmarks	Sweep3D	BT	CG	IS	LU	SP	FT	MG	POP
# of Main Clusters MC	9	1	1	3	9	1	1	2	1
# of Subclusters SC	1	3	4	1	1	3	1	8	16

Sweep3D: Problem size: 100×100×1000, # processes: **any** valid one
BT,IS,SP,FT: Class: **any**, # processes: **any** valid one
MG,CG: Class: **any**, # processes: 16
POP: Problem size: 512×512 blocks, Individual block size: 16×16, # processes: 16

with the same communication pattern (e.g., 9526 and 9500). To compensate for such negligible differences, we implemented a filter that considers two COUNT values to be similar if they differ by only a small percentage (threshold-based filtering), and we record their average. The difference threshold in our experiments is 5%.

(3) The *Sweep3D* neutron-transport kernel and the Lower-Upper Gauss-Seidel solver (*LU*) have nine main clusters and no sub-clusters, meaning that processes within the same main cluster display the same communication pattern. Sweep3D is a stencil code in which each process must wait for boundary information from neighboring processes to the north and west before computing values within its subdomain [22]. Similar to Sweep3D, *LU* is also a stencil code [46] that creates nine different main clusters.

(4) The number of main Multi-Grid (MG) clusters is not constant; as shown in Fig. 3.2, for 16 processes, there are two clusters, and this number increases sublinearly (e.g., (P=32, MC=4), (P=64, MC=8), (P=256, MC=16), (P=1024, MC=36), etc.) while $SC = 4 \times MC$ for this benchmark. MG is a simplified multigrid kernel that solves 3D Poisson equations. This code requires 2^n processes, where n is an integer number. The partitioning of the grid into processes occurs such that the grid is repeatedly halved along the Z, Y, and X dimensions, respectively [7]. This behavior is due to the following two main reasons [15]: (i) The number of processes assigned to each grid depends on the problem size and the total number of processes P. MG might reduce the number of processors assigned to compute on a coarser grid in order to increase the computation-to-communication ratio. Therefore, some processes may participate in more MPI events; (ii) Two types of communication occur in MG: a boundary exchange and an inter-processor extrapolation/interpolation between two adjacent grid levels. Because MG changes the grid resolution at each iteration of the algorithm, these boundaries change. As the algorithm moves from coarser to finer, more boundaries are created.

(5) Conjugate Gradient (CG), Fast Fourier Transform (FT) and Parallel Ocean Program (POP) each only have one main cluster, meaning that there is only one execution structure. However, many sub-clusters exist within the main cluster. In CG and POP, each process has its own unique communication pattern. FT have one main cluster and several sub-clusters.

It is beneficial to our approach that these benchmarks only have one main cluster, as this reduces the computational complexity from $O(n^2)$ to $O(n)$. To further reduce the cost of linear compression at the parameter clustering level, one solution is to forcibly "merge" events with different parameters. For example, for CG, the parameter signature indicates that events differ in $SOURCE$ and $DEST$; therefore, all events with $SOURCE$ or $DEST$ may be merged, while other parameters are preserved. This may still result in a large numbers of clusters.

The alternative is for users to supply a plug-in function capturing unique parameters that otherwise would increase the total number of clusters because they can (at best) be merged forcibly. For instance, Fig. 3.4 shows a CG communication matrix as a heat map for 64 processes, where the x- and y-axes denote mutual communication end-points, and the communication intensity is depicted within a color range (cold/blue=low to hot/red/yellow=high). The orange points (close to the diagonal) in this figure indicate communication occurring with a high frequency. The clustering algorithm can capture the iterative behavior of the orange points easily. However, even though we are using relative addresses for $SOURCE$ and $DEST$, the blue points (further from the diagonal) indicate infrequent communication unique to each process.

To capture this secondary communication pattern while simultaneously reducing the number of sub-clusters, we can use the following formula :

ALGORITHM 2: User-provided Plug-in Function for the CG Code

```
1 if npcols.eq.nprows then
2 |   exch_proc = mod(me, nprows) * nprows + me/nprows
3 end
4 else
5 |   exch_proc = 2 * (mod(me/2, nprows) * nprows + me/2/nprows) + mod(me, 2)
6 end
```

Here, $npcols$ denotes the number of processes per column, and $nprows$ is the number of processes per row. In CG, the total number of processes equals the number of processes per row times the number of processes per column. If the total number of processes is not a square, then the number of processes per column is twice that of the number of processes per row. $exch\_proc$ is the transpose exchange process, and me is the process rank. The information in Table 3.4 indicates that once this function is supplied, the number of sub-clusters decreases significantly.

FT solves a three-dimensional partial differential equation (PDE) using fast Fourier transform (FFT). Because all of the processes have the same sequence of events, there is only one main cluster. However, two parameters, $COLOR$ and KEY used in two MPI_Comm_Split events, have different values for different processes. Similar to CG, we can use the following formula:

ALGORITHM 3: User-provided Plug-in Function for the FT Code

```
1  if np.eq.1 then
2  |   np2 = 1
3  end
4  else if np.le.nz then
5  |   np2 = np
6  end
7  else
8  |   np2 = np/nz
9  end
10 me1 = me/np2
11 me2 = (me%np2)
```

Here, me is the process rank, $me1$ and $me2$ are process coordinates, np is the number of processes and nz is the size of the z dimension. Furthermore, $me1$ and $me2$ are assigned to

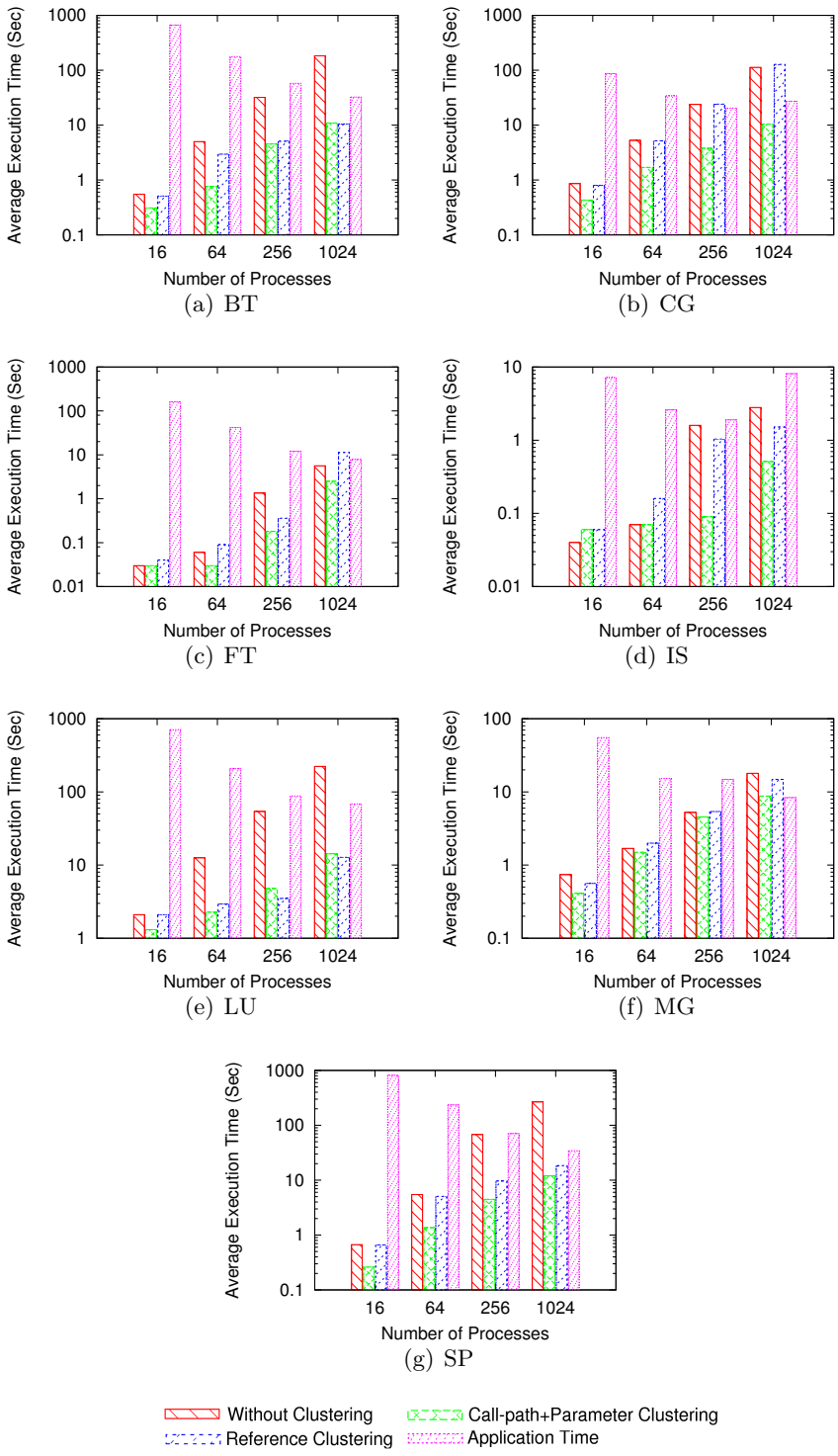


Figure 3.3: Execution Times for Inter-Node Compression Variants and Entire NAS Benchmarks(Strong Scaling) — Nodes/Tasks=1/16

KEY and *COLOR* in one call and vice versa in another call to *MPI_Comm_Split*. We also kept track of the global state to assign these values correctly.

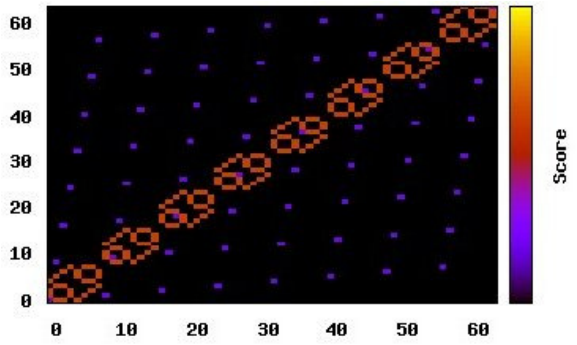


Figure 3.4: CG Communication Matrix

Table 3.4: Number of Clusters for CG

Num. of Processes	16	64	256	1024
Number of Main Clusters	1	1	1	1
Number of Subclusters	4	8	16	64

POP achieves parallelism through domain decomposition, similar to other climate or fluid dynamics models. It divides the input domain into pieces that each node can compute relatively independently. To increase this independence, ghost cells are defined so that a node will have some information about the state variables of neighboring nodes [48].

Even though Call-path+Parameter clustering performs well with user-provided plug-in functions for benchmarks such as CG and FT, we found it difficult for users to provide such functions for complex benchmarks such as POP. As previously noted, for CG, FT and POP, the number of sub-clusters increases linearly with the number of processes. However, we also observed that the number of call-path clusters is very small. Basically, such traces mostly differ in terms of *parameters*, not *call-path*. Instead of sending the complete trace files, we therefore send only those events with different parameters to create the global trace within each main cluster. We refer to this as Call-path+Parameter Clustering with Filtering.

When the number of parameter clusters exceeds the threshold (where the the threshold is set to $P/2$ here), our implementation of Call-path+Parameter Clustering switches dynamically to sending only different parameters within main clusters.

In Subsection 3.6.2, not only did we test this implementation on POP, but we also tested Call-path+Parameter clustering both with user plug-ins and filtering functions for CG and FT.

The next two subsections present the results of our algorithm under both strong and weak scaling application execution scenarios.

3.6.1 Strong Scaling

Under strong scaling, the number of processes is increased under the same program input. We tested our clustering algorithm on the NAS benchmarks under strong scaling. Fig. 3.3 depicts four bars per configuration: (1) the execution cost for the NAS benchmarks during the inter-node compression step for Call-path+Parameter clustering, (2) reference clustering, (3) without clustering and (4) application execution time with instrumentation. The x-axis of the graph denotes the number of processes participating in inter-node compression. The y-axis is the execution time in seconds shown on a logarithmic scale. The execution cost without clustering refers to regular inter-node reduction/compression within ScalaTrace V2.

As the figure shows, Call-path+Parameter clustering has orders of magnitude smaller cost than without clustering. For all benchmarks, the cost of call-path clustering is less than 50% of total program execution time — in contrast to the original inter-node compression without clustering of ScalaTrace, which sometimes exceeds the application runtime for larger number of processes. Notice that these benchmark runtimes are relatively short (seconds) while large-scale applications generally run for hours but experience similar inter-node compression costs as these benchmarks. Call-path+Parameter clustering also has orders of magnitude smaller execution cost than reference clustering for most benchmarks. This is due to the number of processes involved in inter-node compression, as depicted in Table 3.5 for P=256. For MG, Call-path+Parameter clustering and reference clustering have almost the same number of parameters. Nonetheless, the cost is smaller than that of reference clustering because most of the clusters in Call-path+Parameter clustering are sub-clusters. For some Ps, such as P=256 for BT and LU or P=1024 for SP, the call-path+Parameter clustering cost is very close to the reference signature because, after clustering, the number of processes involved in inter-node compression is in the same order of magnitude. However, at the end of this section, we show that Call-path+Parameter clustering performs significantly better than reference clustering in terms of space complexity, including but not limited to these configurations. Notice that that application time of IS is lower at P=256 than at P=1024 indicating that there is not enough work per node left at the latter, i.e., it has hit its limit under strong scaling.

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

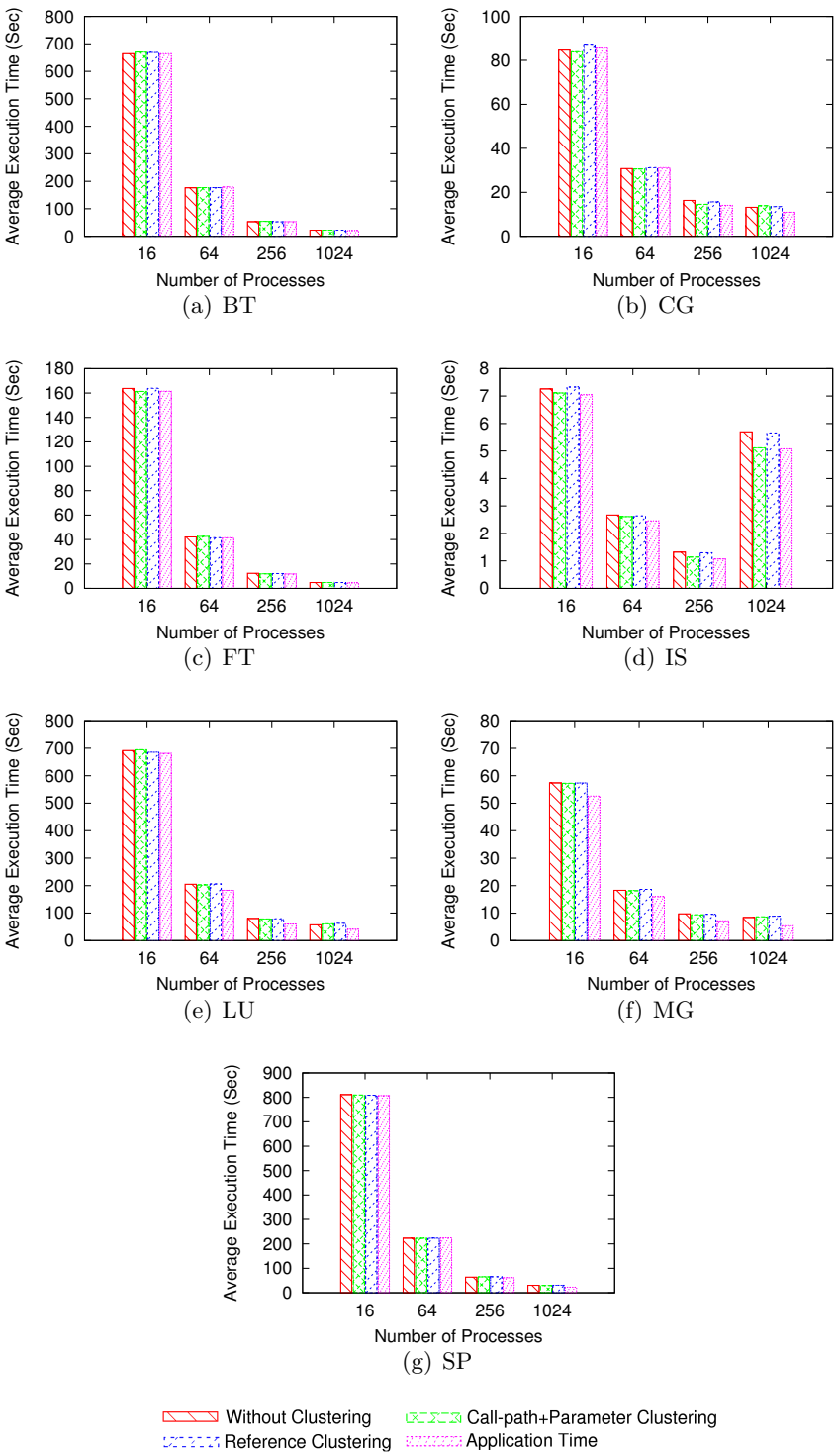


Figure 3.5: Replay Times (Strong Scaling) — Nodes/Tasks=1/16

Table 3.5: # Processes Involved in Inter-Node Compression for Clustering Approaches, P=256

Pgm	Call-path+Param Cl.	Ref. Cl.	w/o Clustering
BT	3	41	256
CG	16	256	256
FT	1	256	256
IS	3	21	256
LU	9	16	256
MG	64	72	256
SP	3	53	256

accordingly, and simulates computation time as sleeps [58]. 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 node 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. Conversely, the error rate is $1 - ACC$.

Fig. 3.5 depicts the overall trace-file replay time, depicted in seconds on a linear y-axis (1) without, (2) with Call-path+Parameter, (3) with reference clustering and (4) of the non-instrumented original application. The x-axis of these graphs denotes the number of processes participating in the inter-node compression phase for the three different methodologies. Replay under Call-path+Parameter clustering is 88% accurate relative to application runtime over all benchmarks and configurations, which is the same accuracy we observe without clustering, where higher accuracy is observed for longer-running experiments (more lead processes) than for shorter running ones (an artifact of strong scaling). This equally applies to call-path+Parameter clustering with *user-provided functions* (CG+FT) and without (all others) showing that replaying with user-provided specification poses no problems.

3.6.2 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.) Due to input

Table 3.6: Number of Processes Involved in Inter-Node Compression — Weak Scaling

# Processes	16	64	256	1024
BT Prob. Size	60^3	101^3	160^3	255^3
BT # Clusters	3	3	3	3
FT Prob. Size	512×256^2	512^3	$1024^2 \times 512$	2048×1024^2
FT Clusters	1	1	1	1
LU Prob. Size	64^3	128^3	256^3	512^3
LU # Clusters	9	9	9	9
POP Prob. Size	512^2	768^2	896^2	1024^2
POP # Clusters	1	3	2	2
Sweep 3D Problem Size Per Process			$100^2 \times 1000$	
Sw3D # Clus.	9	9	9	9

constraints / lack of weak scaling inputs, we only report these results for the benchmarks for which weak scaling inputs are available natively through the benchmark or when available from other work [57].

As Table 3.6 indicates, weak scaling and strong scaling produce an equal number of clusters for NAS BT, LU, FT. The first row of each table indicates the number of processes (MPI tasks); the second one the overall problem size for BT, FT and LU. This table presents the number of main clusters for POP under weak scaling. The number of main clusters, though very small, varied for different problem sizes, mainly because of ghost cells. For Sweep3D, it indicates the per process size; and the last one the number of clusters. We observe the number of clusters for both types of scaling have the same cardinality and identical member sets.

The execution times in seconds on a logarithmic scale on the y-axis of BT, LU, FT, Sweep3D and POP are reported in Fig. 3.6 for different numbers of processors (x-axis). Just as seen for strong scaling, Call-path+Parameter clustering has orders of magnitude shorter execution time than without clustering under weak scaling as well. While Call-path+Parameter and reference result in similar cost for their cluster formation during tracing, we later show that the former outperforms the latter significantly in terms of space complexity. For POP, the execution cost of reference clustering compared to without clustering increases very fast, mainly because of the large number of MPI events (approximately 1500), which creates a large reference signature for each process.

Fig. 3.7 depicts the replay times in seconds on a linear scale (y-axis) for different number of processors (x-axis). In analogy to strong scaling, it illustrates that the overall trace-file replay time under Call-path+Parameter clustering is 93% relative to application runtime over all benchmarks and configurations except for POP, the same as without clustering. The standard deviation of replay traces is always less than 2 except for Sweep3D, where it is 12, which is still small compared to its large execution time of $\approx$ 1500 seconds.

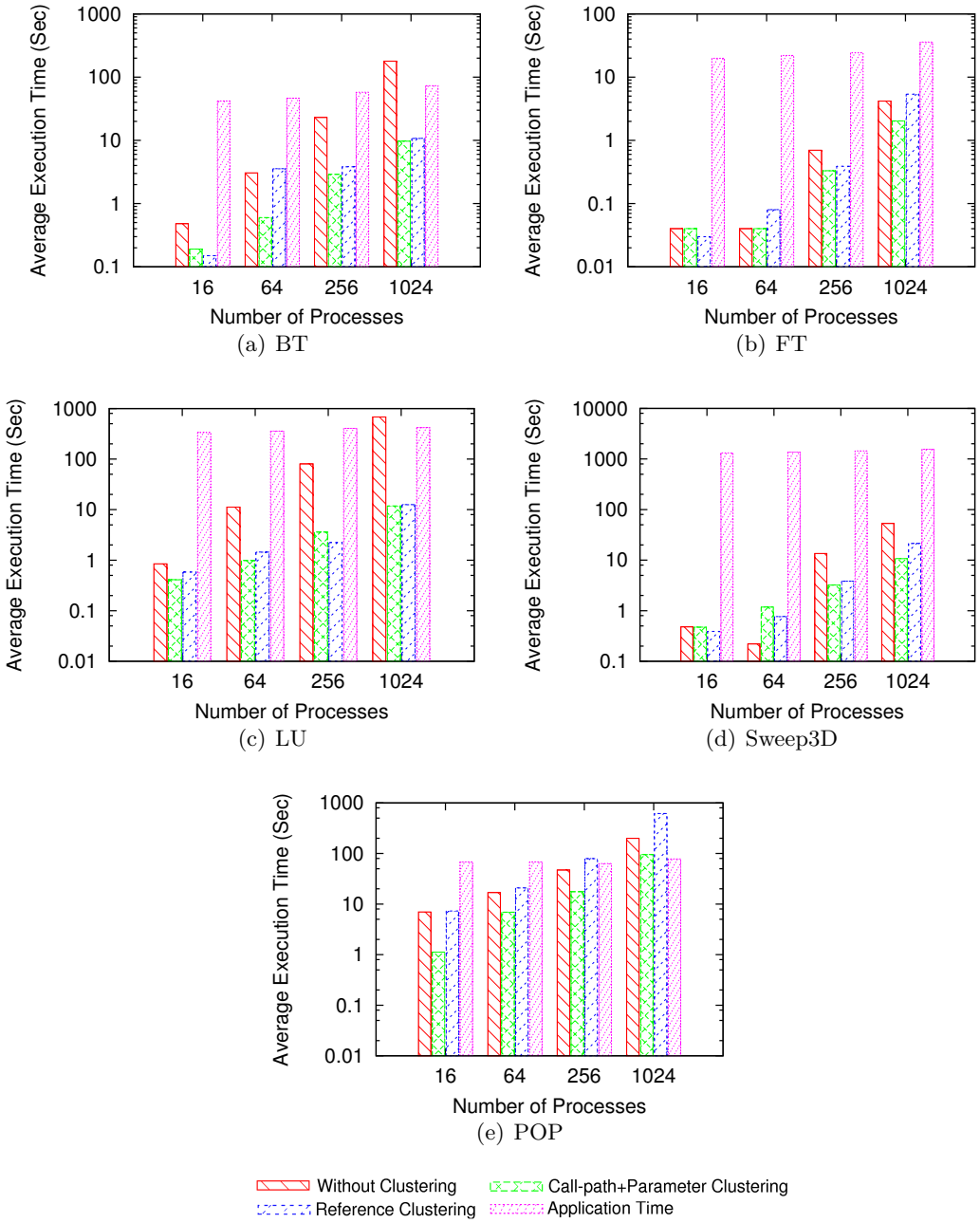


Figure 3.6: Execution Times for Inter-Node Compression Variants and Entire Application (Weak Scaling) — Nodes/Tasks=1/16

The execution times of POP include a filtering function for Call-Path+Parameter clustering. For POP, notice the difference between replay and application time. Yet, replay times with and without clustering are very close. We analyzed this behavior and found that it is due to ghost cell updates in each timestep (e.g., ghost cell updates for forcing terms leading into the barotropic solver). These ghost cells update calls originate from different locations in the program, which creates new stack signatures. Even though they are similar sequences of calls, intra-node compression is ineffective due to a stack signatures mismatch (even for the same MPI calls at the leaf of the call chain). We modified the POP trace without clustering to compressed these MPI calls. Consequently, the replay time becomes very close to the application time over 80%, i.e., replay inaccuracy is reduced significantly. This indicates that we can further improve our intra-node clustering by allowing interim call stack frames to differ as long as higher (close to the top) and lower (close to the leaf) frames match. This is subject to future work.

Fig. 3.8 depicts the execution times of CG and FT both with plug-in and filtering functions of the proposed clustering algorithm. As shown, these two cases had similar runtime costs.

3.6.3 Cost of Clustering

To separately show the small cost of clustering and its high accuracy, we conducted experiments with Sweep3D (Input size: 100*100*1000) for the following three different cases:
1) inter-node compression for only the nodes representing clusters (after clustering in a separate second run);
2) inter-node compression with Call-path+Parameter clustering; and
3) Inter-Node Compression w/o Clustering, i.e., ScalaTrace Original version (# of clusters = P). Fig. 3.9(a) shows the clustering cost during inter-node compression in the first and second column. The difference in costs is small ($< 11\%$). Also, inter-node compression without clustering has a much higher cost.

Fig. 3.9(b) illustrates that replays are nearly identical irrespective of which method is used, i.e., there is no noticeable perturbation/imbalance effect. Relative accuracy of replay is high.

To assess the cost of clustering vs. that of inter-node compression (i.e., merging lead traces), we conducted another experiment. Fig. 3.10 presents the cost of call-path clustering vs. inter-node compression step. For S3D, LU, MG, and POP benchmarks that they have a larger number of clusters, we observe that inter-node compression cost increases. This is because more clusters are involved in the inter-node compression step, and the sizes of traces slightly increases. As expect, by increasing the number of processes, the cost for clustering also increases. For BT, SP, CG, FT and IS, the inter-node compression cost is significantly less than that of other benchmarks. This is because the number of clusters is constant for these benchmarks. The clustering cost still differs between benchmarks since each of them have a unique number of events and number

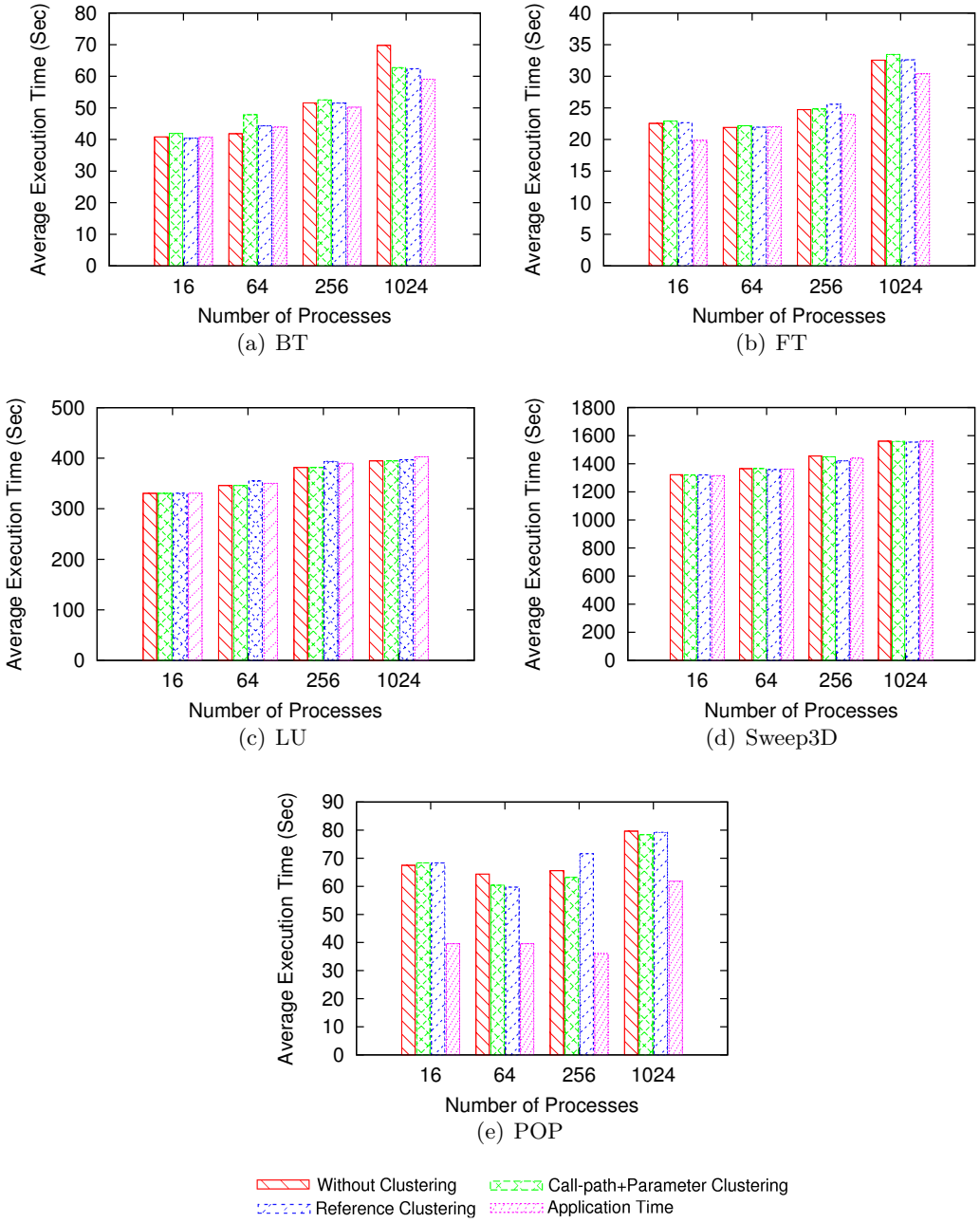
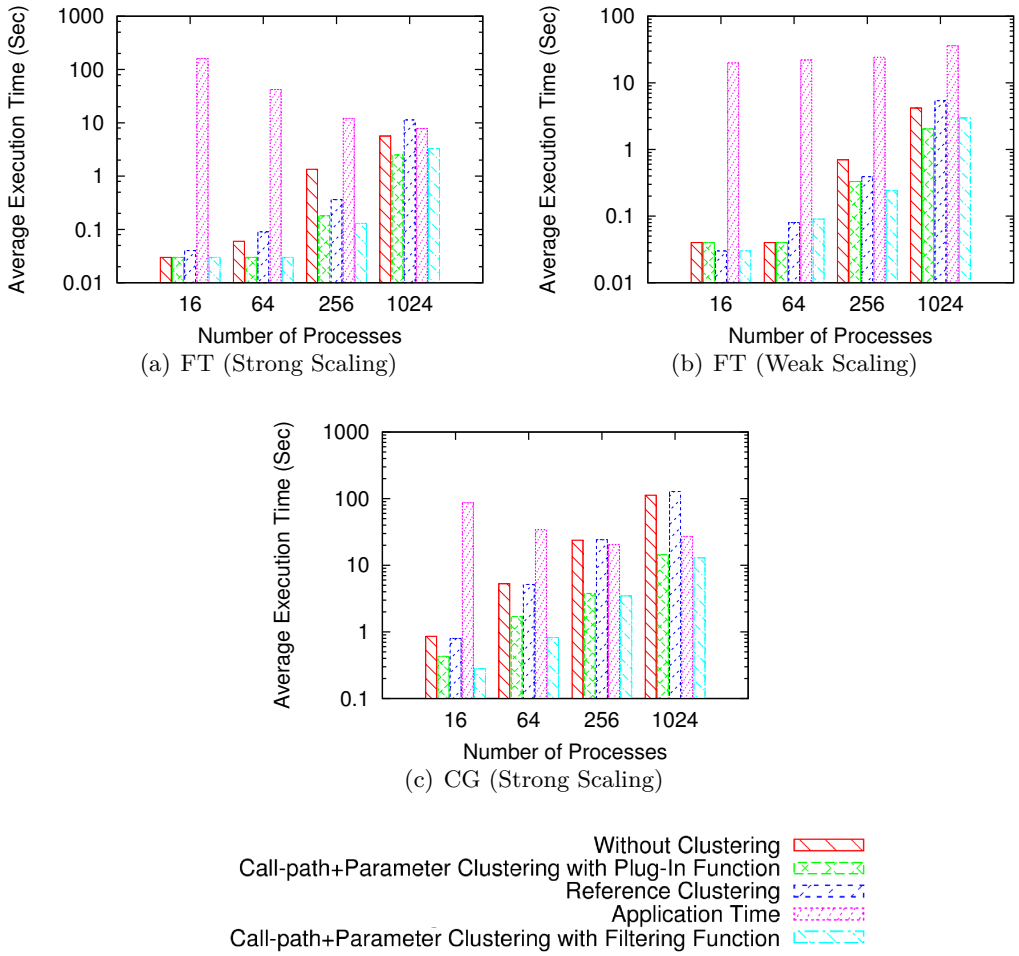


Figure 3.7: Replay Times (Weak Scaling) — Nodes/Tasks=1/16

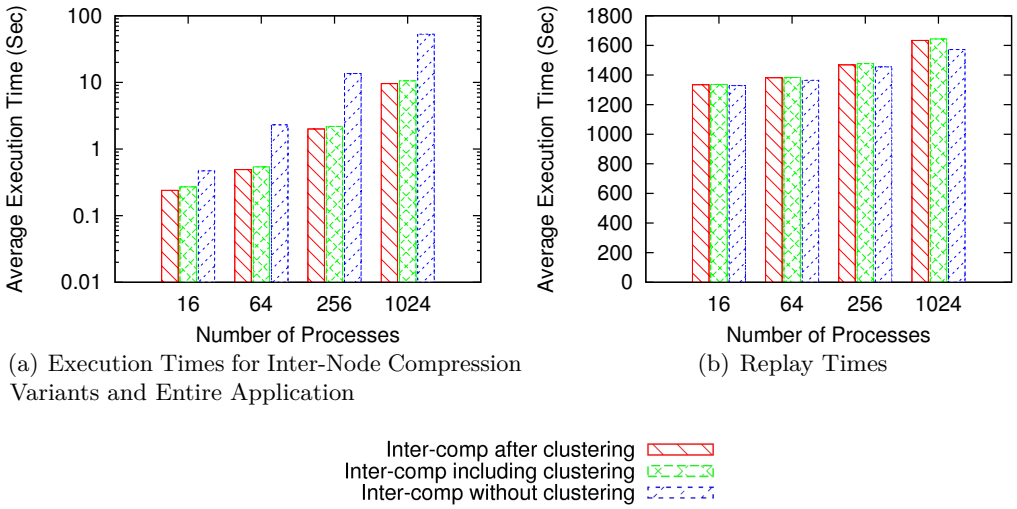


(a) FT (Strong Scaling)

(b) FT (Weak Scaling)

(c) CG (Strong Scaling)

Figure 3.8: Execution Times for Inter-Node Compression Variants and Entire Application (Weak Scaling) — Nodes/Tasks=1/16



(a) Execution Times for Inter-Node Compression Variants and Entire Application

(b) Replay Times

Figure 3.9: Sweep3D Execution and Replay Times (Weak Scaling) — Nodes/Tasks=1/16

Table 3.7: Average Space Complexity Per Process — P=256

		Call-Path+Param Cluster.			Ref. Clustering		W/o Clustering	
pgm	avg trace size	MC	SC	avg Space	# clusters	avg space	# clusters	avg space
BT	72KB	1	3	0.08 KB	41	108.49 KB	256	71.71 KB
CG	44KB	1	16	0.36 KB	256	376.32 KB	256	43.82 KB
FT	8KB	1	1	0.06 KB	256	70.46 KB	256	7.96 KB
IS	8KB	3	1	0.15 KB	21	3.62 KB	256	7.96 KB
LU	72KB	9	1	2.43 KB	16	25.05 KB	256	71.71 KB
MG	216KB	16	64	14.23 KB	72	733.83 KB	256	215.15 KB
SP	68KB	1	3	0.10 KB	53	133.06 KB	256	67.73 KB
Sweep3D	28KB	9	1	1.06 KB	9	4.86 KB	256	27.89 KB
POP	1.1MB	1	256	16 KB	256	1.1 MB	256	1.09 MB

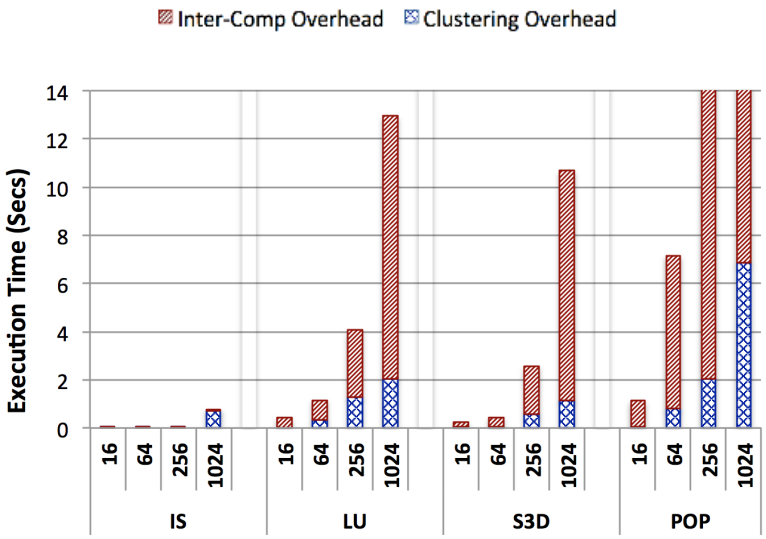
of parameters subject to parameter signatures creation. Note, the inter-node compression cost of POP was around 90 seconds, but we cut the y-axis at 14 seconds for readability.

3.6.4 Space Complexity

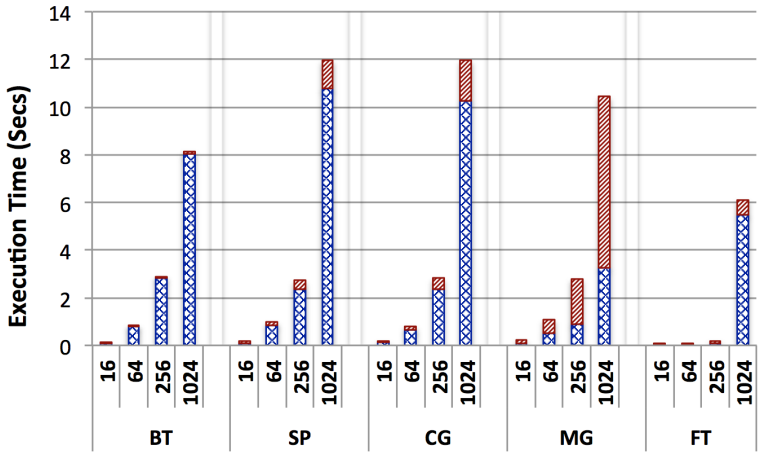
The objective of the last experiment is to assess the space complexity. We calculated the number of bytes required for the different clustering methods. Table 3.7 shows the space complexity of all benchmarks for P=256.

Average space per process for without clustering is calculated as follows:

$$AvgSpaceperProcess_{no\ cluster} = \frac{AvgTraceSize * (P-1)}{P}$$



(b) Cost of Clustering vs. Inter-Node Compression



(c) Cost of Clustering vs. Inter-Node Compression

Figure 3.10: Clustering Cost vs. Inter-Node Compression Cost (Strong Scaling for all except LU and Sweep3D) — Nodes/Tasks=1/16

Here, all processes send their trace files to their parents over a radix tree, except for the root process itself. For reference clustering, the average space is as follows:

$$P1 = AvgTraceSize \times (C - 1)$$
$$P2 = P \times AvgSignatureSize$$
$$P3 = AvgSignatureSize \times C \times (P - 1)$$
$$AvgSpaceperProcess_{ref\ cluster} = \frac{P1 + P2 + P3}{P}$$

where C is the number of clusters, $P2$ and $P3$ denote the space of clustering, and $P1$ is the space of inter-node compression. Finally, for Call-path+Parameter clustering, we have

$$P4 = AvgTraceSize \times (MC - 1)$$
$$P5 = AvgSignatureSize \times (MC + SC) \times (P - 1)$$
$$AvgSpaceperProcess_{call-path+param} = \frac{P2 + P4 + P5}{P}$$

where MC is the number of main clusters, SC is the number of sub-clusters, $P2$ and $P5$ denote the space of clustering, and $P4$ denotes the space of inter-node compression.

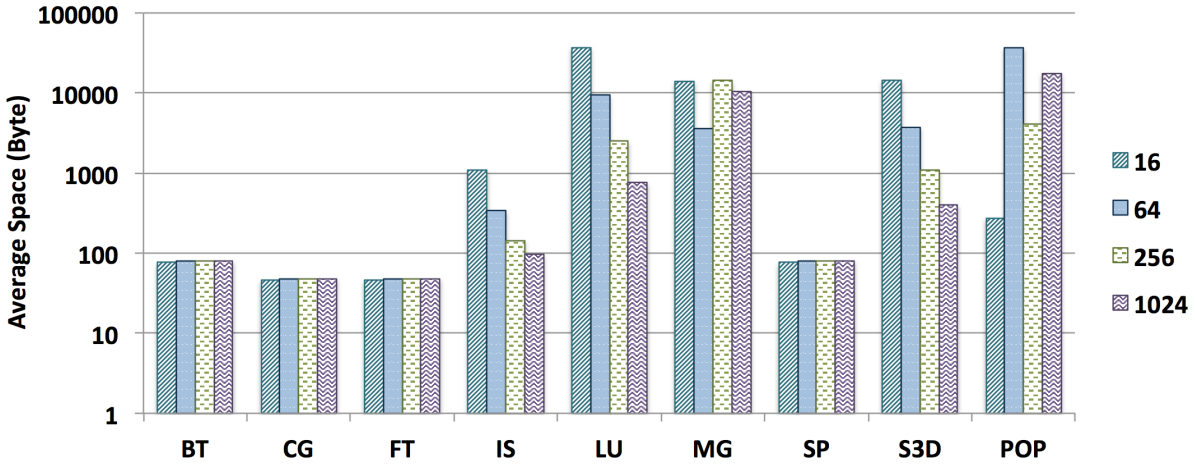


Figure 3.11: Space Complexity — Strong Scaling, except for Sweep3D and POP (Weak Scaling)

Table 3.7 depicts trace sizes and space metrics for the three clustering types with 256 processes. We observe that reference clustering generally increases the average space per process over no clustering by a factor of 1.4-10 depending on the benchmark — except for Sweep3D, IS and LU, which is due to the small number of clusters involved in inter-node compression for those three benchmarks. Call-path+Parameter reduces average space per process by 2-3 orders of magnitudes to 0.1-6% of that without clustering depending on the benchmark. The small size of the signatures and the small number of processes involved in inter-node compression account for this difference. Reference clustering generally significantly increases the average space per process over call-path+Parameter clustering by up to three orders of magnitude, i.e., more specifically a factor of 4.5-1356 depending on the benchmark. The execution cost for both is comparable because it is a function of the number of clusters, and both clustering methods have a similar number of clusters. However, the cost of Call-path+Parameter is often lower than for reference clustering since $MC + SC$ tends to be lower than C in P3 and P5, respectively, as well as due to more effective multi-level clustering optimizations, including plugins.

Figure 3.11 shows the space complexity for Call-path clustering for different numbers of processes (16-1024). The x-axis indicates the benchmarks and the y-axis depicts the average memory space in bytes on a logarithmic scale. BT, CG (plug-in version), FT (plug-in version) and SP benchmarks have almost the same space complexity (on average per node) for equivalent number of processes. This is because they only have one call-path cluster. For IS, LU, and S3D the number of call-path clusters is larger than one, but still constant. Therefore, the average space per each node drops when increasing the number of processes. The number pf clusters changes for both POP (see Table 3.6) and MG (see discussion around Fig. 3.2), the number of call-path clusters varies, which explains why we observe different space complexities.

Overall, the small footprints of traces and space requirements illustrate the benefits of multi-level clustering, which facilitates analysis without incurring extra cost during tracing or sacrificing accuracy, as results demonstrate.

3.7 Related Work

A commonly utilized tracing tool for MPI communication is Vampir [11], a commercial post-mortem trace visualization tool. It uses profiling extensions to MPI and facilitates the analysis of message events of parallel execution, helping to identify bottlenecks and inconsistent run-time behavior. While the trace generation supports filtering on trace files, which are stored locally, trace complexity increases with the number of MPI events in a non-scalable fashion. HPCTOOLKIT [53] uses statistical sampling to measure performance; it provides and visualizes per process traces of sampled call paths. In HPCTOOLKIT, 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. As noted previously, 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, finding an appropriate rate of sampling is complicated, and the cost of having a dense CCT is high. In contrast, clustering with ScalaTrace provides a full trace file without resorting to sampling and it does so at very low cost by leveraging a 64-bit stack signature.

Another approach, utilized in [30] and [31], features k-means clustering to select representative data for migration of objects in $CHARM++$. A density-based clustering analysis was proposed in [36], [20] and [19] 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). The proposed clustering algorithms are expensive in terms of time complexity, especially for extreme-scale sizes. Clustering with ScalaTrace is suitable for exascale computing because it not only utilizes a low overhead clustering algorithm with a $\log P$ complexity, but it also divides clustering and merge processes into two different phases. Separating the clustering algorithm reduces the complexity of the merge process significantly.

Phantom [65], 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 [61]. Second, because the paper focuses on performance prediction, it emphasizes computational similarity and does not sufficiently cover communication behavior. Reporting one or two clusters for SP and BT and one cluster for CG shows how their orthogonal objectives result in different clustering decisions.

Another scalable clustering algorithm for tracing toolsets is CAPEK [17], a parallel clustering algorithm based on CLARA [26] that enables in-situ analysis of performance data at run time. Even though the algorithm is logarithmic, the process of clustering and creating the global trace file is based on trace sampling. The merging overhead and the process by which the sample traces are expanded to present the overall behavior of the cluster apply to the duality of "effort and progress" metrics, but this does not generalize to n-dimensional clustering of metrics while our signature-based parameter clustering does.

For instance, a single parameter, such as the count, could produce a significant difference between two processes with the same execution structure. In contrast, our algorithm is not only logarithmic and has low overhead, but it also captures different parameters within the main clusters by means of parameter signatures. It then merges them in a linear manner and captures the different execution structures by means of call-path signatures.

Since CAPEK is a variant of k-medoids, finding a proper k is a challenge solved via the Bayesian Information Criterion (BIC) [44]. In Call-path+Parameter clustering, by dividing the merge process, the number of clusters is a function of the number of main clusters. As noted previously, the most costly operation in clustering with ScalaTrace is a function of events, not a function of clusters. Sub-clusters merge in a linear fashion within each main cluster.

TotalView [50] and DDT [3] are debugging tools with demonstrated scalability for large numbers of processes but are prone to extended response time during simple operations (e.g., timeline scroll) due to large amounts of data being processed. The Stack Trace Analysis Tool [34] supports petascale debugging with lightweight tools on an entire parallel application to reduce the problem search space to a manageable subset of tasks. These tools process the entire trace data set of all tasks while we operate on a trace of a small subset of nodes (of just one per cluster).

Jumpshot [64] is a trace visualization tool capable of displaying traces of programs running on a large number of processors for along time. ScalaTrace manages highly compressed traces, which would either need to be decompressed before being visualized in Jumpshot, or, even better, Jumpshot would need to be enhanced to interpret relative encodings and provide timeline progressing using delta-times, which would be much more efficient that handling its current uncompressed traces.

Aguilera et al. [2] propose hierarchical clustering to select a representative trace for each cluster of processes. This work first extracts communication data from a trace file, summarizes extracted communication information, creates a distance matrix, performs hierarchical clustering, and eventually identifies process pairs of interest. To reduce the trace file size, Lee et al. [32] use k-Means clustering to select representative data. The main difference between Call-path clustering and the aforementioned clusterings is that they focus on clustering of communication performance data to discover potential communication bottlenecks in distributed applications. ScalaTrace not only provides detailed performance information (e.g., COUNT, SRC, DEST, TAG, communication time, etc.), but also captures computation times between consecutive MPI calls, and it preserves the structure of the program in its traces.

Nickolayev et al. [40] propose another clustering algorithm using the Pablo performance instrumentation library [45] with a real-time statistical clustering infrastructure. The standard Pablo instrumentation software captures dynamic performance data via instrumented source code. Yan et al. [62] propose a dynamic instrumentation strategy. It reconstructs a parse tree from the source code, annotates the parse tree, generates control flow via parse-tree traversal, generates interval durations for each processing node, checks for consistency of event times across nodes, and eventually generates timed events.

All of these methods need access to the source code of the program, but the source code may not always be available. ScalaTrace generates traces without instrumenting source code.

Even though these methods reduce trace data across processes, they do not reduce trace data within a process. In contrast, ScalaTrace has two phases: intra- and inter-node compression.

Knupfer et al. [27] propose a data compression technique. The main focus of the work is on the intra-node compression step. The algorithm declares two sections of a trace as similar if the call graph context and measurements of the events match. 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 during intra-node compression. Traces remain readable after compression in ScalaTrace, while traces created by Knupfer et al. require decompression before they can be processed. Moreover, creating call-graphs is a costly operation. Our clustering algorithm uses call-path signatures, which are only 64-bits in length.

3.8 Conclusion

Scalability is one of the main challenges of scientific applications in HPC. This work contributes a novel multi-level clustering algorithm with $logP$ time complexity and low overhead. The approach relies on signatures to support n-dimensional metrics for cluster selection, much in contrast to a single metric of traditional cluster algorithms. The results of our experiments indicate that our clustering algorithm provides significant reductions in performance overheads making it suitable for extreme-scale computing. Unlike other clustering algorithms designed for large-scale problems, our approach is based on predominantly exact matching rather than on random processes or statistical approaches for sampling with compromised, lower accuracy. Our clustering algorithm is applicable to both strong and weak scaling applications.

Even though *CallPath+Parameter* Clustering 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. We address these issues in the next chapter.

Chapter 4

ACURDION: An Adaptive Clustering-based Algorithm for Tracing Large-scale MPI Applications

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 [55, 8]. Traces are also utilized to drive HPC simulations to determine the effect of interconnect changes for future procurements [49, 63, 38, 52, 66].

Today's tracing tools either obtain lossless trace information at the price of limited scalability (e.g., Vampir [11], Tau [47], Intel's toolsets [23], and Scalasca [18]) or preserve only aggregated statistical trace information to conserve the size of trace files (e.g., mpiP [54]). 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 [67]. In response, a number of communication compression techniques have been designed, including run-length compression [69] and structural compression [28, 41, 59].

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 [59] 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 [67] uses a hybrid static/dynamic compiler-aided compression technique [67] 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 [5]. 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 [5]. 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 lead node of a cluster. In practice, the number of clusters needed tends to be small (up to nine in our experiments) without sacrificing accuracy.

In Chapter 3, we 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 [7]. 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 work describes ACURDION, a scalable clustering algorithm for large scale applications. Figure 4.1 depicts the schematic of ACURDION with respect to ScalaTrace.

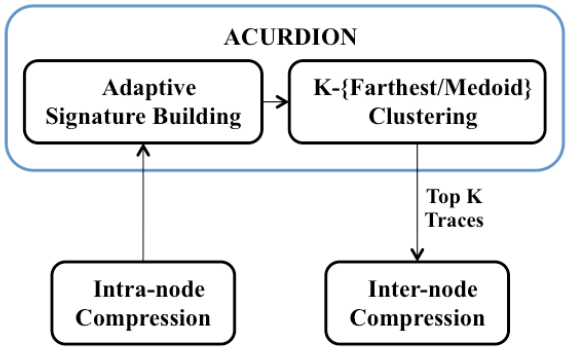


Figure 4.1: A Schematic of ScalaTrace with ACURDION

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.

At the beginning, a signature detection algorithm identifies the main characteristics of programs required for signature clustering. We consider eleven different signatures indicative of application behavior. The first and most important signature is *Call-path*, which is based on the stack 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 from different events. All other signatures are averaged parameter signatures that compose parameters from the MPI call event. We introduce the signatures in detail in the design section.

To evaluate the accuracy and scalability of our algorithm, we also designed a *reference clustering* approach that uses a reference signature. The reference signature covers call-path signatures by adding a sequence number to each MPI event as well as parameter clustering by keeping each MPI event's parameters uncompressed. Detailed implementation information about call-path, parameter clustering and reference clustering algorithms are provided in the following sections.

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.

4.1 Signature Building

In order to generate global traces with high accuracy, we found (based on our prior work in Chapter 3) 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 4 presents the pseudocode of this phase.

ALGORITHM 4: 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 ≠ child signature then
5  |  |  update the signature format;
6  |  end
7  |  Receive the child's signature format;
8  |  signature format = signature format OR child's signature format;
9  end
10 if a parent exists then
11 |  Send your signatures to your parent;
12 |  Send your signature format to your parent;
13 end
14 Broadcast signature format by rank root;
```

The time complexity of the algorithm is $log(P)$. Fig. 4.2 and Fig. 4.3 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).

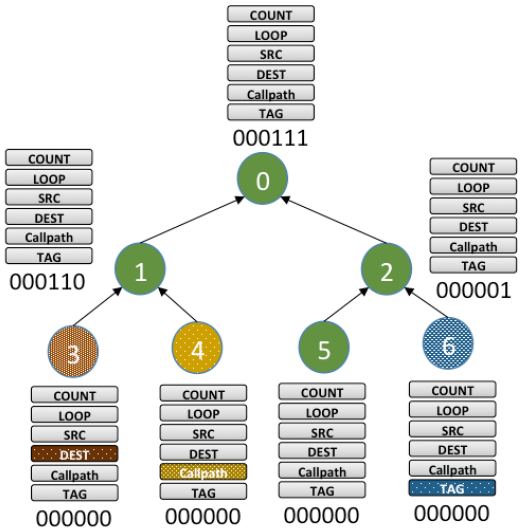


Figure 4.2: A Sample of Creating Signatures $O(1)$

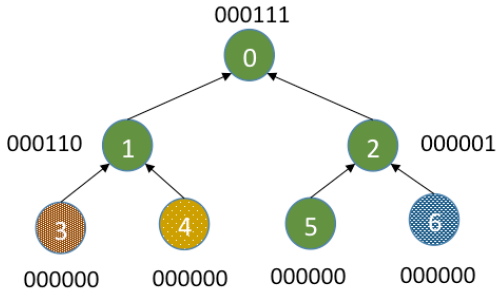


Figure 4.3: A Sample of Building Signature Format $O(log(P))$

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.

4.1.1 Single-Step and Double-Step Clustering

Fig. 4.4 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 lead processes to create the global trace file, inter-compression on the selected traces is applied.

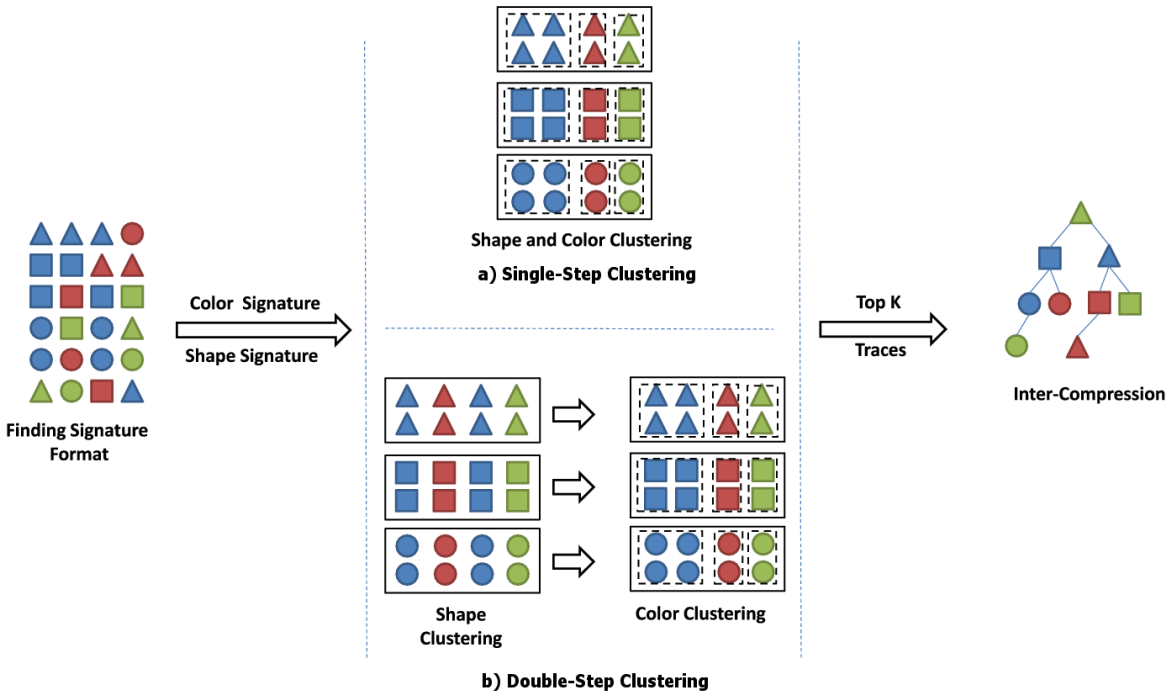


Figure 4.4: Overview of Proposed Clustering Algorithm

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 5: 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         end
11         foreach cluster ∈ AllNode list - TopK list do
12             Find the closest cluster;
13             Assign the cluster to the closest one;
14         end
15     end
16 end
17 if a parent exists then
18     Send your list of K clusters to your parent;
19     Send signature of head of top K clusters to your parent;
20 end
21 Broadcast Top K by root;
```

Algorithm 5 depicts ACURDION's K-Farthest clustering algorithm. Fig. 4.5 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 [42].

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 4.4).

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.

4.2 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 $O(n^2logP)$. On the other hand, for ScalaTrace with the clustering algorithm, only a set of K lead processes with different signatures are participating in this operation. During the last phase of Fig. 4.4, K different nodes are merged.

The cost of inter-compression with clustering is $O(n^2logK)$, 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 trace files of the K selected lead processes considering all members of the cluster. As an example, consider Fig. 4.5. 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., lead processes linearly traverse their trace and replace a cluster ranklist (representing all members' IDs) with an event ranklist (compatible with the original ScalaTraceV2 format).

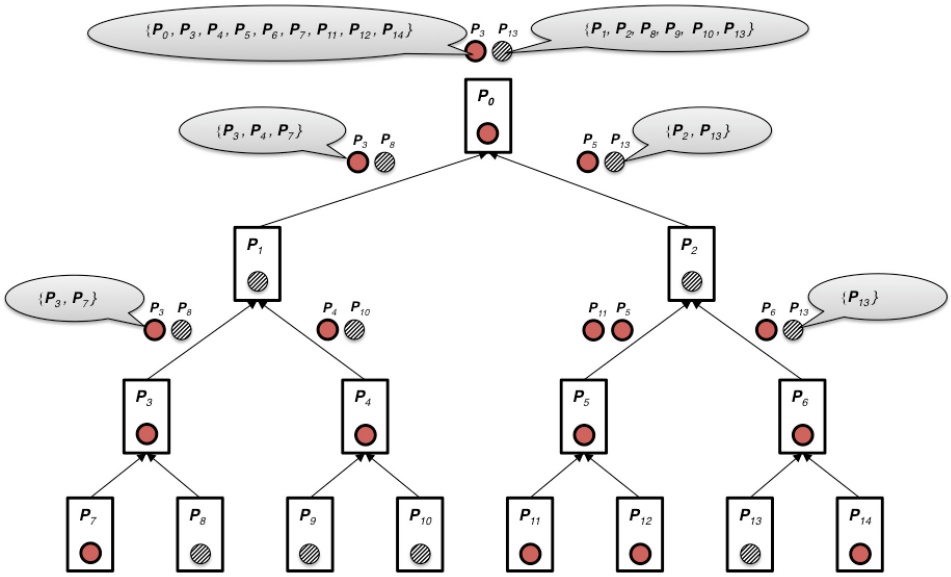


Figure 4.5: An illustration of K-Farthest or K-Medoid Clusterings ($K = 2$)

4.3 Experimental Setup

TACC's Stampede [51], 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 [7]; (2) Sweep3D [29], 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 100×100×1000; (3) LULESH, which approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements [25]. Results of ACURDION are compared to reference clustering and related work on signature-based clustering [5] of which we obtained a copy.

4.4 Results and Analysis

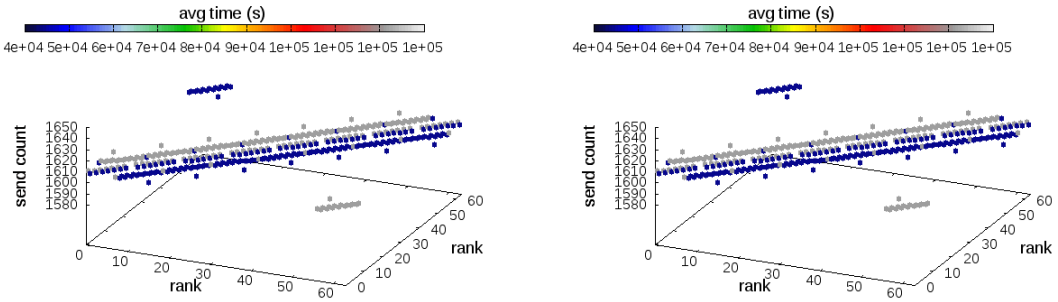
To assess the accuracy of the proposed clustering algorithm, we conducted two types of experiments. First, we tested the accuracy of point-to-point communication through heatmaps to make sure ACURDION captures communication patterns. Second, to verify the accuracy of collective and point-to-point operations, 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. 4.6 and Fig. 4.7 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. 4.7(e)) and after (Fig. 4.7(f)) clustering (same for CG and MG).

Table 4.1 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 Fourier Transform (*FT*) tested in our prior work [5], could benefit from *KEY* and *COLOR* signatures.

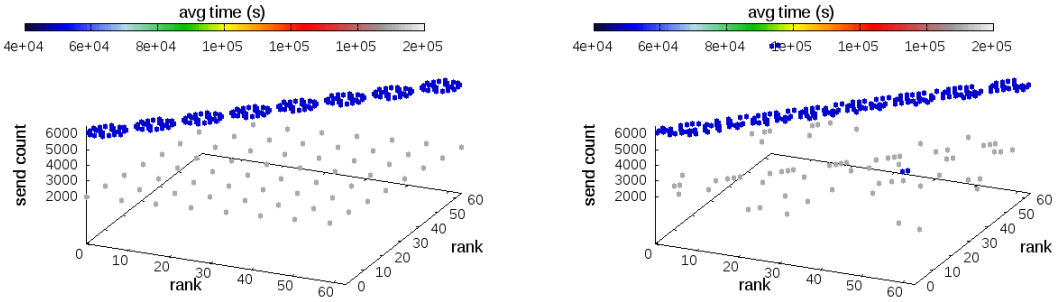
Table 4.1: Signature Format

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



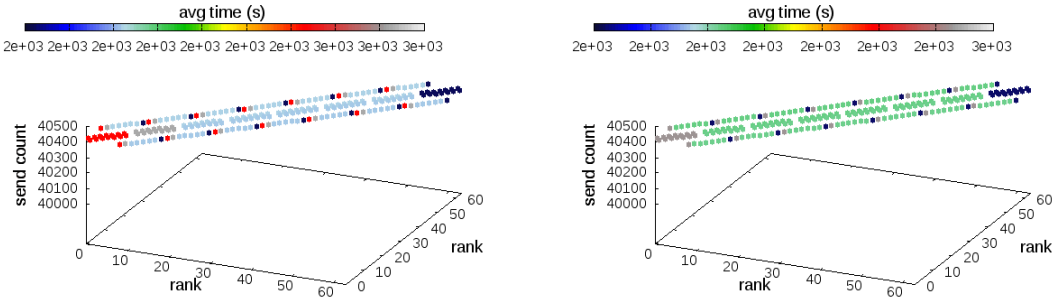
(a) BT

(b) BT*



(c) CG

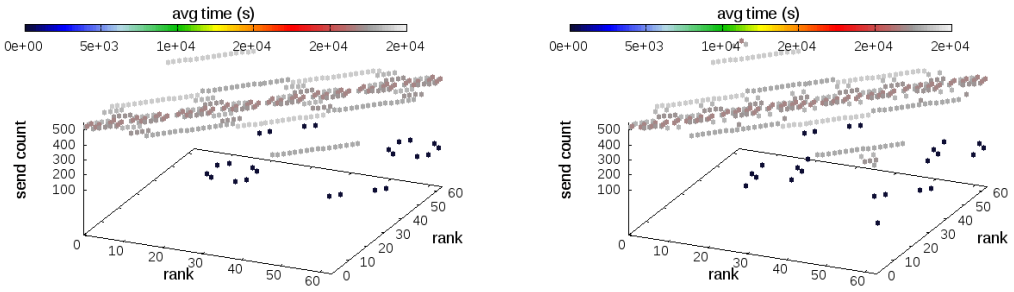
(d) CG*



(e) LU

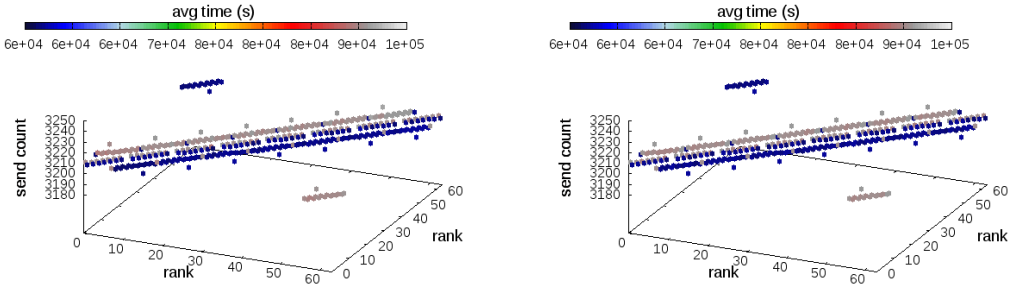
(f) LU*

Figure 4.6: Heatmaps of Point-To-Point Communication for 64 Processes Through ACURDION (K=9)



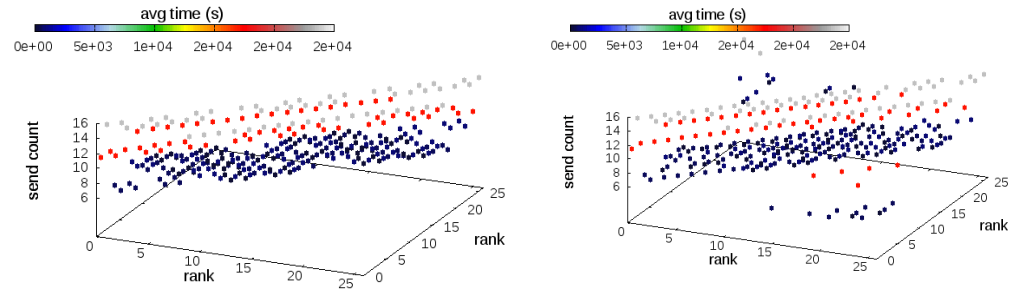
(a) MG

(b) MG*



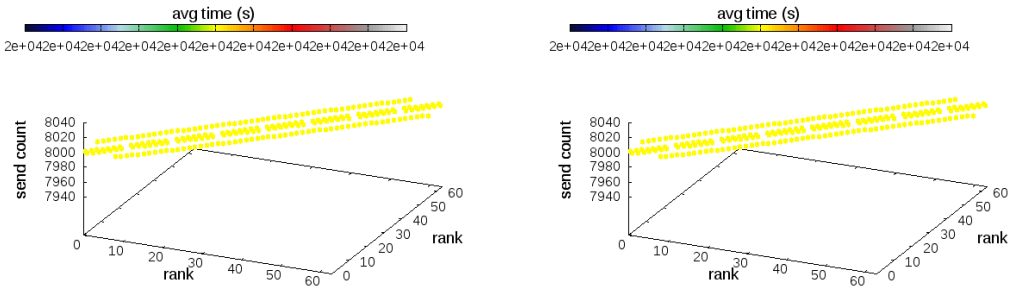
(c) SP

(d) SP*



(e) Lulesh

(f) Lulesh*



(g) S3D

(h) S3D*

Figure 4.7: Heatmaps of Point-To-Point Communication for 64 Processes Through ACURDION (K=9)

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 4.2 covers the percentage of matching clustered parameters relative to non-clustered ones. The similarity of point-to-point events was already depicted in Fig. 4.6 and Fig. 4.7. 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.

Table 4.2: 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%

We chose a maximum number of nine clusters ($K = 9$) for ACURDION, which we experimentally determined based on captured communication patterns of related work [5]. This suffices to represent average communication time, send count, and source and destination ranks for point-to-point communication. Table 4.3 also shows that for CG increasing the number of clusters does not improve the accuracy of 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.

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

Table 4.3: 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%

difference of around 0.1% reported in Table 4.2, which is not visible in Fig. 4.6(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) [25] 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 [25]. We observe a slight diffusion in the communication pattern in Fig. 4.7(f). However, clustering covered all events and retained parameters such as $COUNT$ and $LOOP$ as Table 4.2 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 4.2 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 4.2 indicates that ACURDION clustering captures the parameters accurately. In related work [5], 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.

4.4.1 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.

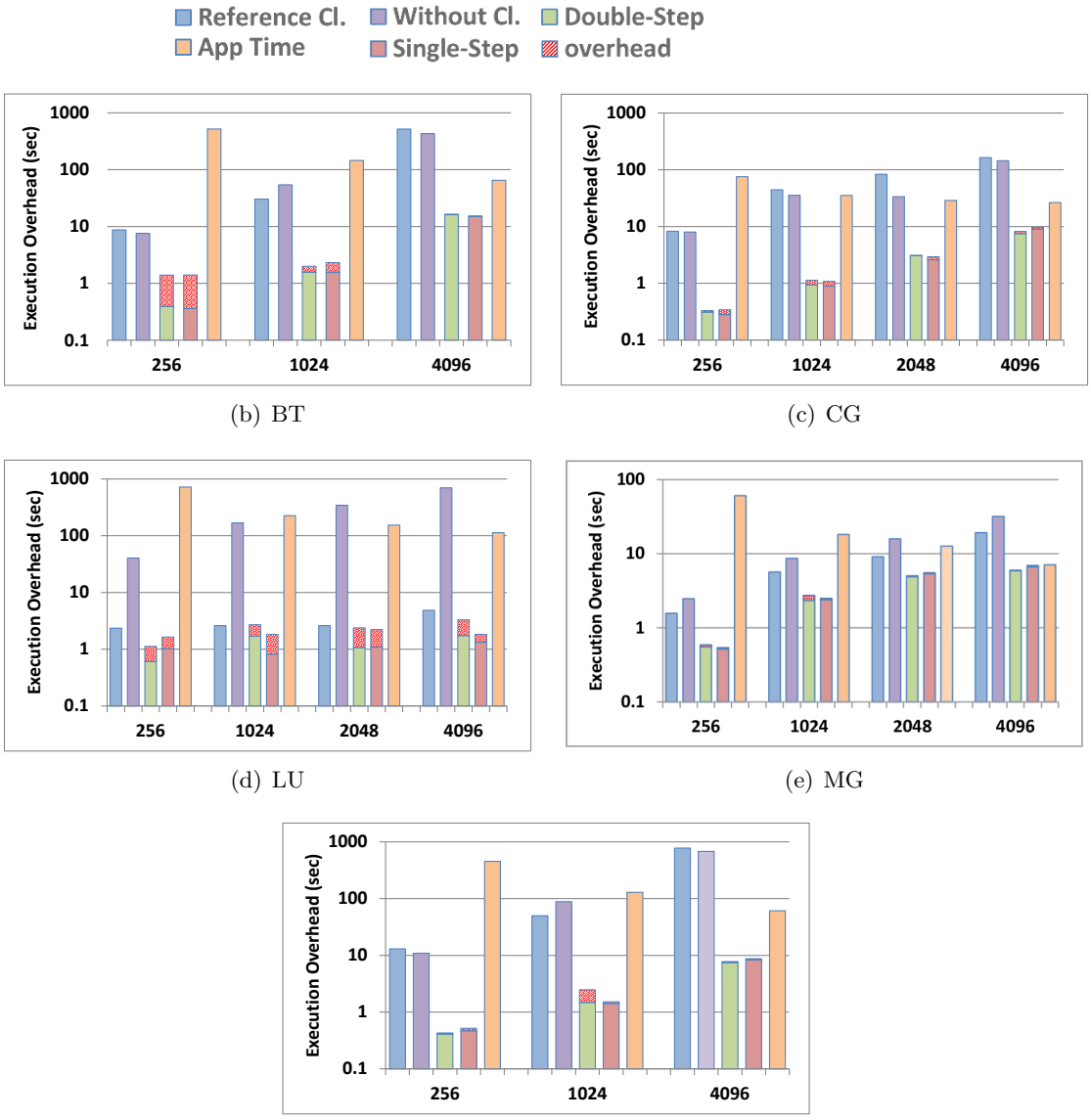
Some of the benchmarks only support strong scaling (most NAS codes) while others support weak scaling (Sweep3D, Lulesh) so that different sets of benchmarks are reported in these experiments.

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. 4.8 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 [59], 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

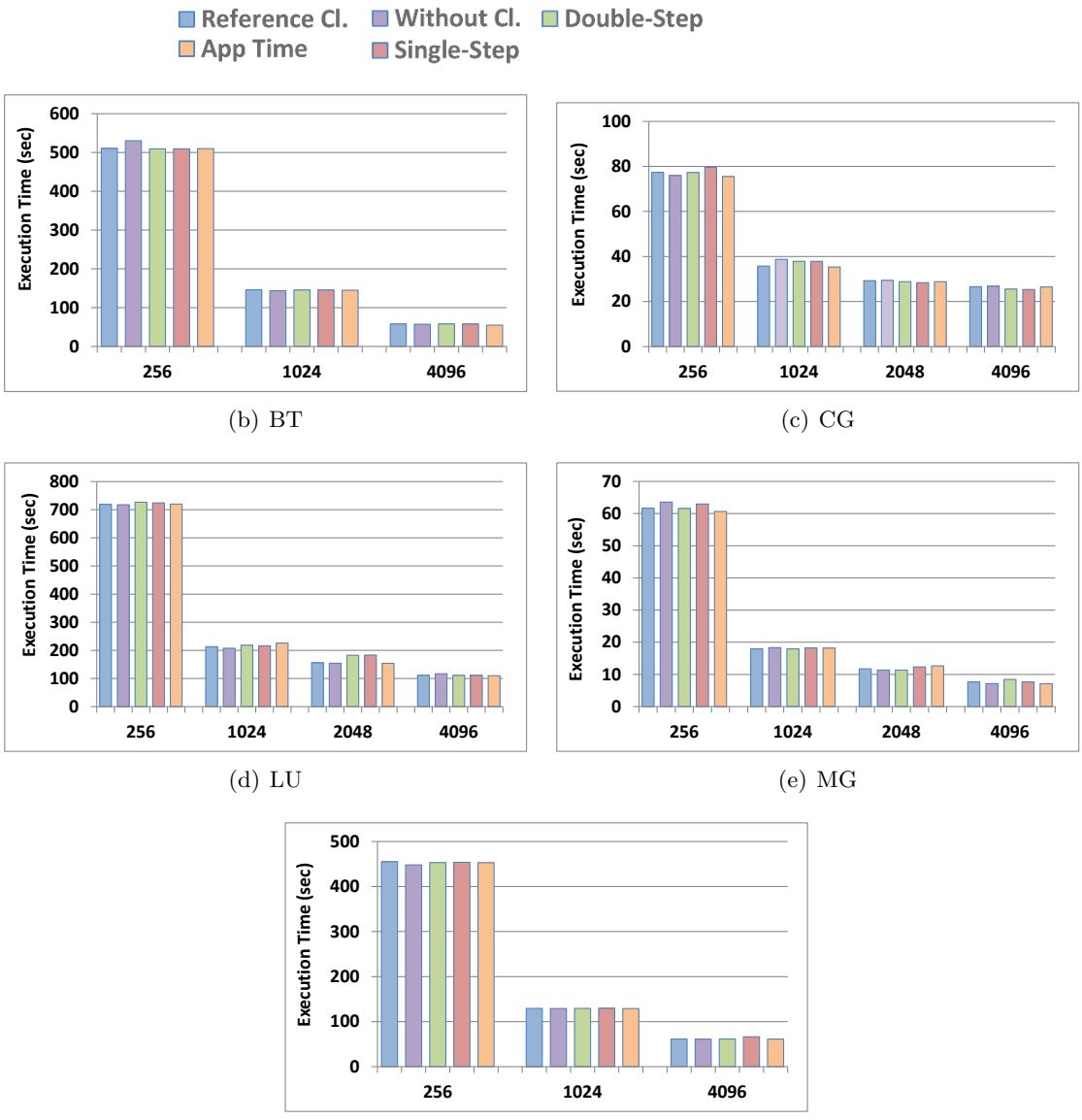


(b) BT (c) CG

(d) LU (e) MG

(f) SP

Figure 4.8: Execution Overhead for NAS benchmarks: Strong Scaling, Nodes/Tasks=1/16



(b) BT

(c) CG

(d) LU

(e) MG

(f) SP

Figure 4.9: Replay Time of Traces: Strong Scaling, Nodes/Tasks=1/16

computation is "replayed" as sleep time to resemble the same distance between communication calls. The communication 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. 4.9 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 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.

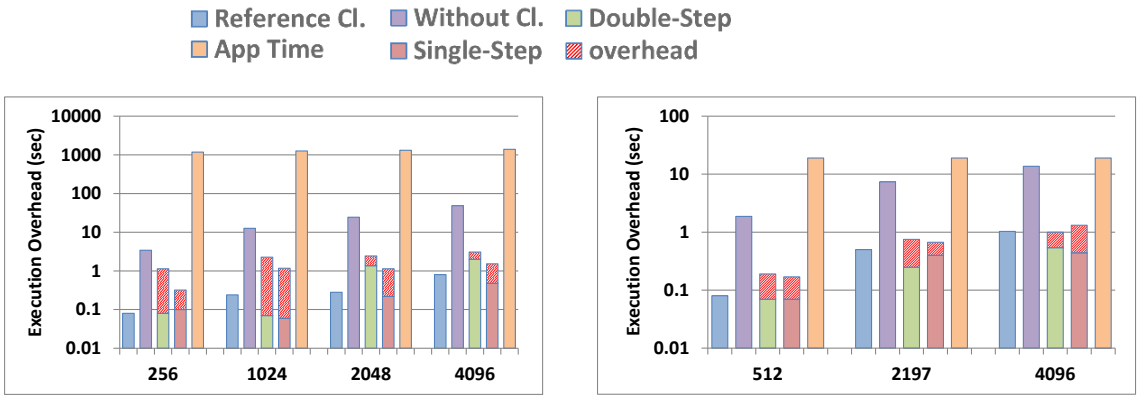
4.4.2 Weak Scaling

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. 4.10 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

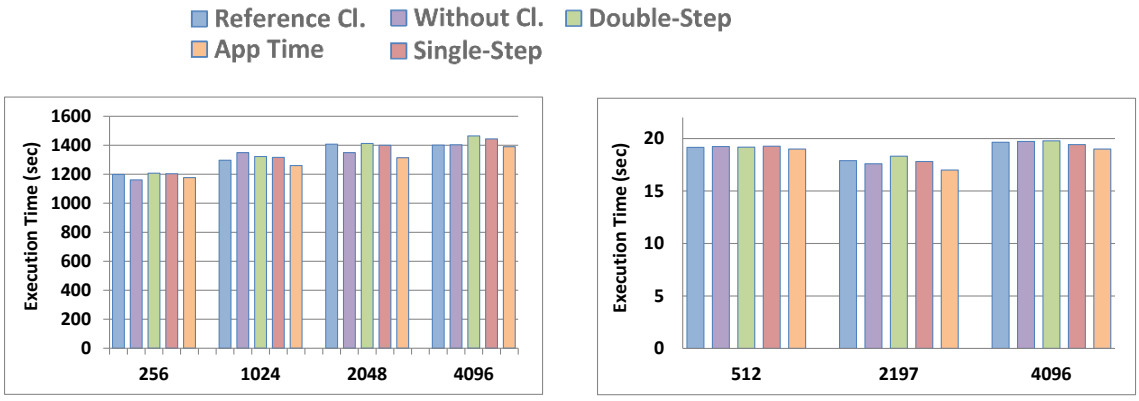
outperforms double-step slightly whereas in all previous experiments no clear winner could be declared between the two. Reference clustering outperforms ACURDION here for the first time. However, we will later show that ACURDION outperforms reference clustering in terms of space complexity, which can have a significant impact for large-scale tracing.

Fig. 4.11 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

Figure 4.10: Execution Overhead: Nodes/Tasks=1/16

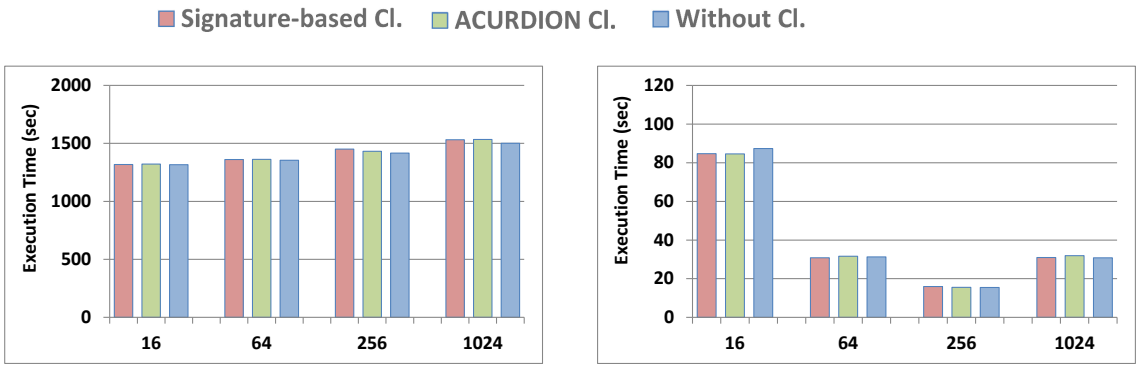


(b) Sweep3D (c) Lulesh

Figure 4.11: Replay Time of Traces: Weak Scaling, Nodes/Tasks=1/16

4.4.3 Signature-based vs. ACURDION

To compare the accuracy of trace files generated by ACURDION to signature-based clustering [5], 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 4.12 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. 4.12. 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

Figure 4.12: Replay Time of Traces: Nodes/Tasks=1/16

4.4.4 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 4.13 shows that the accuracy of traces are very close to each other. Since the execution overheads depicted in 4.4 and 4.6, and standard deviation depicted in 4.5 and 4.7 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 4.4: 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 4.5: 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 4.6: Execution Overhead: Sweep3D - Weak Scaling

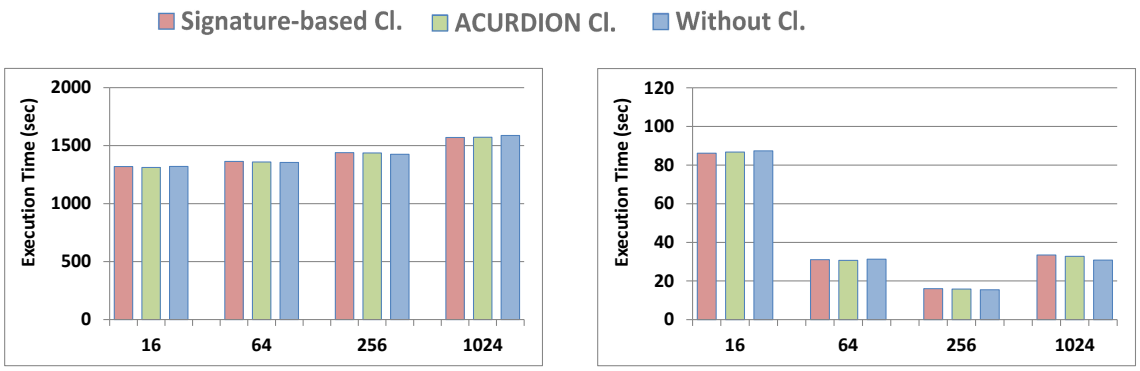
# Processes	16	64	256	1024
K-Medoid Overhead (s)	0.34	0.4	0.36	0.75
K-Farthest Overhead (s)	0.23	0.27	0.29	0.92

Table 4.7: 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

4.4.5 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 4.8 depicts the number of clusters and the required memory space in KB per method



(b) Sweep3D (c) CG

Figure 4.13: Replay Time of Traces: Nodes/Tasks=1/16

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 4.8: 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 [5], 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 lead processes 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).

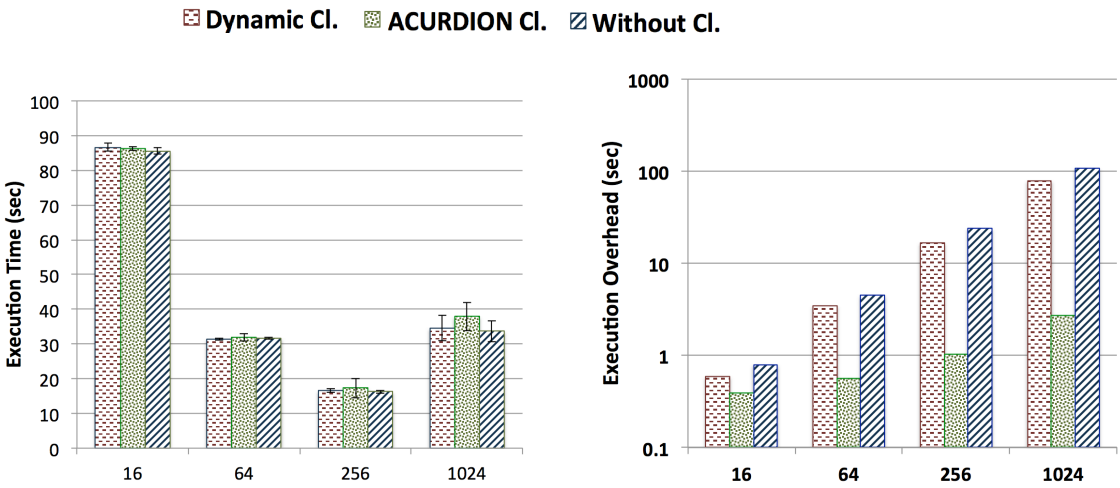
4.4.6 Dynamic Clustering

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 4.14 and Figure 4.15 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.



(b) Replay Time

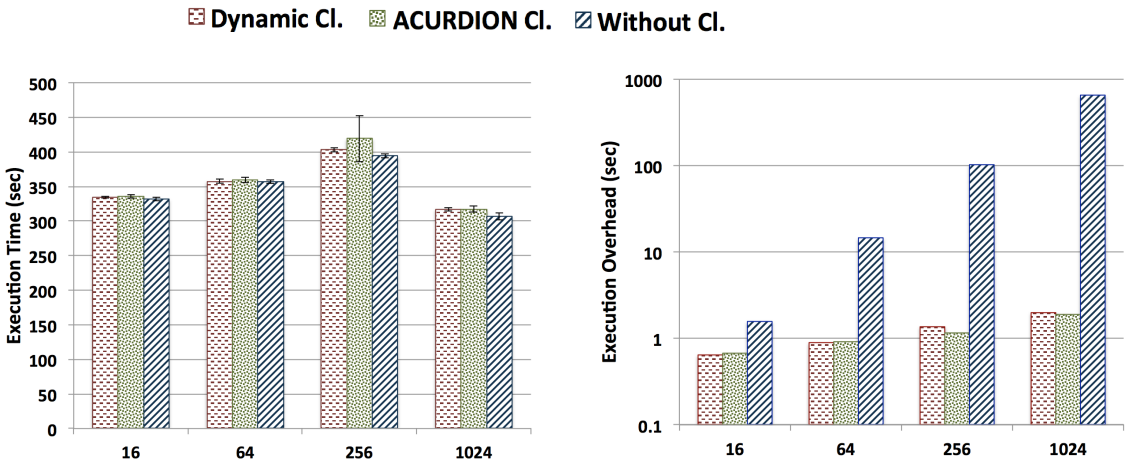
(c) Execution Overhead

Figure 4.14: Execution Overhead and Replay Time for CG Class C: Nodes/Tasks=1/16

4.5 Related Work

ACURDION enhances our prior work *Call-path+Parameter* clustering in two directions. First, the parameter signature of *Call-path+Parameter* 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 user input plug-ins for benchmarks that have unique communication behavior (e.g., CG).

CYPRESS [67] 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, [5] 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



(b) Replay Time

(c) Execution Overhead

Figure 4.15: Execution Overhead and Replay Time for LU Weak Scaling: Nodes/Tasks=1/16

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 [68] 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 [11], 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.

HPCTOOLKIT [53] utilized statistical sampling 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 [36, 20, 19] 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 [30] and [31]. 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 [65], 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 [61]. 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 [14]. 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 [26] was proposed in CAPEK [17] 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) [33] provides scalable detection of task equivalence classes based on the functions that the processes execute. The Probabilistic Calling Context (PCC) approach [9] 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.

4.6 Conclusion

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.

Chapter 5

Chameleon: Online Scalable Performance Analysis of MPI Programs through Signature-Based Clustering Algorithms

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 [55, 8].

Today's tracing tools either obtain lossless trace information at the price of limited scalability (e.g., Vampir [11], Tau [47], Intel's toolsets [23], and Scalasca [18]) or preserve only aggregated statistical trace information to conserve the size of trace files (e.g., mpiP [54]). 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 [67]. In response, a number of communication compression techniques have been designed, including run-length compression [69] and structural compression [28, 41, 59].

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. Therefore, tool designers need to develop new strategies to address these problems.

We showed in previous chapters 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 different application behavior. Therefore, 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 work 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 [59], 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 [41, 60]). 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 2). 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 algorithm addresses the high overhead due to scaling out by significantly reducing P to almost a constant value, thereby effectively eliminating 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.

Contributions:

• We introduce and describe 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.

5.1 The Proposed Clustering Algorithm

In the design of Chameleon, we considered two main steps: (1) an online clustering algorithm helps to group processes with different 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 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.

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 5.1 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 three different states: (1) **All Tracing** (AT): In this state, the $Call - Path$ of the last call is different from the current call. (2) **Clustering** (C): if the $Call - Path$ of the current call is the same as the previous call, then it is an indication of iterative behavior, i.e., clustering occurs, and the partial traces are merged. This transition is unidirectional from "AT" to "C". (3) **Lead** (L): if the system is in state "C", and the $Call - Path$ is still the same as the old one, then there is no change in behavior, so no new clustering is required, and state "L" is entered. Otherwise, if there is any difference, merge all partial traces using the inter-node compression, and then go to state "AT". As an operational optimization to reduce cost, we consider staying in the state of "L" synonymous to the state "AT" in terms of its actions, where lead processes only perform intra-compression and skip clustering/inter-node compression. Once they leave state "L", then they flush partial traces and perform inter-node compression. State "F" depicts $MPI\_Finalize$, which is reachable from any of the marker states.

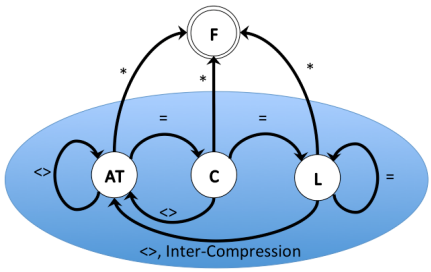


Figure 5.1: Transition Graph

Algorithm 6 presents the pseudocode for the transitions of the graph. Creating a signature has $O(n)$ complexity, where 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 $O(nlog(P))$.

We used K-Farthest, K-Medoid and multi-level hierarchical clustering algorithms in previous section. As previously mentioned, clustering happens over signatures, not traces. There are two phases. First, clustering using signatures, and then lead traces are merged.

Algorithm 8 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 our prior work in sections 3 and 4, 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.

Chameleon ensures that after each process creates three signatures in Algorithm 6, 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. We showed in previous chapters that the number of $Call-Path$ usually is below 9, which is sufficient to cover stencil codes. Nonetheless, one can change the value of K dynamically should the number of different $Call-Path$ signatures exceed K.

Algorithm 7 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). After picking lead processes, it merges other non-selected clusters 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 8:

(1) The complexity of creating signatures is $O(n)$. We 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.

ALGORITHM 6: 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";
```

ALGORITHM 7: 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
```

(2) For the "AT" state, only the Algorithm 6 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 6, 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 8 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 6 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 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

ALGORITHM 8: Online Inter-Compression using Clustering

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

```
Increment Marker_Call_Counter;
if Marker_Call_Counter % Call_Frequency != 0 then
    return;
end
ClusteringState = Call Algorithm 6;
if ClusteringState == "C" OR ClusteringState == "L" then
    if ClusteringState == "C" then
        if a left/right child exists then
            Receive list of left_K / right_K clusters;
            Receive signature of head of top left_K / right_K clusters;
            Merge left_K / right_K clusters + yourself into AllNode list;
            if left_K + right_K + 1 > K then
                local K = K / Number of Call-Paths;
                foreach Call-Path signature do
                    TopK = TopK + findTopK(local K, SRC and DEST signatures);
                end
            end
        end
        if a parent exists then
            Send current list of K clusters to parent;
            Send signature of head of top K clusters to parent;
        end
        MPI_Bcast (Top K) by root;
    end
    if my rank ∈ Top K list then
        Replace ranklist of collected events with my cluster ranklist;
        tempRank = assign a temp rank from Top K;
        if a left/right child exists then
            Receive the left/right child traces;
            Merge their trace with yours;
        end
        if a parent exists then
            Send your trace to parent;
        end
    end
    if my rank == root rank of Top K list OR rank == 0 then
        if root of Top K list != 0 then
            if rank == 0 then
                Receive partial global trace from root of top K;
            end
            else
                //root of Top K list
                Send partial global trace to rank 0;
            end
        end
        if my rank == 0 then
            Merge partial trace with Online Global Trace;
        end
    end
    //All nodes;
    Delete your partial trace;
end
```

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).

Note that 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 8 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. Our experiments in the next section show that both r and K are small numbers for real-world applications. 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 [7] and Sweep3D [29]. 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) [24] 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 is a generic framework for representing and running medical application pipelines in parallel [6]. It follows the master-worker paradigm and layers MPI4Py [13](MPI for Python) over 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.

5.2 Results and Analysis

Table 5.1 depicts the number of clusters considered for the experiments (see Chapter 4).

Table 5.1: # of Clusters for the Tested Benchmarks

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

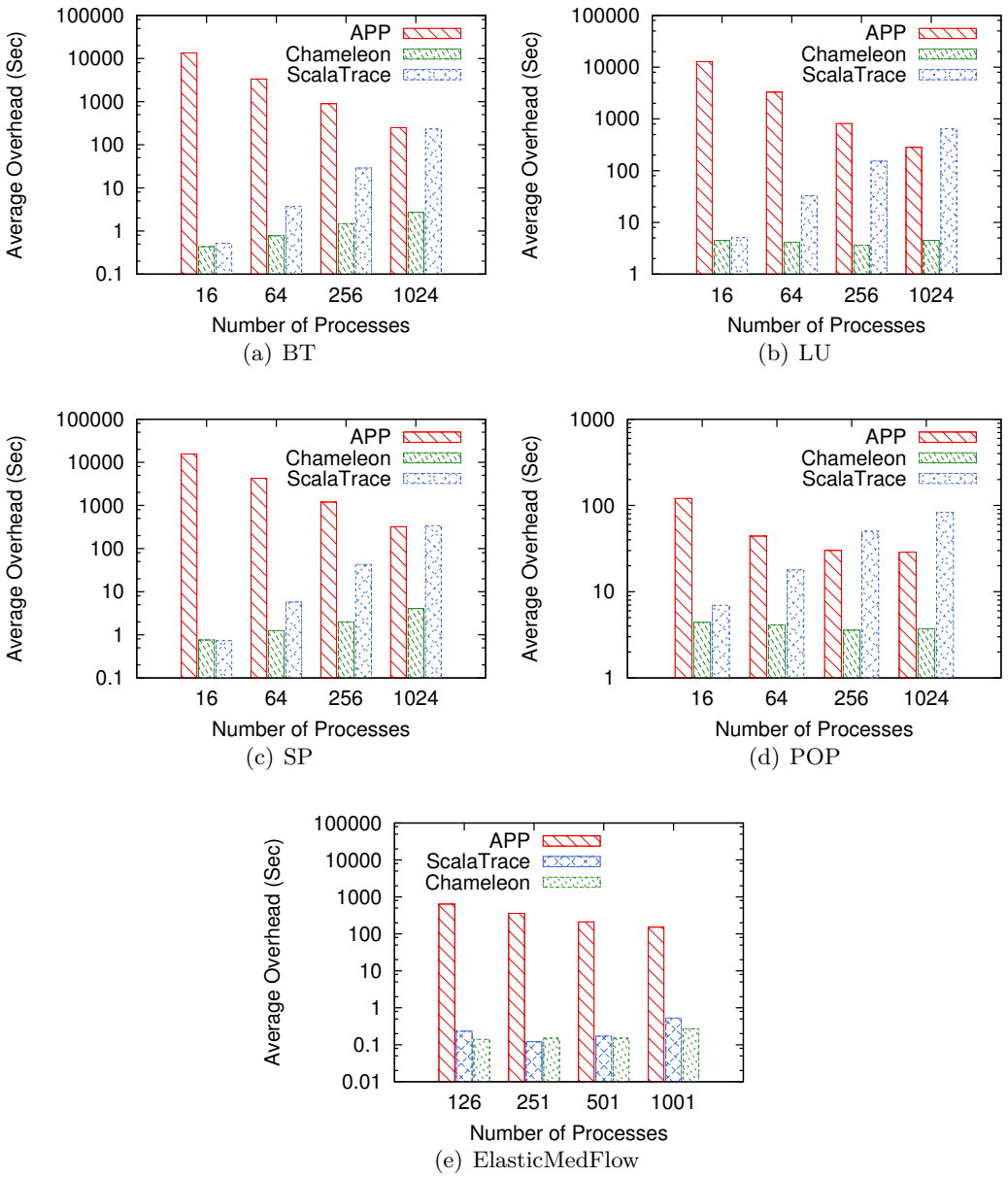


Figure 5.2: Execution Overhead for NAS benchmarks - Strong Scaling - Nodes/Tasks=1/16

Table 5.2 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.)

The results in Tables 5.1 and 5.2 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.

Table 5.2: # 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	9	1	8	1
EMF(126)	288	32	9	1	7	1
EMF(251)	144	16	9	1	7	1
EMF(501)	72	8	9	1	7	1
EMF(1001)	36	4	9	1	7	1

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 5.2.

5.2.1 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 5.2 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 < 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 2-3 orders of magnitude lower overhead than ScalaTrace under strong scaling (except for extremely small traces).

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 [58]. 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 5.3 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 overall as accurately as per-node traces with ScalaTrace under strong scaling.

5.2.2 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 5.4 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.

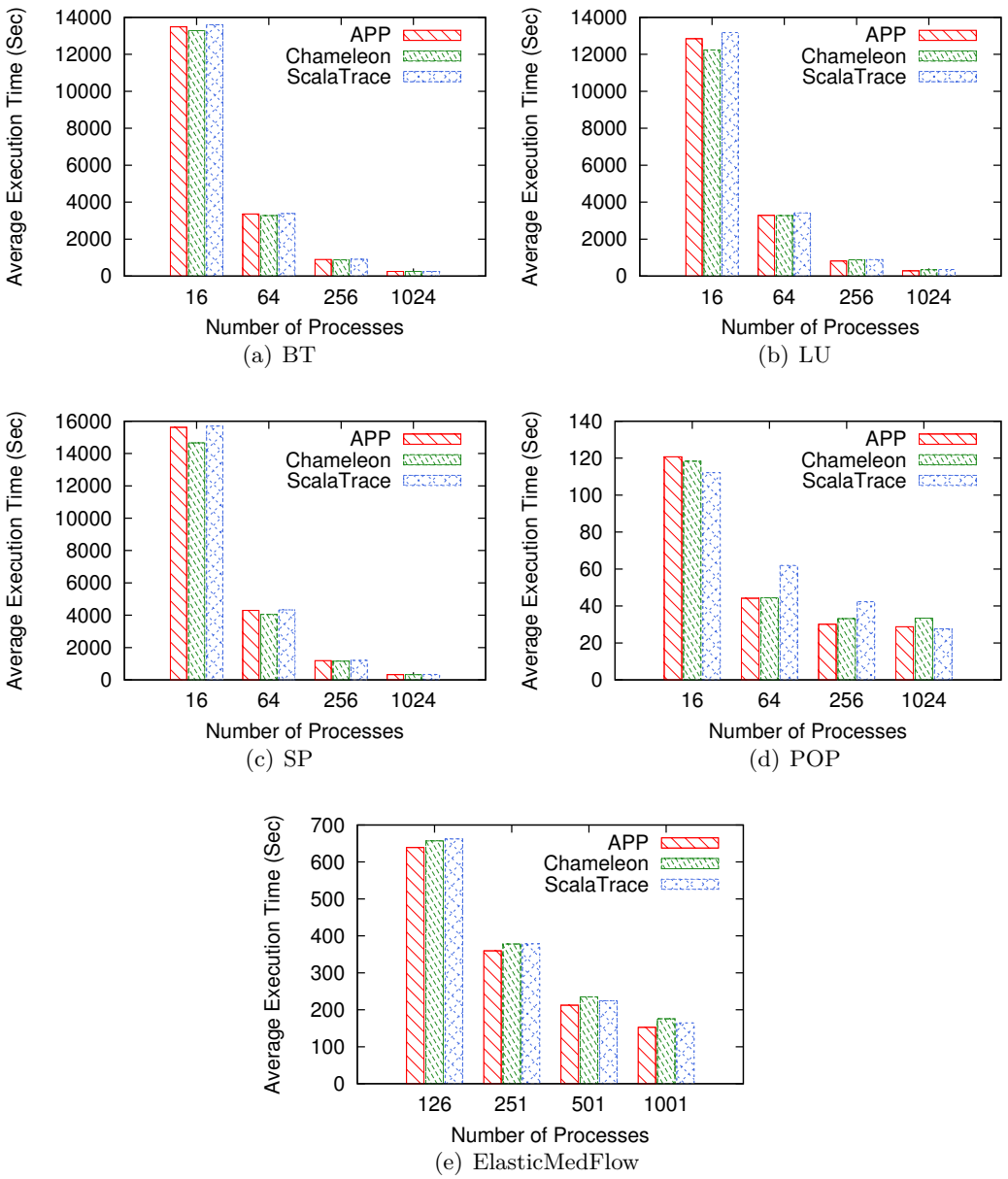
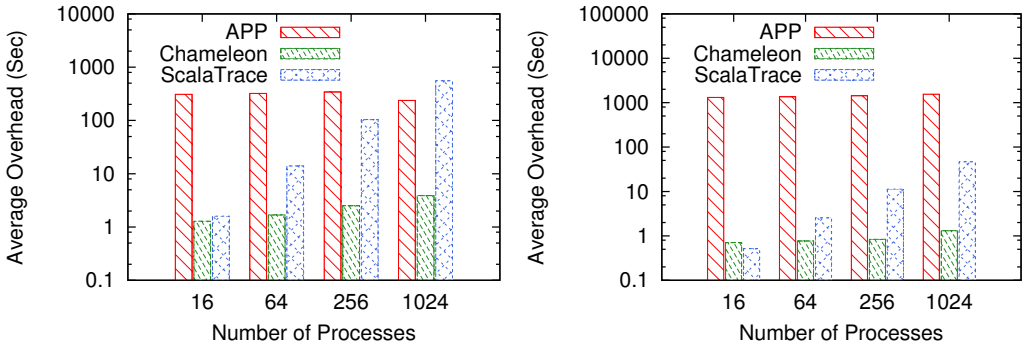


Figure 5.3: Replay Time of Traces - Strong Scaling - Nodes/Tasks=1/16

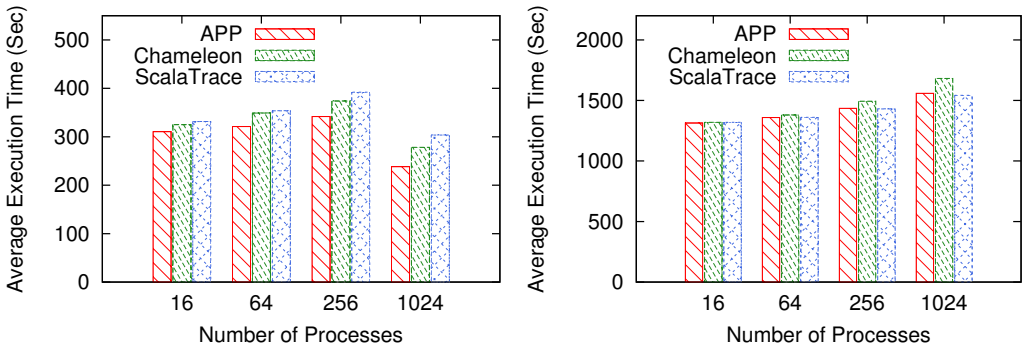
Observation 4: Chameleon's clustering results in 1-3 orders of magnitude shorter execution time than ScalaTrace under weak scaling.

Figure 5.5 depicts the overall trace-file replay time in seconds on a linear scale (y-axis) for different numbers of processors (x-axis). In analogy to strong scaling, it illustrates 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.



(a) LU (b) Sweep3D

Figure 5.4: Execution Overhead - Weak Scaling Nodes/Tasks=1/16



(a) LU (b) Sweep3D

Figure 5.5: Replay Time of Traces - Weak Scaling - Nodes/Tasks=1/16

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

5.2.3 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 5.6 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 5.2). 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

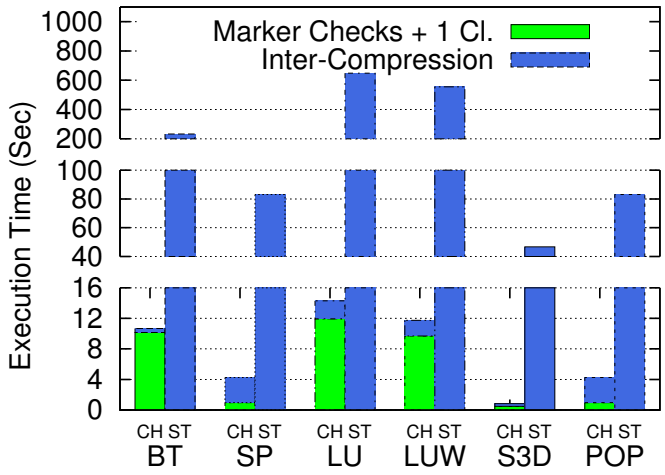


Figure 5.6: Overhead of Chameleon vs. ScalaTrace - Max. # of Marker Calls and P=1024

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.

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 5.7 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, processes call a new $MPI\_Barrier$. This indicates a new $Call - Path$ and changes the program phase. Figure 5.8 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.

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

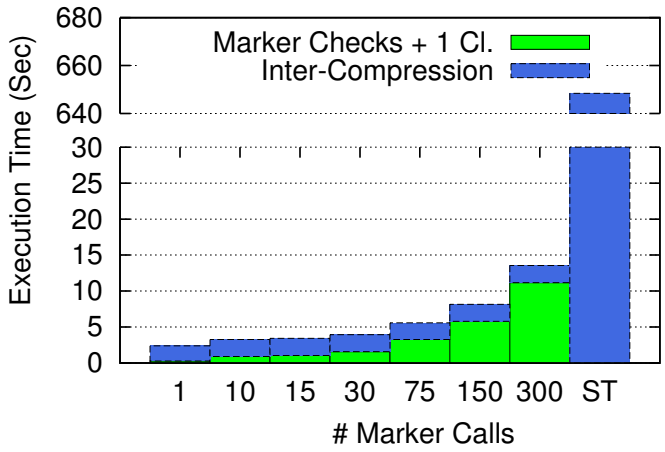


Figure 5.7: Overhead for Varying # of Clustering Calls for LU Class D - P=1024

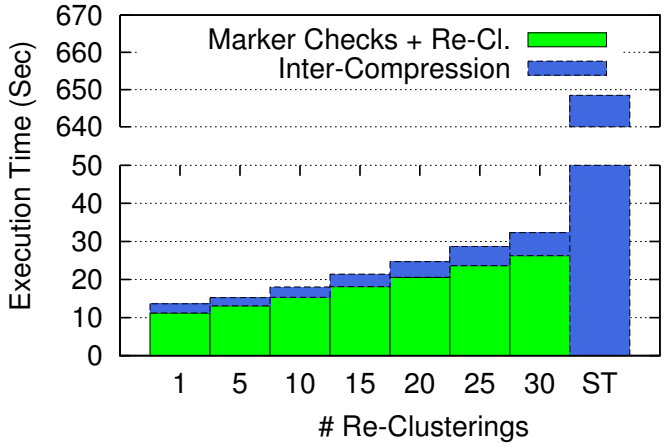


Figure 5.8: Overhead of Re-Clustering for LU Class D under 300 Marker Calls - P = 1024

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 5.9 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.

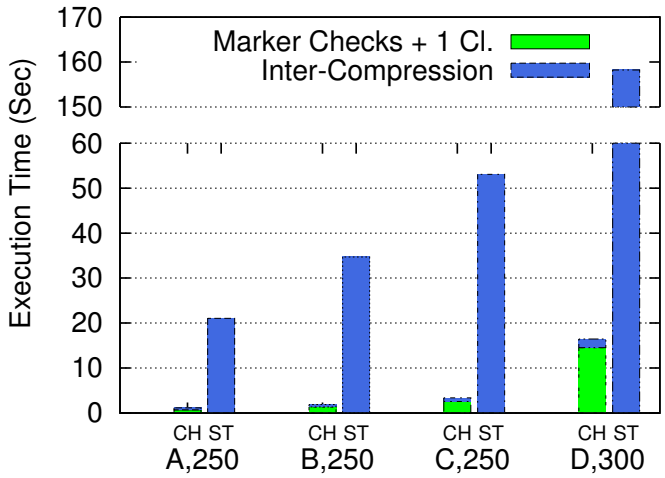


Figure 5.9: Overhead of Chameleon vs. ScalaTrace for Different Problem Sizes of LU - P=256

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

5.2.4 Space Complexity

Our algorithm follows the same space complexity as the $Call - Path + Parameter$ clustering 3 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 traces 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.

5.3 Related Work

We proposed signature-based clustering algorithms for ScalaTraceV2 in Chapters 3 and 4. Chameleon significantly improves over this prior work by developing a context-aware clustering framework that supports *online* inter-node compression.

CYPRESS [67] 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. As previously mentioned, the problem of ScalaTraceV2 was at the inter-compression step. In terms of space complexity, CallPath+Parameter clustering and ACURDION and Chameleon 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 plus Chameleon does not have any overhead during application compile time.

A commonly utilized tracing tool for MPI communication is Vampir [11], a commercial post-mortem trace visualization tool. It uses profiling extensions to MPI and facilitates the analysis of message events of parallel execution, helping to identify bottlenecks and inconsistent run-time behavior. While the trace generation supports filtering of trace files, which are stored locally, the trace complexity increases with the number of MPI events in a non-scalable fashion. ScalaTraceV2+Chameleon leverages *online* intra- and inter-node compression techniques.

HPCTOOLKIT [53] utilized statistical sampling 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, proposed by Gonzalez et al. [36, 20, 19], 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. [30] and [31]. 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 [65], 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 [61]. Second, because the paper focuses on performance prediction, it emphasizes computational similarity and does not sufficiently cover communication behavior.

A parallel clustering algorithm based on CLARA [26] was proposed in CAPEK [17] 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, ScalaTraceV2+Chameleon provides a full trace file without resorting to sampling and it does so at very low cost by leveraging 64-bit stack signatures.

5.4 Conclusion

Scalability is one of the main challenges of scientific applications in HPC. This chapter 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.

Chapter 6

ElasticMedFlow: Design and Implementation of a Scalable, Adaptable Multistage Pipeline for Medical Applications

6.1 Introduction

One of the benchmarks we used to evaluate Chameleon was ElasticMedFlow. We designed ElasticMedFlow as an MPI platform for bioinformaticians to runs their applications leveraging parallel computing capabilities.

We found that ElasticMedFlow represents a challenging data-parallel Python application for Chameleon. This is because ElasticMedFlow is based on a master-worker paradigm, which presents a challenge to the accuracy of traces generated by Chameleon due to the coarse grain of timing extracted from infrequent MPI calls. We encountered several challenges discussed in Section 6.2.

In this chapter, we first introduce ElasticMedFlow and then discuss how we used Chameleon with this Python application.

6.1.1 Background

Bioinformaticians face a number of challenges building medical pipelines. For example, every pipeline is constructed from scratch and thus there is limited code reuse. Researchers commonly build many scripts (written in a variety of programming languages) that simply run each stage in the pipeline sequentially. Since the code to run the pipeline and the pipeline's stages are deeply coupled, there is no way to reuse these scripts when it is time to build another pipeline. Thus, a lot of time is invested in rebuilding a similar infrastructure with small variation. Furthermore,

these solutions do not scale, because they essentially run one pipeline stage at a time. This is not a problem for a small number of patients, but limits scalability once a large number of patients need to be processed. Even in situations when data dependencies require that each stage be run serially, parallelism can be gained by analyzing data for multiple patients at the same time. Traditional pipelines operate on one patient at a time, and thus have no way to achieve such parallelism. Due to a lack of recovery and verification mechanisms, these pipelines often have to be rerun because they produce incorrect results or the machine running the computation fails.

After developing several medical MPI pipelines with the help of the Duke Cancer Institute and the medical school of UNC Chapel Hill, we realized bioinformaticians need a generic framework for representing parallel medical pipelines, which executes across a cluster computer for a large number of patients with a few modifications for different types of medical data.

We built a system called ElasticMedFlow that decouples the processes of running a pipeline from its structure. ElasticMedFlow allows researches to focus on the structure of their pipelines, and not how to execute them. Moreover, to reduce the amount of time wasted due to system failures, we included the ability for the system to recover most of the work that has been completed so far should a component fail. Each stage of the pipeline provides a configurable mechanism to validate the results and thus avoid situations where data was incorrectly processed without the researcher's awareness.

6.1.2 Design and Implementation

We have designed a robust JSON specification for representing a data analytics pipeline. The data dependency information contained within this specification is used to represent the pipeline as a directed acyclic graph (DAG). The structure of this graph prioritizes different stages of the pipeline. To effectively distribute work across a cluster, we created a master-worker paradigm using Python and MPI. This system supports dynamic work scheduling using the aforementioned DAG. We implemented a simple filesystem-based checkpointing system to limit the amount of wasted CPU time due to representing calculations after a system failure. We used two pilot projects to evaluate the capabilities of the designed system.

The first pilot study, a collaboration with UNC Chapel Hill, analyzes MPI data to aid in the diagnosis of osteoarthritis (see Figure 6.1). To quantifying cartilage changes in the knee, HPC facilities are essential as image databases, e.g., from the Osteoarthritis Initiative (OAI), have thousands of MRI images, which need to be analyzed. We estimate that it would take about 400,000 CPU hours of computation time to process this dataset — or about 50 years — on a single computer. Hence, parallel computing is critically needed to make real-time analyses of such datasets possible.

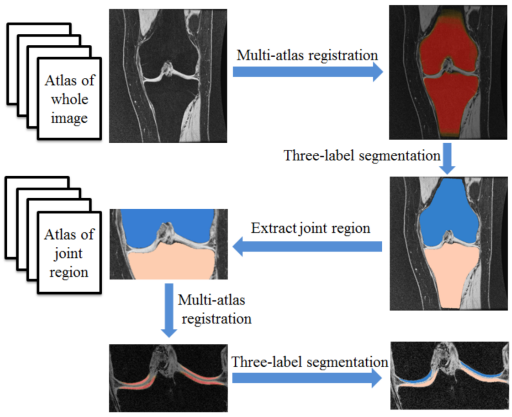


Figure 6.1: Flowchart of Cartilage Segmentation Pipeline

The second area of investigation focused on processing within a DNA pipeline for cancer diagnostics. The National Cancer Act of 1971 "declared war" on cancer. Today, cancer continues to be a major cause of ailment and deaths. According to a study conducted by the International Agency for Research on Cancer in Lyon, France, the incidence of cancer is expected to increase by more than 75% by the year 2030 in developed countries, and over 90% in developing nations [10]. Claudia Fischbach (Cornell) believes the key requirement in this war — to get ahead of this insatiable disease — is collaborative research between engineers and cancer biologists [16]. Researchers also believe success in biology and life sciences is a function of our ability to properly analyze big data sets, which in turn requires us to adopt advances in informatics [43].

For this pilot project, we focused on a sample *somatic mutation* detection pipeline. A *somatic mutation* is a genetic alteration acquired by a cell that can be passed on in the course of cell division.

Discovery of somatic mutations is exploited for prognostic and predictive biomarkers in the development of cancer therapeutics [4].

There are several methods for finding somatic mutations. One method is to take a tumor biopsy for genotyping. These studies have several limitations such as difficulties for selecting patients, the variable quality and quantity of tumor tissue, etc. Another method is to focus on the use of circulating cell-free DNA (cfDNA) from blood plasma as an alternative source of DNA to test for cancer-specific genetic alterations. This approach may also have issues with the quality and quantity for the sample, but is easier and safer to acquire than a new or archival tumor biopsy [4]. Our pilot project is an example of the latter. In either case, a germline DNA [1] sample is used as a baseline for comparison to identify mutations in the tumor/cfDNA.

[1]Germline DNA is a tissue derived from reproductive cells (egg or sperm) that become incorporated into the DNA of every cell in the body of the offspring

6.1.3 Pilot Project 1: Knee Segmentation and Registration Toolkit (KSRT)

This KSRT system is comprised of the first two pipelines stages of preprocessing and segmentation. We tested the initial parallel version on both the AWS cloud facility and the ARC cluster computer at NC State. The initial MPI-based [1] implementation of the code has been published on bitbucket and is publicly accessible at `http://wwwx.cs.unc.edu/ mn/KSRT/html/ksrtDoc.html`.

Preprocessing Phase

This phase of the application prepares test/train images by applying multiple sub-phases, such as bias-field correction, intensity-scaling, smoothing and downsampling. The application is data-parallel because each image is independent. Therefore, in our parallel implementation for the preprocessing phase, we assign one process per image.

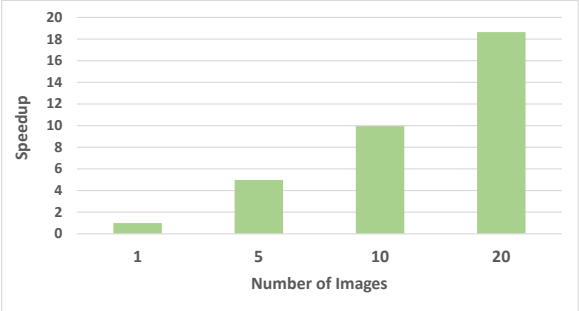


Figure 6.2: One Phase of Preprocessing - # of images per process = 1

To analyze the scaling behavior of the parallel algorithm, we tested the parallelized preprocessing algorithm under weak scaling [21]. Figure 6.2 indicates the speedup of the parallelized version over the serial base version. Speedup is defined as $S = \frac{T_s}{T_p}$, where T_s is the sequential execution time and T_p is the parallel execution time. 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. In this experiment, we increase the number of input images and number of processes. Because of the data-parallel nature of the algorithm, we can see almost a perfectly linear speedup in this experiment.

Segmentation Phase

This phase of the code includes several sub-tasks. In this phase, each test image is registered and segmented according to the number of atlases [2]. In Figure 6.3, there are three and four test images and atlas images, respectively. Due to data dependencies, parallelization of this phase is not as easy as during the preprocessing phase. We partitioned the pipeline into several phases based on the granularity of tasks. Therefore, the number of processes changes during the execution of the pipeline based on potential parallelization (e.g., the number of processes is three in phase 0 where each test image is prepared, and it is 12 in phase 1 where each test image needs to be registered to four atlas images).

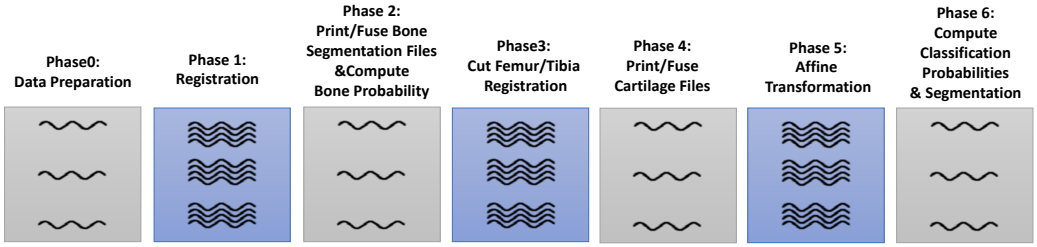


Figure 6.3: A Schematic of a Sample Segmentation Pipeline

6.1.4 Pilot Project 2: A Standard Scalable Pipeline for a Somatic Mutation Detection

Table 6.1: Execution Time: Processing Paired DNA Data

Stage	Avg. Time (sec)
Alignment (BWA)	4160 per lane
Deduplication (Picard)	1500 per lane
Realignment (GATK)	1769 per lane
Recaliberation (GATK)	758 per lane
PrintRead (GATK)	2218 per lane
Somatic Identifications (MuTect)	30063 per 16 lanes

Table 6.1 depicts the standard components of Figure 6.4 (e.g., bwa [35], GATK [39] and MuTect [12]) that comprise a second pipeline for somatic mutation detection.

[2] Atlases are reference images, in which regions of interest are delineated by experts

This prototyping work has been motivated by a simple preprocessing pipeline that we have developed using MPI and C++. The times required for each step to pre-process the data for one patient (paired data over 8 lanes) on one AWS c3.4xlarge instance are shown in Table 6.1.

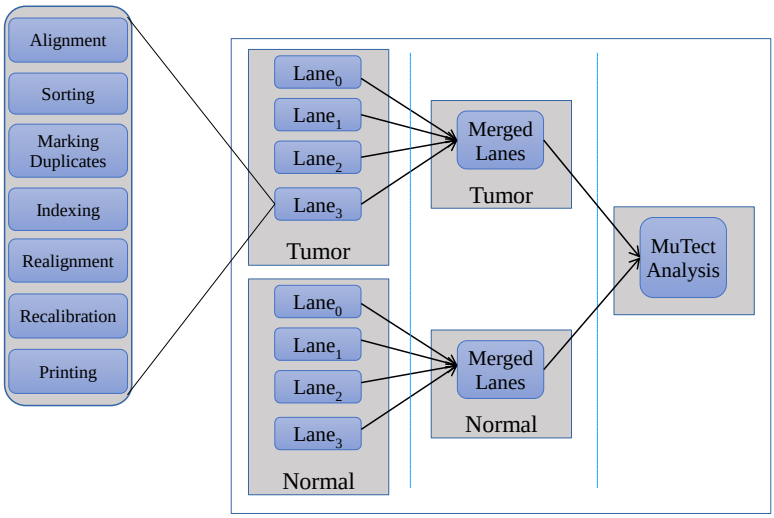


Figure 6.4: A Schematic of the Preprocessing Pipeline for Paired DNA Data

6.1.5 Scalability

To demonstrate the effectiveness of this system, both the KSRT and DNA pipelines have been implemented using our ElasticMedFlow framework.

Figure 6.5 shows the results of our experiments with a DNA pipeline for ElasticMedFlow. The x-axis denotes execution time, and and the y-axis denotes the configuration of the experiment, where P, N and N represent the number of patients, nodes and workers, respectively. We used the ARC cluster computer in this experiment, which is composed of 108 nodes with two AMD Opteron 6128 processors, each with 8 cores that communicate via QDR InfiniBand.

The red bar represents the sequential execution time, the blue bars represent time under strong scaling, and the green bars depict time under weak scaling. We observe that ElasticMedFlow with a single worker on a single node has very similar performance to the sequential version of the pipeline. The DNA pipeline has a maximum concurrency of 8 for approximately half of the pipeline, and a concurrency of 2 for the other half. This resulted in cutting the execution time almost into half when running with two workers on one node. Smaller, but still significant

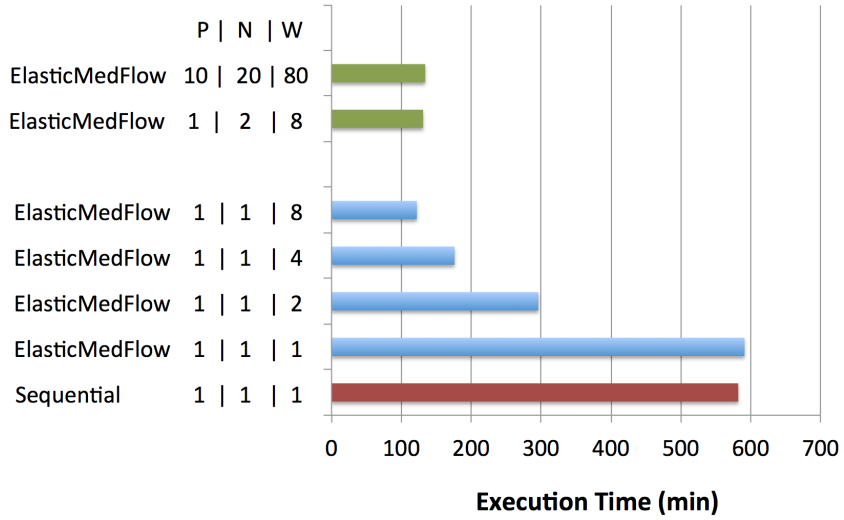


Figure 6.5: Scalability of our ElasticMedFlow System

performance gains where observed when increasing the number of workers to 4 and then 8. We also tested the DNA pipeline with 10 patients using 80 workers on 20 nodes (4 workers per node) and similar observed results to the single patient test with 8 workers on 2 nodes, which shows how well ElasticMedFlow parallelizes medical data analytics.

6.2 Chameleon, MPI4Py and ElasticMedFlow

To link Chameleon with ElasticMedFlow, several issues have to be overcome. Figure 6.6 shows the software layers. The application ElasticMedFlow calls MPI4Py commands, and these commands call PMPI wrappers of Chameleon. We first modified Chameleon and added new wrappers to cover MPI4Py's communication calls (e.g., MPI_Init_thread). After linking the application, we faced another problem. The replay accuracy of generated traces was not accurate enough (less than 70%).

We found that because ElasticMedFlow is running within the Python interpreter, after the Python interpreter calls any MPI4Py command, once the call is intercepted by Chameleon, Chameleon unwinds the stack frames and creates stack signatures. However, the stack frames did not have proper addresses to indicate the origin of calls in the application. Instead, most stack frames only contained functions from the interpreter code (e.g., PyEval_EvalFrameEx and and PyEval_EvalCodeEx).

This lack of information about the source code results in the same stack signatures for the master process and worker processes. Thus, Chameleon was not able to distinguish different

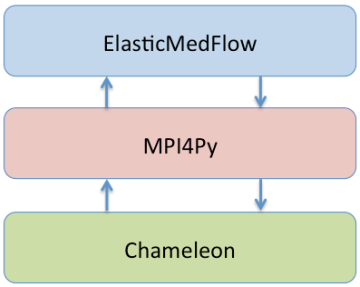


Figure 6.6: Application Layers

processes because of the seemingly identical callpaths, which resulted in merging events from master and workers incorrectly.

To address this problem, we wanted to avoid changing the application source code. Instead, we decided to use a shared buffer between MPI4Py and Chameleon. Every time the application calls an MPI command in MPI4Py, we invoke a Python "traceback" to obtain the Python application stack frames and populate stack signatures.

We used Cython to emit the address of Python objects called in MPI4Py and share the address of the string object the origin of calls with Chameleon (i.e., pointer of pointers). At the beginning of the application, in MPI_Init, we first send the address of the shared buffer to Chameleon using a customized MPI wrapper. On the MPI4Py side, we then modify the shared buffer by assigning the pointer to the string object containing Python stack frame information whenever MPI calls are issued.

On the Chameleon side, we dereference the shared buffer and use it in creating stack signatures. Such computation of stack signatures helped us form the two mains clusters (master and workers) and distinguish MPI events.

6.3 Conclusion

In this section, we introduced ElasticMedFlow, a generic framework for representing parallel medical pipelines, which executes across a cluster computer for a large number of patients with only few modifications from on application to another.

We used ElasticMedFlow, an application with small trace files and small sequences of events, to evaluate the capabilities of Chameleon in generating accurate traces.

As discussed in Chapter 5.2, Chameleon detects only two distinct traces (i.e., the trace of master process and the trace of one worker to represent all other workers), and the accuracy of the generated final trace exceeds 87% for ElasticMedFlow.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We proposed three unique clustering algorithms, each of which enables unique features and develops novel capabilities for tracing exemplified by ScalaTrace (i.e., CallPath+Parameter hierarchical clustering with only two 64-bit signatures, ACURDION with adaptive signature building and top K clustering, and Chameleon with online clustering and inter-compression).

The results of our experiments indicate that our clustering approaches provides significant reductions in performance overheads over ScalaTraceV2 making it suitable for extreme-scale computing.

Overall, this work is centered around scalable tracing of parallel applications. It contributes novel clustering approaches on communication trace compression.

7.2 Discussion and Future Work

Our proposed systems have the following weaknesses subject to future work:

(1) Inserting marker calls for Chameleon requires source code access/modifications. For some applications, the source code might not be available. In such cases, one could automatically insert the marker into binary files using software tools such as Pin [37].

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

Considering iterative scientific applications (most scientific codes), the main loop is executed by all processes (and marker insertion can be automated), and processes might diverge 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, but this might change with exa-scale computing.

(3) Leveraging the idle time for non-lead processes at interim execution points by utilizing dynamic voltage and frequency scaling (DVFS) would reduce power and make clustered tracing energy efficient as well.

REFERENCES

[1] MPI-2: Extensions to the message passing interface, July 1997.

[2] G. Aguilera, P. J. Teller, M. Taufer, and F. Wolf. A systematic multi-step methodology for performance analysis of communication traces of distributed applications based on hierarchical clustering. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–pp. IEEE, 2006.

[3] Allinea. The distributed debugging tool (DDT), http://www.allinea.com.

[4] K. Aung, R. Board, G. Ellison, E. Donald, T. Ward, G. Clack, M. Ranson, A. Hughes, W. Newman, and C. Dive. Current status and future potential of somatic mutation testing from circulating free dna in patients with solid tumours. *The HUGO journal*, 4(1-4):11–21, 2010.

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

[6] 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. URL `https://bitbucket.org/amirbahmani/elasticmedflow/`.

[7] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. Nas parallel benchmark results. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(1):43–51, 1993.

[8] D. Becker, F. Wolf, W. Frings, M. Geimer, B. J. N. Wylie, and B. Mohr. Automatic trace-based performance analysis of metacomputing applications. In *International Parallel and Distributed Processing Symposium*, pages 1–10, 2007.

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

[10] F. Bray, A. Jemal, N. Grey, J. Ferlay, and D. Forman. Global cancer transitions according to the human development index (2008–2030): a population-based study. *The lancet oncology*, 13(8):790–801, 2012.

[11] H. Brunst, M. Winkler, W. E. Nagel, and H.-C. Hoppe. Performance optimization for large scale computing: The scalable vampir approach. In *Computational Science-ICCS 2001*, pages 751–760. Springer, 2001.

[12] K. Cibulskis, M. S. Lawrence, S. L. Carter, A. Sivachenko, D. Jaffe, C. Sougnez, S. Gabriel, M. Meyerson, E. S. Lander, and G. Getz. Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nature biotechnology*, 31(3):213–219, 2013.

[13] L. Dalcin. mpi4py, 2007.

[14] Y. Ding, Y. Zhao, X. Shen, M. Musuvathi, and T. 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.

[15] Y. Dotsenko. *Expressiveness, programmability and portable high performance of global address space languages*. ProQuest, 2007.

[16] C. Fischbach. Fighting the war on cancer. Feb., 2015.

[17] T. Gamblin, B. R. De Supinski, M. Schulz, R. Fowler, and D. A. Reed. Clustering performance data efficiently at massive scales. In *International Conference on Supercomputing*, pages 243–252. ACM, 2010.

[18] 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.

[19] J. Gonzalez, J. Gimenez, and J. Labarta. Automatic detection of parallel applications computation phases. In *International Parallel and Distributed Processing Symposium*, pages 1–11.

[20] J. Gonzalez, K. Huck, J. Gimenez, and J. Labarta. Automatic refinement of parallel applications structure detection. In *Workshop on Large-Scale Parallel Processing*, pages 1680–1687, 2012.

[21] D. S. Gustavson. Scalable coherent interface. In *Proc. of the 34th IEEE Int'l Computer Conf. (COMPCON Spring'89)*, pages 536–544, Feb. 1989.

[22] A. Hoisie, O. Lubeck, and H. Wasserman. Performance analysis of wavefront algorithms on very-large scale distributed systems. In *Workshop on wide area networks and high performance computing*, pages 171–187. Springer, 1999.

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

[24] P. W. Jones, P. H. Worley, Y. Yoshida, J. White, and J. Levesque. Practical performance portability in the parallel ocean program (pop). *Concurrency and Computation: Practice and Experience*, 17(10):1317–1327, 2005.

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

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

[27] A. Knüpfer. A new data compression technique for event based program traces. In *Computational ScienceâĂŤICCS 2003*, pages 956–965. Springer, 2003.

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

[29] K. R. Koch, R. S. Baker, and R. 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.

[30] C. W. Lee and L. 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.

[31] C. W. Lee, C. Mendes, and L. V. Kalé. Towards scalable performance analysis and visualization through data reduction. In *International Parallel and Distributed Processing Symposium*, pages 1–8.

[32] C. W. Lee, C. Mendes, and L. V. Kalé. Towards scalable performance analysis and visualization through data reduction. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[33] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. De Supinski, B. P. Miller, and M. Schulz. Benchmarking the stack trace analysis tool for bluegene/l. In *PARCO*, pages 621–628, 2007.

[34] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. De Supinski, M. Legendre, B. P. Miller, M. Schulz, and B. Liblit. Lessons learned at 208k: towards debugging millions of cores. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–9, 2008.

[35] H. Li and R. Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[36] G. Llort, J. Gonzalez, H. Servat, J. Gimenez, and J. Labarta. On-line detection of large-scale parallel application's structure. In *International Parallel and Distributed Processing Symposium*, pages 1–10, 2010.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[38] 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.

[39] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data. *Genome research*, 20(9):1297–1303, 2010.

[40] O. Y. Nickolayev, P. C. Roth, and D. A. Reed. Real-time statistical clustering for event trace reduction. *International Journal of High Performance Computing Applications*, 11(2): 144–159, 1997.

[41] M. Noeth, P. Ratn, F. Mueller, M. Schulz, and B. R. de Supinski. Scalatrace: Scalable compression and replay of communication traces for high-performance computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710, 2009.

[42] H.-S. Park and C.-H. Jun. A simple and fast algorithm for k-medoids clustering. *Expert Systems with Applications*, 36(2, Part 2):3336 – 3341, 2009. ISSN 0957-4174. doi: http://dx.doi.org/10.1016/j.eswa.2008.01.039. URL `http://www.sciencedirect.com/science/article/pii/S095741740800081X`.

[43] M. Parsian. *Data Algorithms*. O'Reilly Media, 2015. ISBN 978-1-4919-0618-7.

[44] D. Pelleg, A. W. Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *ICML*, pages 727–734, 2000.

[45] D. A. Reed, P. C. Roth, R. A. Aydt, K. A. Shields, L. F. Tavera, R. J. Noe, and B. W. Schwartz. Scalable performance analysis: The pablo performance analysis environment.

In *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, pages 104–113. IEEE, 1993.

[46] T. Schneider, R. Gerstenberger, and T. Hoefler. Application-oriented ping-pong benchmarking: how to assess the real communication overheads. 2013.

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

[48] R. Smith, P. Jones, B. Briegleb, F. Bryan, G. Danabasoglu, J. Dennis, J. Dukowicz, C. Eden, B. Fox-Kemper, P. Gent, et al. The parallel ocean program (pop) reference manual: ocean component of the community climate system model (ccsm). *Los Alamos National Laboratory, LAUR-10-01853*, 2010.

[49] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Supercomputing*, Nov. 2002.

[50] R. Software. Totalview debugger, http://www.roguewave.com/products/totalview.aspx.

[51] T. A. C. C. Stampede. http://www.tacc.utexas.edu/resources/hpc/stampede, 2014.

[52] R. Susukita, H. Ando, M. Aoyagi, H. Honda, Y. Inadomi, K. Inoue, S. Ishizuki, Y. Kimura, H. Komatsu, M. Kurokawa, K. J. Murakami, H. Shibamura, S. Yamamura, and Y. Yu. Performance prediction of large-scale parallell system and application using macro-level simulation. In *Supercomputing*, 2008.

[53] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto. Scalable fine-grained call path tracing. In *International Conference on Supercomputing*, pages 63–74. ACM, 2011.

[54] 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.

[55] 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, Sept. 2003.

[56] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *ACM SIGPLAN Notices*, volume 36, pages 123–132. ACM, 2001.

[57] X. Wu and F. Mueller. Scalaextrap: Trace-based communication extrapolation for spmd programs. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 113–122. ACM, 2011.

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

[59] X. Wu and F. Mueller. Elastic and scalable tracing and accurate replay of non-deterministic events. In *International Conference on Supercomputing*, June 2013.

[60] X. Wu, F. Mueller, and S. Pakin. Automatic generation of executable communication specifications from parallel applications. In *Proceedings of the international conference on Supercomputing*, pages 12–21. ACM, 2011.

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

[62] J. C. Yan and M. A. Schmidt. Constructing space-time views from fixed size trace filesâĂŤgetting the best of both worlds. *Advances in Parallel Computing*, 12:633–640, 1998.

[63] T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Supercomputing*, Nov. 2005. doi: http://dx.doi.org/10.1109/SC.2005.20.

[64] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with jumpshot. *International Journal of High Performance Computing Applications*, 13(3): 277–288, 1999.

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

[66] J. Zhai, W. Chen, and W. 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.

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

[68] I. Zhukov and B. 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.

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