Efficient Clustering for Ultra-Scale Application Tracing ☆

Amir Bahmani[a], Frank Mueller[a,*]

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

Abstract

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. But not only applications are "scalability challenged", current tracing toolsets also fall short of exascale requirements for low background overhead since trace collection for each execution entity is becoming infeasible. One effective solution is to cluster processes with the same behavior into groups. Instead of collecting performance information from each individual node, this information can be collected from just a set of representative nodes. This work 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 based on signatures further generalizes to a novel multi-metric clustering technique with unprecedented low overhead.

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

1. Introduction

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

☆An earlier version of this paper appeared at ICS'14 [1]. This journal version extends the earlier paper by proposing a new method at the trace creation level called Filtering Function. The paper also compares and reports the performance of clustering under different trace creation approaches. This work was supported in part by NSF grants CCF-1217748 and CNS-0958311.

*Corresponding author
Email addresses: abahman@ncsu.edu (Amir Bahmani), mueller@cs.ncsu.edu (Frank Mueller)

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 Vampir [2], or preserve only aggregated statistical trace information to conserve the size of trace files, as in mpiP [3].

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 representative nodes) per cluster group.

This paper proposes 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 [4], 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 [5, 6]). 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.

2. Background

Our work builds on ScalaTrace as an MPI tracing toolset. 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 [7]. ScalaTrace not only captures the communication time per each MPI event, but also captures the execution time between MPI events and stores it in histograms. Consider the example in the following code snippet:

for $i = 0 \rightarrow 1000$ **do**
  for $k = 0 \rightarrow 100$ **do**
    $MPI\_Send(...)$;
    $MPI\_Recv(...)$;
  end for
  $MPI\_Barrier(...)$
end for

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

ScalaTrace has the following three main properties: (1) ScalaTrace 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 [5]. Consider Fig. 1 with relative encoding of nodes 5 and 9 in terms of communication end-points -4, -1, $+1$ and $+4$, i.e., these nodes have identical *relative* communication end-points.

(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 [5].

(3) ScalaTrace provides communication group encoding: ScalaTrace 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,

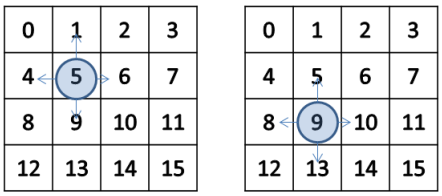
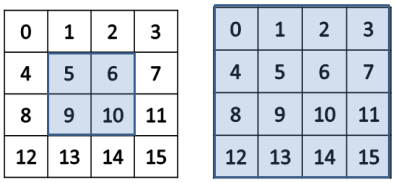


Figure 1: Communication End-point Encoding

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 [8]. In Fig. 2(a), the shaded nodes are presented as ranklist ⟨*2 5 2 4 2 1*⟩, and in Fig. 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).



(a) (b)

Figure 2: Ranklists for Communication Group

3. A Novel Clustering Algorithm

This section details design and implementation of the Call-path+Parameter clustering and reference clustering algorithms. Call-path+parameter clustering has two main phases. A first call-path clustering phase discovers processes with different numbers or sequences of events, and a second phase distinguishes processes with the same call-path cluster but different event parameters. As noted previously, the reference signature is the uncompressed version of the Call-path+Parameter signatures.

3.1. Call-Path Clustering

Figure 3 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.

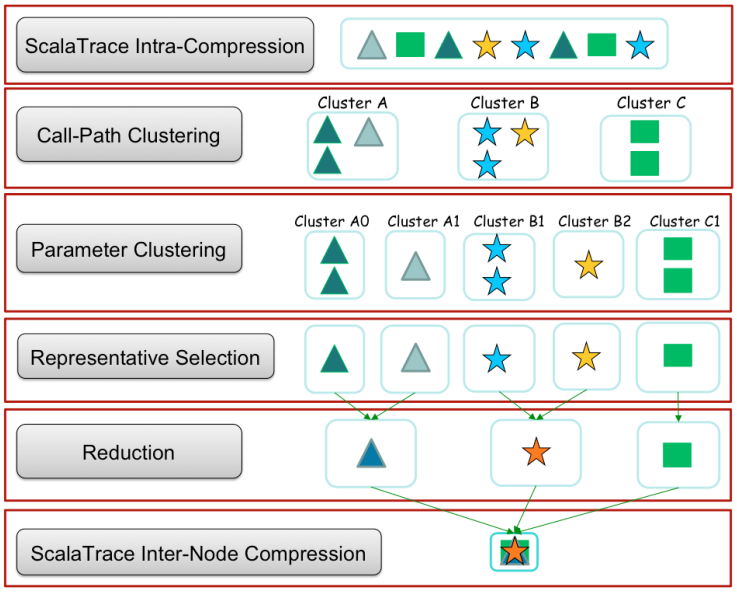


Figure 3: Overview of Proposed Clustering Algorithm

Table 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

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 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 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 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 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 representative 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 representative 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 representative, 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 2):
1) *Default version*: the representatives are merged within each main cluster. Subclusters with different parameters, such as A1 and A2, are merged pairwise linearly at a node within a radix tree (facilitating relative encoding matches [5]) so that the overall reduction over the tree is logarithmic in complexity. For instance, at the reduction phase in Fig. 3, 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 [5] 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 representatives with different call-path signatures have to participate in this operation. During the last phase of Fig. 3, 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 representative sends only different parameters and filters out value-identical parameters. Details of our implementation are provided in Section 5.

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 subclusters 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

Table 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

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 Signature

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 5, 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.

4. 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 [9] and Sweep3D [10]. 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 [11] 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.

5. 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. 4 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 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.

Table 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

Fig. 4 and Table 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 pro-

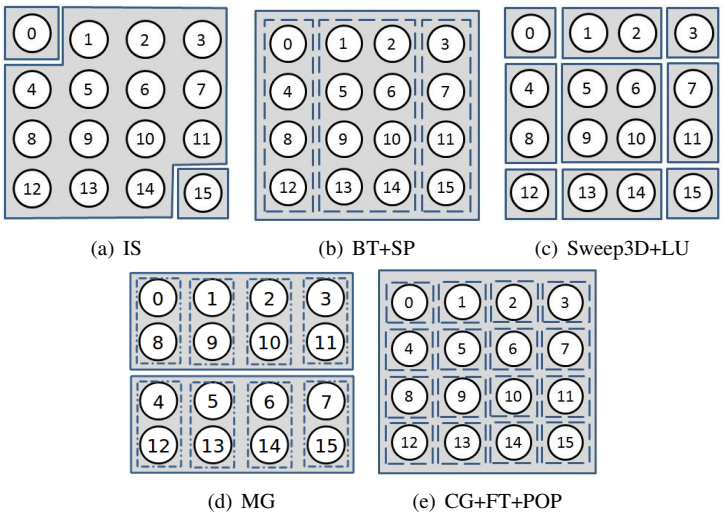


Figure 4: Topology of Different Benchmarks for 16 Processes Through Call-path+Parameter Clustering

cess 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 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 [12]. Similar to Sweep3D, LU is also a stencil code [13] that creates nine different main clusters.

(4) The number of main Multi-Grid (MG) clusters is not constant; as shown in Fig. 4, 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 [9]. This behavior is due to the

following two main reasons [14]: (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. 6 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 (as a user-provided plug-in function for the CG code):

if $npcols.eq.nprows$ **then**
  $exch\_proc = mod(me, nprows) * nprows + me/nprows$
else
  $exch\_proc = 2 * (mod(me/2, nprows) * nprows + me/2/nprows) + mod(me, 2)$
end if

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 4 indicates that once this function is supplied, the number of sub-clusters decreases significantly.

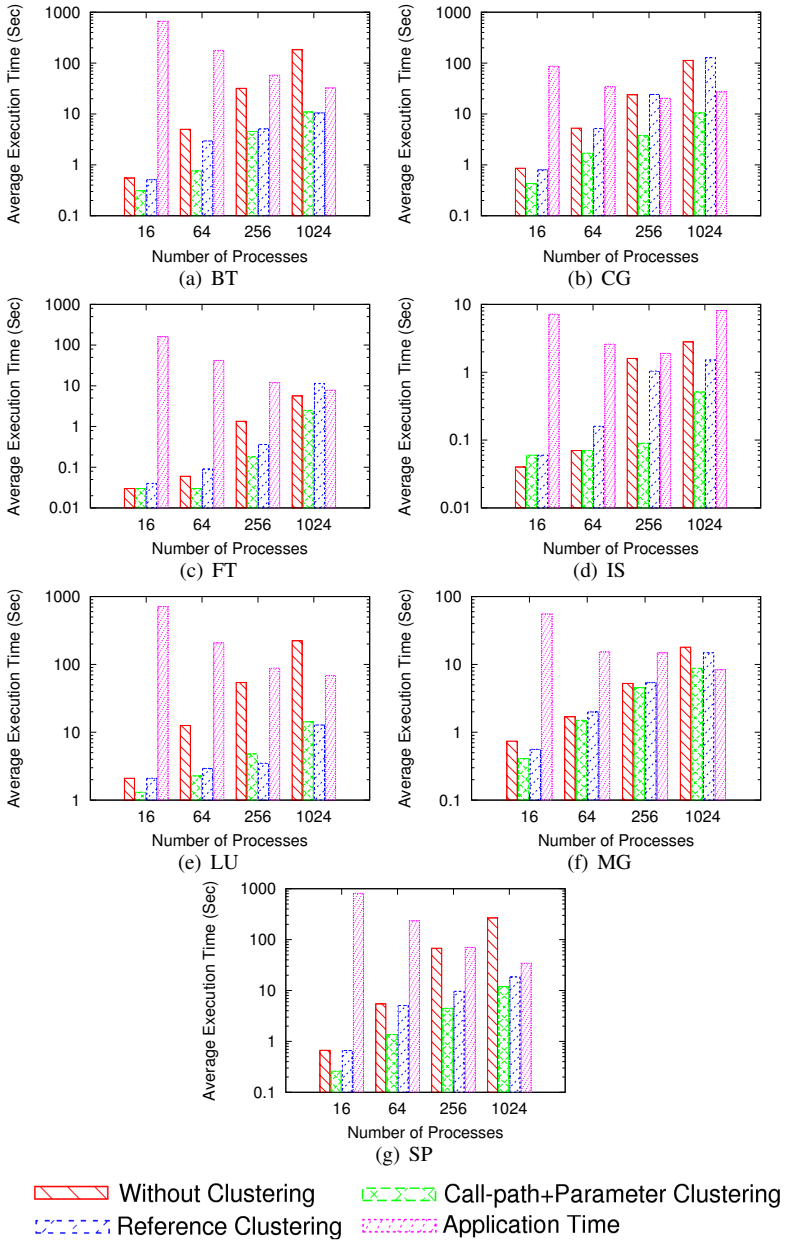


Figure 5: Execution Times for Inter-Node Compression Variants and Entire NAS Benchmarks(Strong Scaling) — Nodes/Tasks=1/16

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 (as a user-provided plug-in function for the FT code):

if $np.eq.1$ **then**
  $np2 = 1$
else if $np.le.nz$ **then**
  $np2 = np$
else
  $np2 = np/nz$
end if
$me1 = me/np2$
$me2 = (me\%np2)$

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 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.

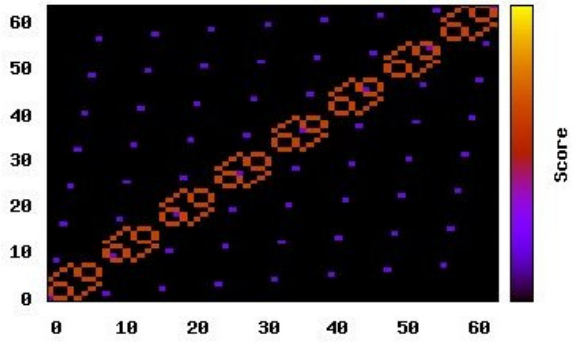


Figure 6: CG Communication Matrix

Table 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 [15].

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 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 5.2, 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.

5.1. 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. 5 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 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

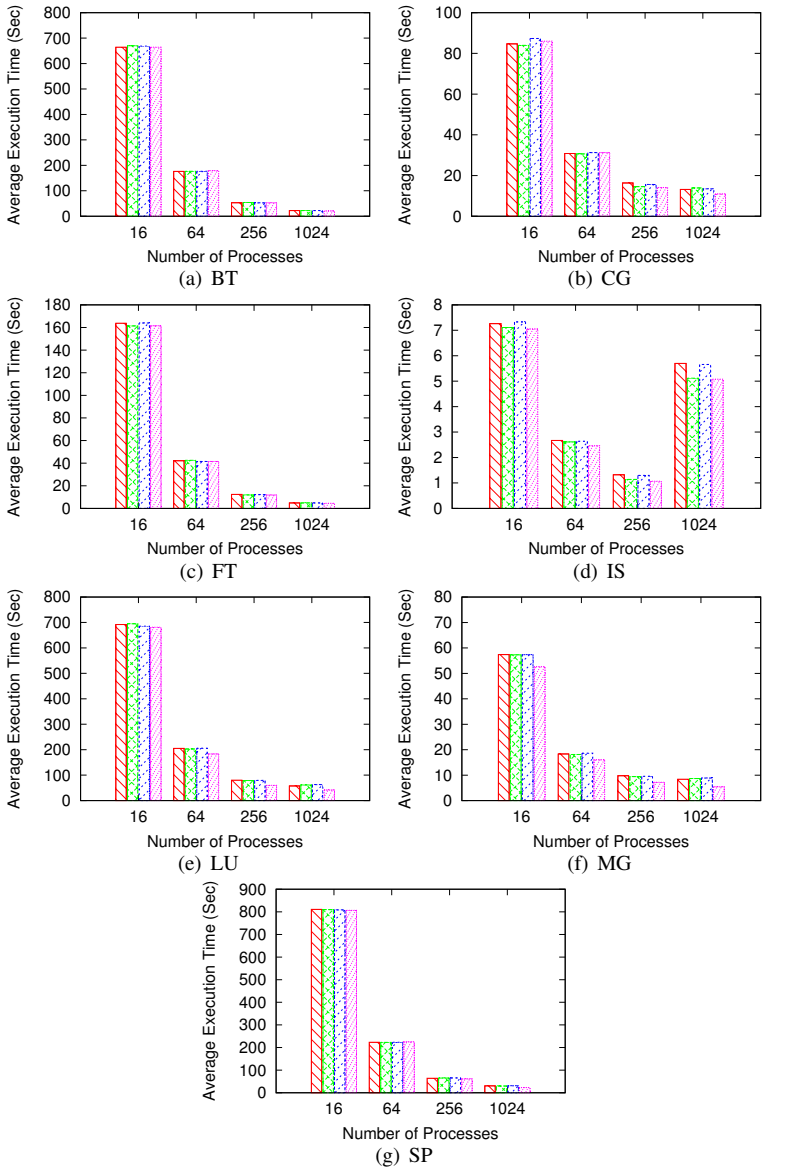


Figure 7: Replay Times (Strong Scaling) — Nodes/Tasks=1/16

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.

Table 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

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 computation time as sleeps [16]. 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. 7 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 representative) 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.

5.2. 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

Table 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

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 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 [8].

As Table 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. 8 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. 9 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), which 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.

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% accurate), 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. We only report weak scaling for POP since strong scaling has similar results. The challenge lies in intra-node compression and is agnostic of strong/weak scaling, so reporting one seems sufficient (as the other does not add anything).

Fig. 10 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.

5.3. 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. 11(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. 11(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 representatives), we conducted another experiment. Fig. 12 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 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.

Table 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

5.4. 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 7 shows the space complexity of all benchmarks for P=256.

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

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

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.

Table 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 13 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 6) and MG (see discussion around Fig. 4), 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.

6. Related Literature

A commonly utilized tracing tool for MPI communication is Vampir [2], 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 [17] 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 [18] and [19], features k-means clustering to select representative data for migration of objects in $CHARM++$. A density-based clustering analysis was proposed in [20], [21] and [22] 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 [23], 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 [24]. 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 [25], 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) [27]. 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 [28] and DDT [29] 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 [30] 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 [31] 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. [32] 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. [33] 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. [34] propose another clustering algorithm using the Pablo performance instrumentation library [35] with a real-time statistical clustering infrastructure. The standard Pablo instrumentation software captures dynamic performance data via instrumented source code. Yan et al. [36] 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.

Knüpfer et al. [37] 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.

7. Conclusion and Future Work

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

We currently apply the clustering algorithm at the end of program execution. However, if we were to group processes with the same execution behavior at interim execution points, e.g., at timestep boundaries of scientific codes, inter-node compression could be performed online. This would reduce the execution time by overlapping the I/O and computation time. Such online clustering is the focus of our ongoing work beyond the scope of this paper. Moreover, in this work, we only considered perfectly matched signatures; basically, Call-path+Parameter clustering creates complete traces with all parameters and events so that there is no problem at the reply level. Ongoing work attempts to approximate matching and to support probabilistic replay.

References

[1] A. Bahmani, F. Mueller, Scalable performance analysis of exascale mpi programs through signature-based clustering algorithms, in: ICS, 2014.

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

[3] J. S. Vetter, M. O. McCracken, Statistical scalability analysis of communication operations in distributed applications, in: ACM SIGPLAN Notices, Vol. 36, ACM, 2001, pp. 123–132.

[4] X. Wu, F. Mueller, Elastic and scalable tracing and accurate replay of non-deterministic events, in: International Conference on Supercomputing, 2013.

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

[6] X. Wu, F. Mueller, S. Pakin, Automatic generation of executable communication specifications from parallel applications, in: Proceedings of the international conference on Supercomputing, ACM, 2011, pp. 12–21.

[7] J. Marathe, F. Mueller, T. Mohan, B. R. de Supinski, S. A. McKee, A. Yoo, METRIC: Tracking down inefficiencies in the memory hierarchy via binary rewriting, in: International Symposium on Code Generation and Optimization, 2003, pp. 289–300.

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

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

[10] K. R. Koch, R. S. Baker, 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) (1992) 198–199.

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

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

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

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

[15] 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.

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

[17] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, L. Adhianto, Scalable fine-grained call path tracing, in: International Conference on Supercomputing, ACM, 2011, pp. 63–74.

[18] C. W. Lee, L. V. Kalé, Scalable techniques for performance analysis, Parallel Programming Laboratory, Department of Computer Science, University of Illinois, Urbana-Champaign, Tech. Rep (2007) 07–06.

[19] C. W. Lee, C. Mendes, L. V. Kalé, Towards scalable performance analysis and visualization through data reduction, in: International Parallel and Distributed Processing Symposium, pp. 1–8.

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

[21] J. Gonzalez, K. Huck, J. Gimenez, J. Labarta, Automatic refinement of parallel applications structure detection, in: Workshop on Large-Scale Parallel Processing, 2012, pp. 1680–1687.

[22] J. Gonzalez, J. Gimenez, J. Labarta, Automatic detection of parallel applications computation phases, in: International Parallel and Distributed Processing Symposium, pp. 1–11.

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

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

[25] T. Gamblin, B. R. De Supinski, M. Schulz, R. Fowler, D. A. Reed, Clustering performance data efficiently at massive scales, in: International Conference on Supercomputing, ACM, 2010, pp. 243–252.

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

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

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

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

[30] G. L. Lee, D. H. Ahn, D. C. Arnold, B. R. De Supinski, M. Legendre, B. P. Miller, M. Schulz, 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, 2008, pp. 1–9.

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

[32] G. Aguilera, P. J. Teller, M. Taufer, 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, IEEE, 2006, pp. 8–pp.

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

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

[35] D. A. Reed, P. C. Roth, R. A. Aydt, K. A. Shields, L. F. Tavera, R. J. Noe, B. W. Schwartz, Scalable performance analysis: The pablo performance analysis environment, in: Scalable Parallel Libraries Conference, 1993., Proceedings of the, IEEE, 1993, pp. 104–113.

[36] J. C. Yan, M. A. Schmidt, Constructing space-time views from fixed size trace filesgetting the best of both worlds, Advances in Parallel Computing 12 (1998) 633–640.

[37] A. Knüpfer, A new data compression technique for event based program traces, in: Computational ScienceICCS 2003, Springer, 2003, pp. 956–965.

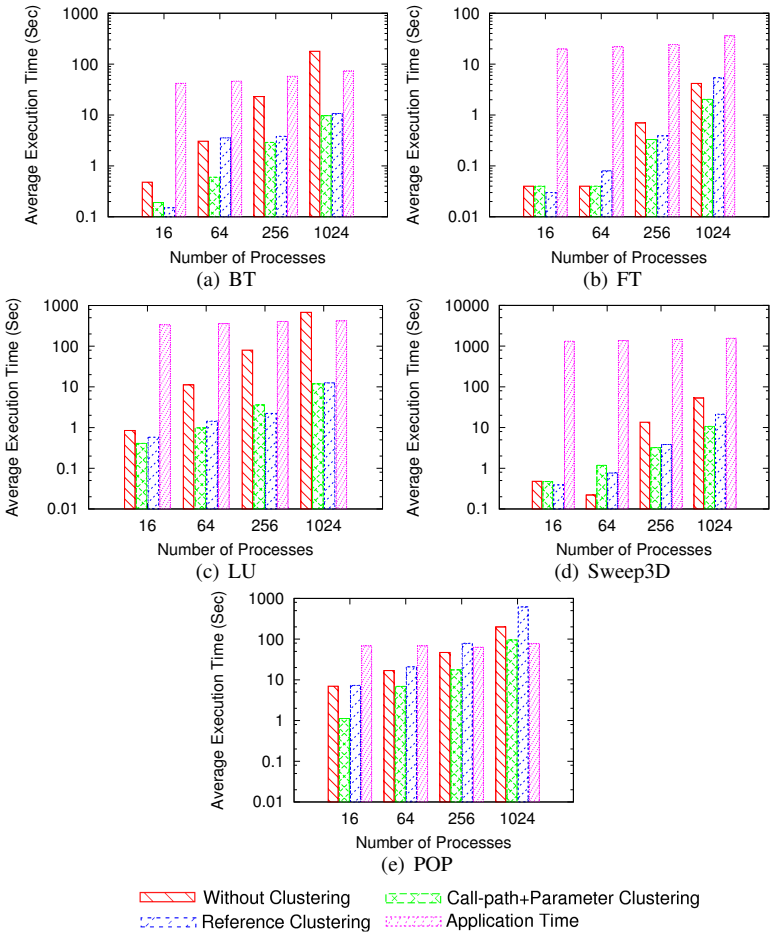


Figure 8: Execution Times for Inter-Node Compression Variants and Entire Application (Weak Scaling) — Nodes/Tasks=1/16

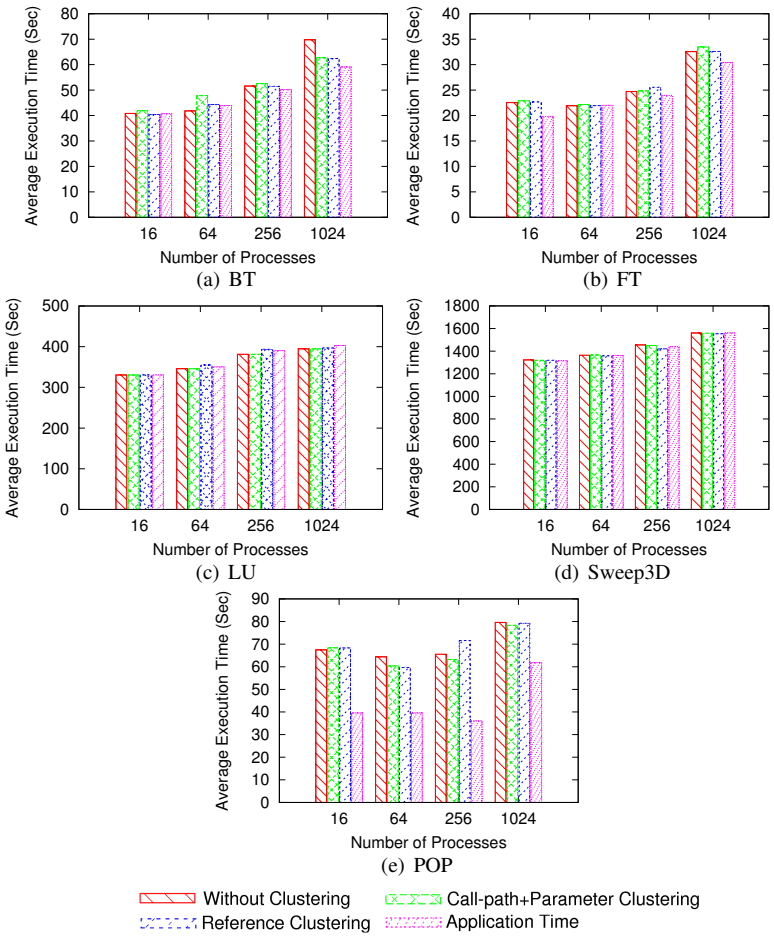


Figure 9: Replay Times (Weak Scaling) — Nodes/Tasks=1/16

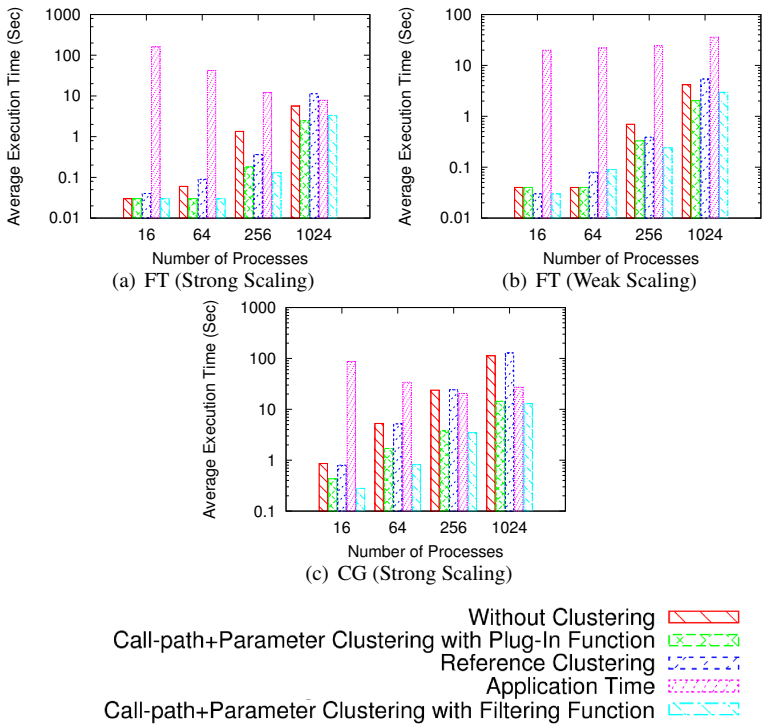


Figure 10: Execution Times for Inter-Node Compression Variants and Entire Application (Weak Scaling) — Nodes/Tasks=1/16

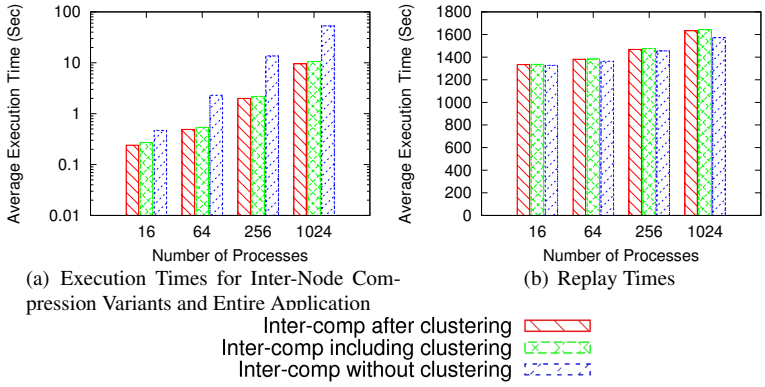
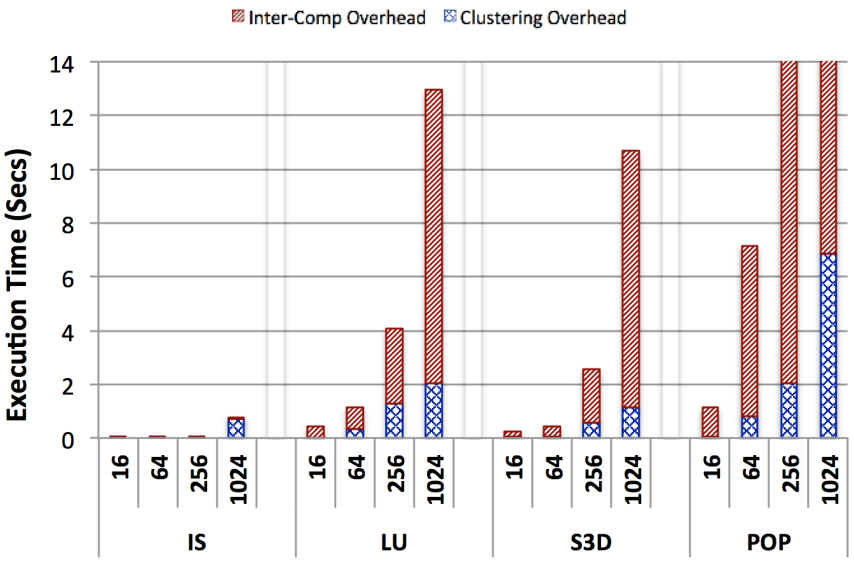
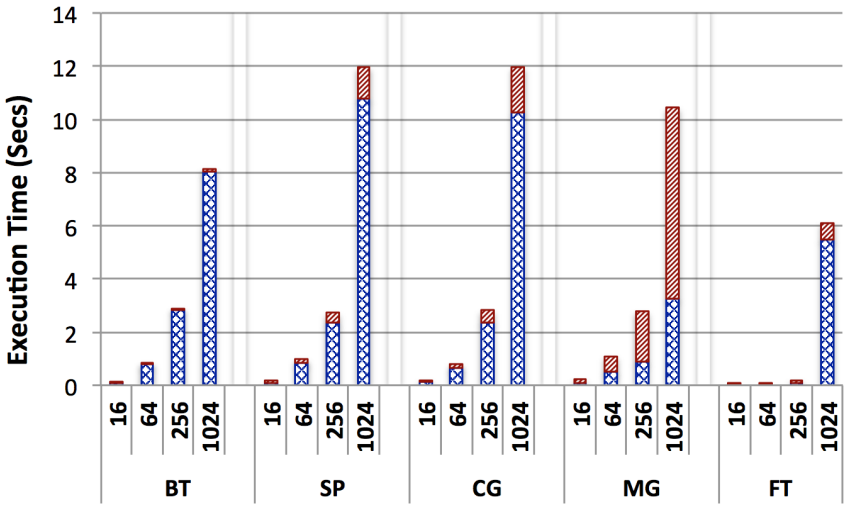


Figure 11: Sweep3D Execution and Replay Times (Weak Scaling) — Nodes/Tasks=1/16



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



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

Figure 12: Clustering Cost vs. Inter-Node Compression Cost (Strong Scaling for all except LU and Sweep3D) — Nodes/Tasks=1/16

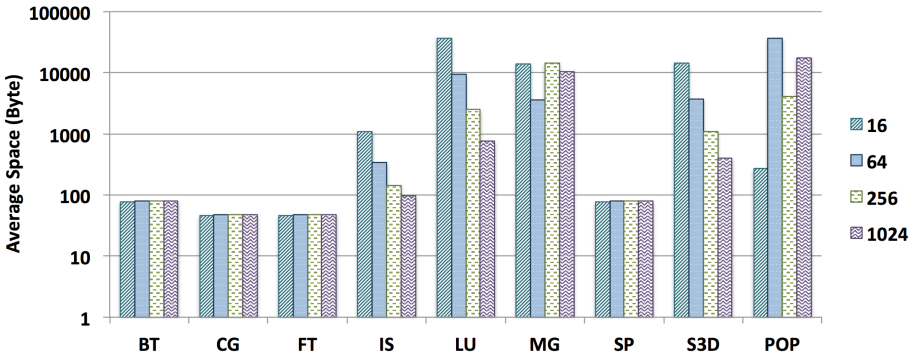


Figure 13: Space Complexity — Strong Scaling, except for Sweep3D and POP (Weak Scaling)