2016 IEEE International Parallel and Distributed Processing Symposium

Structural Clustering: A New Approach to Support Performance Analysis at Scale

Matthias Weber*, Ronny Brendel*, Tobias Hilbrich*, Kathryn Mohror†, Martin Schulz†, and Holger Brunst*
*Technische Universität Dresden, Germany
{matthias.weber, ronny.brendel, tobias.hilbrich, holger.brunst}@tu-dresden.de
†Lawrence Livermore National Laboratory, USA
{kathryn, schulzm}@llnl.gov

***Abstract*—The increasing complexity of high performance computing systems creates high demands on performance tools and human analysts due to an unmanageable volume of data gathered for performance analysis. A promising approach for reducing data volume is classification of data from multiple processes into groups of similar behavior to aid in analyzing application performance and identifying hot spots. However, existing approaches for structural and temporal classification of performance data suffer from lack of scalability or produce misleading results.**

To address this problem, we present a novel and effective structural similarity measure to efficiently classify data from parallel processes and introduce a method for efficient storage of the classified data. Using four examples, we show how existing performance analysis techniques benefit from our structural classification. Finally, we present a case study with 15 applications on up to 65,536 parallel processes that demonstrates the generality and scalability of our classification approach.

I. Introduction

Performance analysis is a central part of the software life cycle for High Performance Computing (HPC) applications. However, performance analysis can be extremely challenging, in particular for long-running, large-scale jobs due to the potentially very large volume of performance data collected for analysis. Performance tools vary in how much information they collect and ultimately retain, ranging from large, highly-detailed event traces to compact, high-level profiles. Traces are sequential, typically time-stamped, records of execution events, and are useful for uncovering the root causes of performance problems with a temporal component. On the other hand, profiles typically provide a summary of execution information, such as the total amount of time spent in each function, or the average number of bytes transferred in a particular communication operation, and are useful for understanding the overall picture of an execution.

Tracing and profiling each have their purpose and benefits, but both also have drawbacks. In tracing, all details are retained, but that makes analysis extremely challenging due to the potentially large numbers of events for a large number of processes. Profiles only contain summary data, but because the data is reduced naively, performance differences across time or processes can be lost, making understanding performance problems impossible. In each case, the amount of data collected serves the purpose of the method, but also presents challenges for analysis. We need to find a middle ground, a way to intelligently reduce the amount of data for analysis, such that trace analysis is tractable, while at the same time selectively aggregating data in targeted profiles with less information loss.

In this paper, we present a new approach that finds such a middle ground and complements existing methods for analysis of traces and profiles, while retaining their respective strengths. To achieve this goal, our approach performs grouping of performance data, and automatically compares and categorizes performance measurements of parallel applications. In particular, we develop a lightweight system that pre-clusters structurally similar processes into a few distinct groups in preparation for more advanced analyses, such as a time-based clustering, alignment-based comparison [24], or comparison based on profiles of different executions. By providing only a small number of groups from possibly thousands of processes we reduce the workload imposed on successive analysis techniques and focus their usage on meaningful subsets of processes, resulting in quicker detection of performance problems.

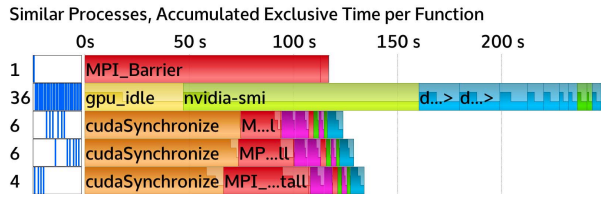
Our approach is highly scalable, and considers structural differences between processes, thereby extracting needed information to understand performance differences. Existing approaches for comparing performance across executions [4], [13], [14], [19]–[21], [24] either lack scalability or provide no capability to compare executions with structurally different processes within a single execution. At the same time, approaches that compare processes within a program run [4], [11], [13] either do not consider program structure or lack scalability.

Our contributions in this paper include:

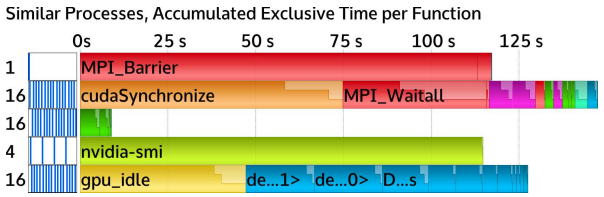
- A similarity measure based on execution structure,
- A method to efficiently group processes using our similarity measure,
- An extended similarity measure that discerns a special class of differences, e.g. parent-child relations between processes,
- Usage scenarios that detail how our grouping approach can aid existing analysis approaches, and
- An application study with performance measurements from a wide range of applications at up to 65,536 processes.

We found that our structural pre-clustering approach can greatly reduce the amount of data needed for application analysis. For the traces we analyzed in our application study, most could be reduced to less than 10 process traces, down from the original process counts, e.g., 16,384 for PEPC.

1530-2075/16 $31.00 © 2016 IEEE
DOI 10.1109/IPDPS.2016.27



(a) Purely time-based clustering: Processes with no similarity, e.g., threads and CUDA streams, are clustered together.



(b) Structural pre-clustering: Processes with no similarity are in distinct groups and not clustered together.

Fig. 1. PIConGPU Analysis. (a) Time-based clustering yields unsatisfactory results. (b) Structural pre-classification yields higher quality results.

II. Motivating Example

To motivate the importance of considering code structure in our approach, we consider the problem of comparing the performance of processes[1] with different execution structure. Figure 1 shows two summary profiles for the same execution of the PIConGPU application using two different clustering methods: traditional time-based clustering (top), and our structural pre-clustering (bottom). PIConGPU uses a combination of MPI processes, threads, and CUDA [7] streams. In the profiles, each bar represents a cluster of processes determined to have similar behavior, with a number next to each bar that indicates the number of processes in the cluster. The colors represent different functions, and the relative size of each color block gives the amount of time spent in that function for the processes in each cluster.

Figure 1(a) depicts the current state-of-the-art, a purely time-based clustering from the Vampir [4] suite. In this approach, processes are clustered solely according to the time spent in the most time consuming functions without taking program structure into account. This leads to unintuitive results, since the clustering does not consider the differences stemming from PIConGPU using MPI, threads, and CUDA: without this distinction, the approach classifies threads and CUDA streams as similar, as shown in the second bar in Figure 1(a), which prohibits comparing their differences. Additionally, the approach separates MPI processes into three different clusters (bottom three bars in Figure 1(a)), which further complicates any comparative analysis.

Our approach adds this missing structural classification as a pre-clustering step, before applying time-based clustering. Figure 1(b) highlights how our pre-clustering step results in higher quality time-based clusters: now, data from structurally different processes is classified into individual clusters. The first cluster contains one MPI process, as it is structurally different from the other MPI processes. The remaining 16 MPI processes are placed in the second cluster. The third cluster groups the threads that render simulation output, and the fourth cluster contains GPU measurement management threads. Finally, all CUDA streams are grouped together in the fifth cluster.

This example shows that our structural pre-clustering step results in a more logical and intuitive grouping of processes in the final time-based clustering. Thus, it can facilitate comparative analysis performed by analysts, potentially greatly reducing the time to discover root causes of performance problems.

[1] In this paper, for brevity we use the term *process* to refer to any possible processing element of a parallel application, which may be an MPI process, a thread, a CUDA stream, or a different type of control flow.

III. Related Work

While our novel structural pre-classification method is a contribution in and of itself, its true benefit is that it can significantly enhance the results of existing performance data comparison approaches. Here, we provide an overview of these existing approaches, and later, in Section VIII we detail how our method can benefit these existing approaches.

The tools eGprof [19], Open|SpeedShop [20], Cube [21], and the prototype by Karavanic and Miller [14] all compare profiles of executions. While eGprof can compare the call graphs of two executions, the parallelism that eGprof can handle is limited. Karavanic and Miller present an experiment management system and provide operators for structural differences. Their tool can analyze application performance across multiple execution environments and program versions. Cube specializes this approach with an algebra for comparing, integrating, and summarizing performance data from multiple sources. It provides basic arithmetic operations to merge, subtract, and average data from different experiments.

Both the Intel Trace Analyzer [13] and Vampir [4] can compare trace files. The Intel Trace Analyzer displays two trace files together in one view and additionally computes differences and speedups based on profile information. Similarly, Vampir enables viewing multiple traces side by side. Additionally, it provides comparison for the processes of a single execution, as introduced in Figure 1.

An extension of the CEPBA tool suite [11] and a trace alignment approach [24] both target a detailed comparison of the behavior that is included in tracing information. The CEPBA tool suite extension describes a tool that characterizes computation phases. It clusters these periods with a common clustering algorithm. The result is a classification of phases that differ in instructions-per-second rates. The second trace alignment approach [24] introduces several metrics for trace comparison. One of them describes the similarity of two processes' call stacks over time. These metrics build on sequence alignment techniques for parallel traces [23]. In contrast to our approach, aligning traces is computationally much more expensive, as we detail in Section VIII. Consequently, trace alignment is limited to pairs of processes in practice, as opposed to the lightweight classification that we contribute for large numbers of processes.

Our approach relies on structural information that we derive from caller-callee relationships and is related to trace compression approaches by Mohror and Karavanic [16], by Knüpfer [15], and an extension of ScalaTrace [18]. Mohror and Karavanic [16] evaluate similarity metrics in the context of trace file compression. The evaluated metrics identify similar regions, of which

only one representative is then stored. Knüpfer [15] proposes the *Compressed Complete Call Graph*, a data structure that behaves like a trace, but internally uses a lossy compression scheme. The technique is also used to highlight repeating function call patterns. Finally, the ScalaTrace extension employs comparison techniques to achieve lossless compression of communication traces. While these approaches also consider structural behavior, they target a reduction in trace file size, as opposed to our lightweight classification of processes.

IV. Methodology Overview

A good criterion for structural comparison of processes is information on the invocation of functions, disregarding timing. This information is available in almost any type of performance profile. Information on function invocation can be represented in *call trees*. These trees represent merged call stacks of all functions that a process invokes. Differences between the call trees of two processes provide a measure of similarity. Comparing call trees, however, is computationally expensive and small differences close to the root of a call tree yield low similarity, even if large parts of the trees are equal.

Thus, in this work, we use a simplification of the information contained in a call tree, the so-called *call matrix*, made up out of caller/callee function pairs. The size of a call matrix is independent of application runtime or the process count and is limited to the squared number of existing functions. For most applications, this number is less than 1,000, which makes the call matrix a powerful approach for comparison. Based on the call matrix, we introduce a similarity measure for comparing processes.

Scalable structural process classification not only requires a similarity measure based on a call matrix, but also requires a scheme that is efficient even for large process counts. Since a pair-wise comparison of all processes is highly inefficient, we use *concept lattices* that stem from *formal concept analysis* [9] to store and compute groups of similar processes. This approach removes the need for pair-wise process comparison. The result of using the concept lattice is a classification of all processes into groups with structurally identical behavior, along with a relation between these groups. In practice, we find our method generates a low number of groups and therefore provides a compact representation for structural information that can be computed scalably for large-scale executions.

Based on the classification from the concept lattice, we then compute a *similarity matrix* that contains similarity values between process groups for comparison. We developed two similarity measures in this work. Our first measure, *pairsim*, compares the overall behavior of two processes. A second measure, *pairsub*, highlights parent-child relations between processes and sharpens the understanding of similarity that we provide. These similarity measures are described in detail in Sections V and VII.

Throughout this paper, we use two running examples to explain our concepts. These examples are real applications and use multiple process types in their runs. The first is a run of the BT [3] fluid dynamics simulation kernel with 16 MPI processes, which use 15 OpenMP threads each. The second is a particle-in-cell equation solver named PIConGPU [6]. It uses 17 MPI processes, 20 threads spawned with the Pthreads API, and 16 CUDA streams on GPGPU devices. The introductory example of Figure 1 depicts process clusterings for this second example.

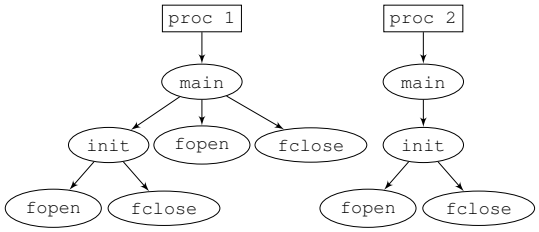


Fig. 2. Call trees for two example processes.

V. Determining Similarity

In this section, we outline our definitions of similarity for the purposes of our process clustering. We begin with general definitions and follow with development of our first similarity measure for structure.

A. Structural Similarity of Processes

We determine structural information for a process by examining the functions that the process invokes during execution. Then, we group processes based on their structure. To extract the structural information and perform our grouping, we use the following definitions. Let $\mathcal{P}$ denote the set of all processes and let individual processes be labeled $P, P_1, P_2, P_3, \ldots$. Additionally, for illustrative purposes, we will use names like `process 1` and `thread 3` to refer to individual processes in $\mathcal{P}$. Let $\mathcal{F}$ denote the set of the functions that any process invokes. We label individual functions as $F, F_1, F_2, F_3, \ldots$ or for illustrative purposes with names like `main` and `fopen`. We refer to the power set of a set A as 2^A, which is the set of all subsets of A.

The mapping $\mathit{funcs} : \mathcal{P} \to 2^{\mathcal{F}}$ assigns each process the set of functions that it invokes during an execution. Formally we have: $\mathit{funcs}(P) := \{F \in \mathcal{F} \mid F \text{ is called at least once by } P\}$. We consider processes that invoke many common functions as similar. We express the structural similarity of two processes as the number of functions that they have in common, divided by the number of functions that either process calls:

$$\mathit{funcsim}(P_1, P_2) := \frac{|\,\mathit{funcs}(P_1) \cap \mathit{funcs}(P_2)\,|}{|\,\mathit{funcs}(P_1) \cup \mathit{funcs}(P_2)\,|}$$

Consequently, $funcsim$ maps a pair of processes to a real number between 0 and 1. We use 1 to express total similarity and 0 to express complete dissimilarity. We call *funcsim* a *similarity measure*. As an example, Figure 2 illustrates call trees of two processes. In the example, the sets of invoked functions are identical, i.e.:

$$\begin{aligned}\mathit{funcs}(\texttt{proc 1}) &= \mathit{funcs}(\texttt{proc 2})\\ &= \{\texttt{main}, \texttt{init}, \texttt{fopen}, \texttt{fclose}\}\end{aligned}$$

Consequently, similarity according to *funcsim* is 1.

B. Structural Similarity Measure

Given the high degree of difference between the two call trees in Figure 2, a similarity measure that indicates exact similarity between them is undesirable. Improvement options

for the similarity measure include consideration of supplied arguments, global state, and possibly even external influences, such as incoming messages. Incorporating these factors is hard due to the sheer size of this influential state space. Therefore, we use pairs of functions that represent the *caller-callee* relation in order to refine our measure. We extend the function *funcs* with the function *pairs* : $\mathcal{P} \rightarrow 2^{\mathcal{F}\times\mathcal{F}}$. Formally we define it as $\textit{pairs}(P) := \{(F_1, F_2) \in \mathcal{F} \times \mathcal{F} \mid F_1 \text{ calls } F_2 \text{ on } P\}$. The key advantage of considering the caller-callee relationship is that we can retain structural information in the form of functions, their context in the form of the relation, and that we still rely on a set-based representation that allows us to directly retrieve a similarity measure *pairsim*:

$$\textit{pairsim}(P_1, P_2) := \frac{|\ \textit{pairs}(P_1) \cap \textit{pairs}(P_2)\ |}{|\ \textit{pairs}(P_1) \cup \textit{pairs}(P_2)\ |} \qquad (1)$$

Continuing the example in Figure 2, the function pair sets *pairs* of the two processes exactly contain the directed edges of their corresponding call trees, i.e.:

$$\begin{aligned}\textit{pairs}(\texttt{proc 1}) = \{&\epsilon \rightarrow \texttt{main}, \texttt{main} \rightarrow \texttt{init},\\ &\texttt{main} \rightarrow \texttt{fopen}, \texttt{main} \rightarrow \texttt{fclose},\\ &\texttt{init} \rightarrow \texttt{fopen}, \texttt{init} \rightarrow \texttt{fclose}\}\\ \textit{pairs}(\texttt{proc 2}) = \{&\epsilon \rightarrow \texttt{main}, \texttt{main} \rightarrow \texttt{init},\\ &\texttt{init} \rightarrow \texttt{fopen}, \texttt{init} \rightarrow \texttt{fclose}\}\end{aligned}$$

For readability, we use $F_1 \rightarrow F_2$ to denote a caller-callee pair (F_1, F_2). Functions that are a source in the call stack, such as `main`, have no caller. We use a virtual root function called ϵ to include these source functions in *pairs*, i.e., $\epsilon \rightarrow \texttt{main} \in \textit{pairs}(\texttt{proc 1})$. Based on the results of *pairs* we can directly compute the similarity between processes 1 and 2, which is $\textit{pairsim}(\texttt{proc 1}, \texttt{proc 2}) = \frac{4}{6}$.

Processes that have a similarity of 1 form groups of *similar processes*. We expect that: first, a single execution will exhibit a reasonably low number of such groups; and second, that the number of these groups remains about constant when application scale increases. This expectation results from the fact that a wide range of HPC applications rely on a single executable that is executed by multiple threads, processes, or GPGPU devices. Consequently, the variation in the structural behavior that we observe within a single execution or between multiple executions is limited by the statically compiled executable. Existing approaches [1], [2] for parallel call stacks support this expectation. Section IX analyzes a wide range of applications to evaluate the validity of this assumption.

VI. Computation and Storage

A key requirement for scalability of our approach is an efficient scheme to compute and store similarity information. Storing function pairs for all processes in a table requires memory space that increases with scale. Thus, we needed to find a solution to store input for similarity computations with a small memory footprint.

A. Concept Lattice

Given that we follow a set-based approach to compute similarity, we use *concept lattices* from *formal concept analysis* [9] in order to store groups of similar processes along with their function pairs. A concept lattice is based on a *formal context* [9], which is a triple (O, A, I), where O is a set of objects, A a set of attributes, and $I \subseteq O \times A$ an incidence relation. The incidence relation associates each object with a set of attributes. For process classification, we use the set of processes as the set of objects and the set of all function pairs as the attributes. A formal context defines a *concept lattice* by specifying *concepts*, and a partial order on them. A concept lattice can be represented as a directed acyclic graph where concepts are nodes and the order on them determines the edges. We illustrate this formalism using an example. Let $(\mathcal{P}, \mathcal{A} \subseteq \mathcal{F} \times \mathcal{F}, \mathcal{I})$ be a formal context with:

$$\begin{aligned}&\mathcal{P} := \{P_1, P_2, P_3, P_4\},\\ &\mathcal{F} := \{F_1, F_2, F_3\},\\ &\mathcal{A} := \{\epsilon \rightarrow F_1, F_1 \rightarrow F_2, F_1 \rightarrow F_3\},\\ &\mathcal{I} := \big\{(P_1, \epsilon \rightarrow F_1),\ (P_2, F_1 \rightarrow F_2),\ (P_2, \epsilon \rightarrow F_1),\\ &\qquad (P_2, F_1 \rightarrow F_3),\ (P_3, \epsilon \rightarrow F_1),\ (P_3, F_1 \rightarrow F_2),\\ &\qquad (P_3, F_1 \rightarrow F_3),\ (P_4, \epsilon \rightarrow F_1),\ (P_4, F_1 \rightarrow F_3)\ \big\}.\end{aligned}$$

TABLE I
An incidence relation example.

	$\epsilon \rightarrow F_1$	$F_1 \rightarrow F_2$	$F_1 \rightarrow F_3$
P_1	×	×	
P_2	×		×
P_3	×	×	×
P_4	×		×

The incidence relation can also be described as a table (Table I). In the example, every process has the function pair $\epsilon \rightarrow F_1$, i.e., processes share F_1 as their main function. Processes P_1 and P_3 have the additional pair $F_1 \rightarrow F_2$, i.e., for these processes F_1 calls F_2. Finally, P_2, P_3 and P_4 also have the pair $F_1 \rightarrow F_3$, i.e., for these processes F_1 calls F_3. Figure 3 illustrates the resulting concept lattice. It reads as:

- Process sets subsume those that are reachable following edges downwards;
- Function sets subsume those that are reachable following edges upwards;
- The top node indicates that all processes share the function pair $\epsilon \rightarrow F_1$;
- The bottom node signifies that only P_3 has all function pairs, and in particular that it includes all function pairs that P_1, P_2 and P_4 exhibit;
- P_1 is different from P_2 and P_4; and
- P_2 and P_4 have the same function pairs.

This graph contains redundant information in its labels, e.g., processes occur multiple times. Without loss of expressiveness, we can remove this redundancy to derive the more compact graph in Figure 4.

Once we remove redundant labels, the resulting lattice has the property that each object (process) and attribute (function pair) is contained in the concept lattice exactly once. This is a general property of every concept lattice and not specific to our example. Consequently, the nodes of the lattice provide the process grouping that we desire, since the above property guarantees that each process belongs to exactly one group.

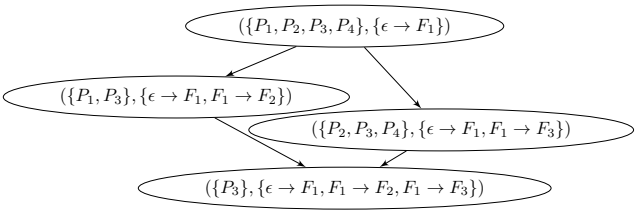


Fig. 3. Concept lattice for the incidence relation of Table I (including redundant information).

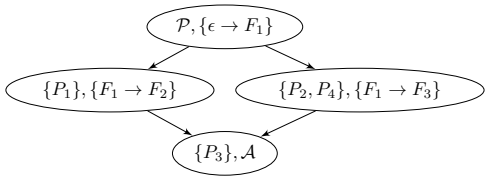


Fig. 4. Concept lattice for the incidence relation of Table I (without redundant information).

Existing algorithms can extract concepts and their partial order from a formal context. One algorithm to create a concept lattice is *Ganter's Next Closure* algorithm [9]. It requires the complete incidence relation to be present in main memory. As previously stated, for large-scale executions, this is not desirable. Thus, we use an iterative algorithm by van der Merwe, Obiedkov, and Kourie [22]—from here on called *van der Merwe's algorithm*—to generate concept lattices.

To clarify our approach, Figure 5 shows the concept lattice for an execution of the real-world application BT with 16 MPI processes, which use 15 OpenMP threads each. Every node contains a set of processes and a set of function pairs. Since the names of individual processes or their function pairs are most accessible to developers of the target application, we only specify how many processes or function pairs belong to a node, e.g., `procs (15)` for a set of 15 processes. Even with 256 overall processes—MPI processes and OpenMP threads—-the concept lattice for this example remains small. All processes share 56 common function pairs. OpenMP threads (right) exhibit different structural behavior than MPI processes (left). All MPI processes have the same function pairs, except for process 0, which invokes one additional pair.

Figure 6 depicts the concept lattice for the application PIConGPU, which uses 17 MPI processes, 20 threads spawned with the Pthreads API, and 16 CUDA streams. The resulting lattice, Figure 6, has 20 nodes and 33 edges. The figure highlights which nodes contain processes of the different parallel programming paradigms—MPI, Pthreads, and CUDA—that PIConGPU uses. Nodes with MPI processes are red (left), nodes with threads are blue (middle), and nodes with CUDA streams are turquoise (right).

B. Similarity Matrix

A concept lattice provides a process grouping, but does not provide similarity values for the distinct groups of processes. Thus, we use the process groups identified in the concept lattice and compute a *similarity matrix* from it. This matrix compares

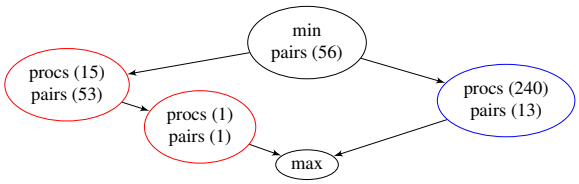


Fig. 5. Concept lattice for an execution of the BT kernel with 16 MPI processes and 15 OpenMP threads per process.

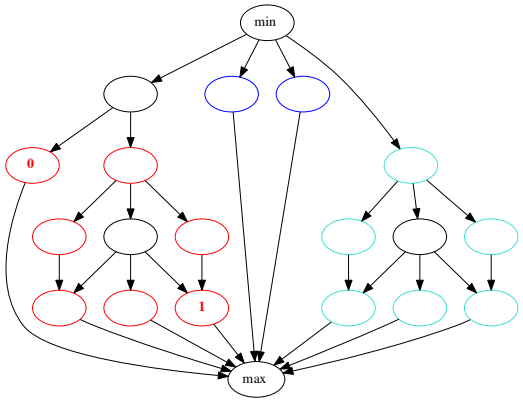


Fig. 6. Concept lattice for an execution of PIConGPU using 17 MPI processes, 20 threads, and 16 CUDA streams.

equivalence classes, since processes on the same lattice node yield the same values when compared to other processes. We obtain function pair similarity (Equation 1) for a node in the lattice as follows: $\mathit{pairs}(P_1) \cap \mathit{pairs}(P_2)$ is the union of function pair sets of nodes reachable from both P_1's and P_2's node along upward-edges. $\mathit{pairs}(P_1) \cup \mathit{pairs}(P_2)$ is the union of function sets of nodes reachable from either P_1's or P_2's node along upward-edges. During computation, we do not need to compute these union sets, since each function pair is contained in the chosen lattice representation exactly once. Thus, we can instead add up the sizes of the involved sets, which is a key strength of using concept lattices for storage and grouping.

We use the application PIConGPU to illustrate an example similarity matrix. The overall concept lattice in Figure 6 includes 20 nodes, of which 17 have associated processes. Consequently, the full similarity matrix is 17×17 entries large. For readability, we restrict ourselves to just the groups that include MPI processes. Figure 7 depicts this subgraph of the overall concept lattice, which only includes nodes important to computing the similarity of MPI processes. The figure specifically highlights MPI processes 0 and 1 in red, we will refer to them in the following.

The subgraph of the lattice has the following properties:

- There are seven groups in total that correspond to the seven nodes labeled with at least one process;
- MPI process 0 has 11 function pairs in common with the others (top node) and introduces 66 new function pairs;
- MPI processes 1–16 all share 171 ($11 + 160$) pairs; and
- The differences between these processes are very small, since only few new function pairs are introduced at nodes descending from here.

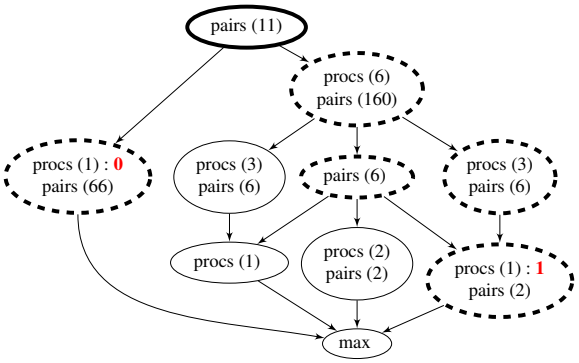


Fig. 7. Detailed subgraph of the concept lattice from Figure 6.

TABLE II
SIMILARITY MATRIX FOR THE PROCESS GROUPS OF THE PICONGPU CONCEPT LATTICE SUBGRAPH IN FIGURE 7.

	0	1	2, 3	4	5, 9, 13	6, 7, 10, 11, 14, 15	8, 12, 16
0							
1	0.04						
2, 3	0.04	0.95					
4	0.04	0.93	0.96				
5, 9, 13	0.05	0.96	0.92	0.90			
6, 7, 10, 11, 14, 15	0.05	0.92	0.96	0.93	0.97		
8, 12, 16	0.05	0.90	0.92	0.97	0.93	0.97	

We use MPI processes 0 and 1 to illustrate a similarity computation. To calculate $|\ \mathit{pairs}(0) \cap \mathit{pairs}(1)\ |$, Figure 7 highlights the single common ancestor node for the nodes of MPI processes 0 and 1 with a thick solid line. Therefore, this expression evaluates to the 11 function pairs that these processes have in common. To compute $|\ \mathit{pairs}(0) \cup \mathit{pairs}(1)\ |$, Figure 7 highlights the ancestor nodes of MPI processes 0 and 1 with thick lines (dashed or solid). Thus, this expression evaluates to the sum of all function pairs in the highlighted nodes. Similarity between MPI processes 0 and 1 is about 4.4%, i.e.: $\mathit{pairsim}(0,1) = \frac{11}{11+66+160+6+6+2}$.

Table II depicts the similarity matrix for all 7 groups of MPI processes. Since it is symmetric and every diagonal entry is 1, we only include the bottom left triangle. The table illustrates that process 0 is highly dissimilar from all other processes, since it serves in a master role. All other groups of processes exhibit very similar structure and have a similarity of 90% or more.

C. Scalability Considerations

Our process classification must first construct a lattice. Afterwards, we use the lattice to extract a similarity matrix. To construct the concept lattice, van der Merwe's algorithm iteratively inserts function pair sets for each process. Based on the completed concept lattice, for each pair of nodes, with both nodes having at least one process, we perform an upwards traversal in the graph. The traversal calculates the numerator and denominator according to Equation 1. We then compute the desired fraction for the pair of nodes and, thus, retrieve the similarity between the two groups that the nodes represent.

The worst-case space complexity of storing a concept lattice, as shown by Godin et al. [10], is $\mathcal{O}(2^a p)$ where a is the total number of attributes and p the process count. Therefore, and according to van der Merwe et al. [22], the worst-case time complexity of generating the lattice and computing the similarity matrix is $\mathcal{O}(2^a p^3 a)$.

In practice, as mentioned in the previous section, processes are expected to fall into few groups. Particularly we expect that this count is constant across scale. The attribute count is constant as well, since the number of functions in a program does not increase with rising process counts or increasing problem size. The space complexity of storing the lattice is, therefore, expected to be independent of the process and attribute count, i.e. $\mathcal{O}(1)$. Because similarity matrix computation only depends on the number of nodes in the lattice, the most expensive part of our approach is inserting each process's function pairs into the lattice. Therefore, the expected time complexity of the lattice creation and similarity extraction is $\mathcal{O}(p)$ with p being the process count.

We present performance measurements with our scheme in Section IX. These measurements demonstrate that even at increasing scale $\mathcal{O}(p)$ remains acceptable. However, to handle cases with millions of processes, a parallel scheme can become necessary. Algorithms to generate concept lattices in parallel exist [12], [17]. They suggest feasibility of an algorithm that creates a concept lattice in $\mathcal{O}(\log p)$ steps. A straightforward solution would be to first create multiple lattices in parallel. These intermediate lattices can then be merged into a final lattice in a tree-esque fashion.

VII. SUBSUMPTION MEASURE

We presented function pair similarity, *pairsim*, as a measure of the similarity between processes. To improve this measure, consider that one process is spawned from another process. Even if the structural behavior of the spawned process is completely included in the parent process, the *pairsim* measure can indicate low similarity between them. Reasons for this include differences in the initial portion of call stacks or that the parent process exhibits additional behavior. Thus, we define the *subsumption measure* that identifies cases where structural behavior of one process is included in another process. This measure is useful for cases such as:

- A compiler replaces a function call with its implementation (*inlining*);
- A process calls additional functions, for example when printing debug output or writing to disk;
- For a tracing tool, processes were recorded with different levels of detail due to filtering; and
- Threads are spawned from a process using the fork-join model.

The subsumption measure does not replace function pair similarity, but rather adds a second type of comparison. If two process groups have low function pair similarity, the subsumption measure can differentiate process groups that have distinct behavior from cases where one group has a subset of the other groups behavior. Figure 8 shows an example to illustrate this subsumption. `proc 1` calls function F_1, which then calls F_2,

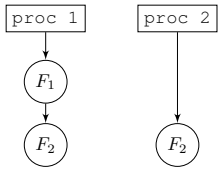


Fig. 8. Example call trees, where function $F1$ has been inlined in `proc 2`.

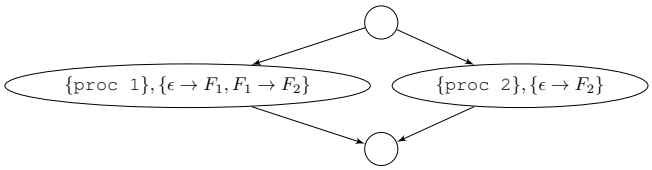


Fig. 9. Concept lattice for the function pairs of the call trees in Figure 8.

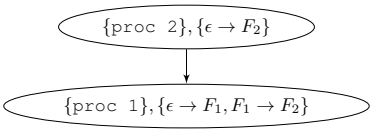


Fig. 10. Concept lattice of the transitively closed function pair sets for the example call trees.

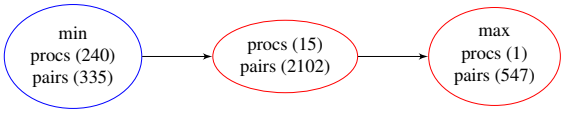


Fig. 11. Concept lattice (drawn left to right) of the transitively closed function pair sets for a BT kernel run utilizing 16 processes and 15 threads each.

while for `proc 2`, F_1 is inlined. Figure 9 presents the concept lattice for this example, showing no similarity between the two processes.

The subsumption measure enhances the function pair similarity measure to include more behavior information. To achieve this, we apply the transitive closure $(\ )^+$ to the function pair set of each process to factor out differences in lower call stack levels. The transitive closure A^+ on a set of function pairs $A \subseteq \mathcal{F} \times \mathcal{F}$ ensures that if F' is called by F via zero or more intermediate functions, A^+ includes $F \rightarrow F'$. In the example, the transitive closure for the function pairs of `proc 1` is $\{\epsilon \rightarrow F_1, \epsilon \rightarrow F_2, F_1 \rightarrow F_2\}$. This procedure has no significant impact on overheads, since the number of function pairs is constant, the transitive closure can be computed for all processes independently, and the size of the resulting lattice does not increase. Informally, differences between function pair sets cause lattice nodes to be split. Taking the transitive closure of two function pair sets can only lessen differences between them, but never creates new differences.

After computing the transitive closure, we construct the lattice as before. For the example, Figure 10 presents the concept lattice that results with transitively closed function pair sets. As opposed to the overall behavioral classification of the lattice in Figure 9, this lattice shows that both processes share behavior in calling function F_2, irrespective of the intermediate function F_1.

Based on this lattice of transitively closed function pair sets, a third step computes the subsumption measure for each pair of process groups. To sharpen subsumption towards considering shared behavior, we define the measure *pairsub* as:

$$pairsub(P_1, P_2) := \frac{|\ pairs(P_1)^+ \cap pairs(P_2)^+\ |}{|\ pairs(P_2)^+\ |}$$

Again, if P_2's function pair set is empty, we define the result to be 1. For the example, the function pair subsumption measure is 1. This means, `proc 1` includes all structural behavior of `proc 2`. Asking the reverse yields $\frac{1}{3}$, since `proc 2` invokes fewer functions than `proc 1`. Use of $pairs(P_2)^+$ as divisor instead of $pairs(P_1)^+ \cup pairs(P_2)^+$ enables computing inclusion rather than similarity.

To illustrate the subsumption measure we demonstrate it for the application runs of BT and PIConGPU. For BT, Figure 11, the subsumption measure highlights that MPI processes 1 to 16 subsume the behavior of the threads. One can, therefore, safely assume that the total 240 threads perform a subset of the work of the MPI processes. Consequently, with function pair similarity we understand that processes differ from threads and with subsumption-based similarity we then enrich this knowledge with relating threads as a functional subset of the processes. The relation between process 1 and 2–16 does not change, process 1 subsumes processes 2–16 with both measures. In contrast for the PIConGPU run, the subsumption measure highlights that this application actually uses a functional decomposition, as this application shows no subsumption. Again, this result enriches the information that we gain with function pair similarity.

VIII. Benefits of Pre-Clustering

The key benefit of our lightweight structural pre-clustering method is its ability to significantly improve the result quality of existing performance data comparison approaches. We describe the possible improvements from using our method with four use cases from existing comparison approaches:

1) Additional information for manual analysis,
2) Pre-grouping for time-based clustering approaches,
3) Pre-grouping and differentiation for profile-based comparison of different program runs, and
4) Pre-selection of processes for alignment-based comparison.

A. Manual Analysis

To support manual analysis, we provide the user with the similarity matrix directly. The matrix shows groups of equal processes and the similarity of the groups according to our similarity measure. This aids the user in choosing processes on which to focus during manual inspection. Performance analysis frameworks such as Paraver [11], Intel Trace Analyzer [13], and Vampir [4] allow users to specify a set of processes to inspect. These analysis tools can then work more quickly, display more meaningful information, or for example, provide individual statistics and visual comparison of different process types inside one application run.

B. Time-based Clustering

The example in Figure 1 introduced time-based clustering [4], to compare all processes of an execution. This approach groups processes that exhibit similar timing for the functions that they invoke. Wide ranges of clustering algorithms support this grouping. However, as we found in our example, by only considering time, this approach can group together processes that share close to no behavior similarities.

Our approach provides structural information so that processes that exhibit starkly different behavior will not be compared meaninglessly. We achieve this by pre-clustering processes before applying time-based analysis. Therefore, we first calculate the similarity matrix and then combine pairs of groups using a greedy scheme. The scheme iteratively combines groups that exhibit a high degree of similarity, using a fixed threshold σ, according to the *pairsim* similarity measure. We define the similarity between a process and a group to be the average similarity weighted by the number of contained processes.

Table II shows the similarity matrix for PIConGPU excluding CUDA streams and threads. While the matrix directly collapses processes with absolute function pair similarity into one group, it still contains multiple distinct groups that have a high degree of similarity. Applying our greedy scheme with a threshold of $\sigma = 0.83$ yields five groups of processes: One group for MPI process 0, one group for MPI processes 1–16, two groups for two different thread types, and one for all CUDA streams. By applying this pre-clustering we improve the quality of the time-based analysis in Figure 1(b). This clustering now aids analysts since it highlights timing variations within each group—with the shade overlays on the runtime bars.

C. Differential Profiling

Approaches comparing multiple executions [14], [19]–[21] commonly create one profile for an execution by accumulating the timing information of all processes. Comparing such profiles then reveals differences between executions. However, if an execution has multiple groups of distinctly behaving processes, the creation of one overall profile can easily hide interesting features. Assume that an application spawns an extra thread on each tenth MPI process, e.g., for checkpointing. Given the much lower number of these threads in comparison to the processes, say 10 to 100, the profile will largely hide the behavior of the additional threads, since it will average them out. At the same time, given a slow I/O subsystem, the behavior of the threads can easily have a strong impact on the overall performance.

Towards this end, we propose to apply the same pre-clustering as we use for time-based clustering and to consider behavior groups when creating a differential profile. Instead of creating one overall profile, we can then create one profile for each behavior group. A performance comparison of two executions then compares matching pairs of behavior groups.

D. Alignment-Based Comparison

The fourth use case for our structural pre-clustering is to improve alignment-based performance comparison [24]. Alignment approaches compare traces to identify where in an execution two processes exhibit differing behavior (in terms of structure) and when they exhibit different execution times. As opposed to profile-based approaches they show a comparison over the runtime of the application. In general, alignment of performance traces provides a drastically increased level of detail. However, current approaches require $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$ time [23] to align two traces with n events. In many trace data sets, n is in the order of millions or larger. Exhaustively comparing all processes of an execution with p processes requires $\mathcal{O}(p^2)$ alignments, which is prohibitive.

The pre-clustering of processes with the aforementioned greedy scheme and a threshold of σ overcomes this limitation. The distinct behavior groups of our pre-clustering share little similarity. Consequently, an alignment between processes of different groups will commonly not be meaningful. Instead, we propose to compare two pairs of processes from within each behavior group. The first pair serves to investigate structural difference within the group and should include two elements of maximum dissimilarity. If the greedy scheme combines groups $G_1, G_2, \ldots, G_n$ to a new group G', then the most dissimilar processes $g_i \in G_i, g_j \in G_j$ is the pair with G_i and G_j having the lowest similarity score. The pair shows when during the execution the behavior within a group differs and to which degree it differs. The second pair of processes that we align gives two processes with a high degree of difference in their timing behavior. We propose to calculate a time-based profile for all processes within a group. We then select two processes with a high or maximum difference in their profiles.

In summary, using our pre-clustering, instead of aligning all possible combinations of process pairs, we propose to compare $2 \cdot g$ pairs of processes where g is the number of groups. As we will demonstrate in the next section, g remains stable with rising scale and increasing problem size for a wide range of applications. Therefore, our approach improves the applicability of alignment-based performance analysis drastically.

IX. Results

We demonstrate the applicability and scalability of our structural pre-clustering approach in a study with 15 HPC applications. For this, we extract data from application traces and use the data to construct concept lattices, as well as to compute similarity matrices. Note that while we use traces for this analysis, traces are not required for our structural pre-clustering method; performance profiles could be used instead.

A. Applicability

We first demonstrate that our technique applies to a variety of applications with different characteristics. The test applications are a selection of benchmarks, pseudo-applications, and real-world applications[2]. Table III summarizes the results. For each application, we construct a concept lattice with our function-pair-based approach. Then we compute the similarity between all process groups that the lattice identifies. We conducted all measurements on an Intel Core i5-2520M system with dual

[2] *HPL* "High-Performance Linpack", *GROMACS* "Groningen Machine for Chemical Simulations", *CCLM* "COSMO Climate Limited-area Modelling", *COSMO-SPECS* COSMO with added cloud microphysics, *WRF* "The Weather Research & Forecast Model", *FD4* dynamic load balancing for COSMO-SPECS and WRF, *HOMME* "High-Order Methods Modeling Environment", *AMG* "Algebraic Multigrid Solver", *IRS* "Implicit Radiation Solver", *LULESH* "Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics", *ParaDiS* "Parallel Dislocation Simulator", *PIConGPU* "Particle-In-Cell on Graphics Processing Units", *BT-MZ* part of NAS Parallel Benchmarks, *HPCC MPI-FFTE* part of HPCC benchmark suite, *PEPC* "Pretty Efficient Parallel Coulomb Solver"

TABLE III
APPLICATION AND SCALE IMPACT ON PRE-GROUPING RUNTIME, AS WELL AS LATTICE SIZE.

Application			Result		
Name	Processes	Func. pairs	t_{eval} (ms)	Lattice Nodes	Process Groups
HPL	2,360	8	< 10	2	2
GROMACS	36	1,381	< 10	24	11
CCLM	180	180	< 10	6	3
COSMO-SPECS	100	50	< 10	1	1
WRF	64	774	< 10	2	2
FD4	65,536	55	55	22	14
HOMME	1,024	179	< 10	3	3
AMG2006	1,024	440	66	11	7
IRS	64	989	34	18	7
LULESH	432	406	49	182	35
ParaDiS	128	649	3,486	6,367	74
PIConGPU	39	474	11	60	17
BT-MZ	16	126	< 10	5	3
HPCC MPI-FFTE	128	109	70	7	4
PEPC	16,384	113	15	2	2

channel DDR3 PC3-10600 RAM, using the GNU C Compiler and `-O2` as optimization flag.

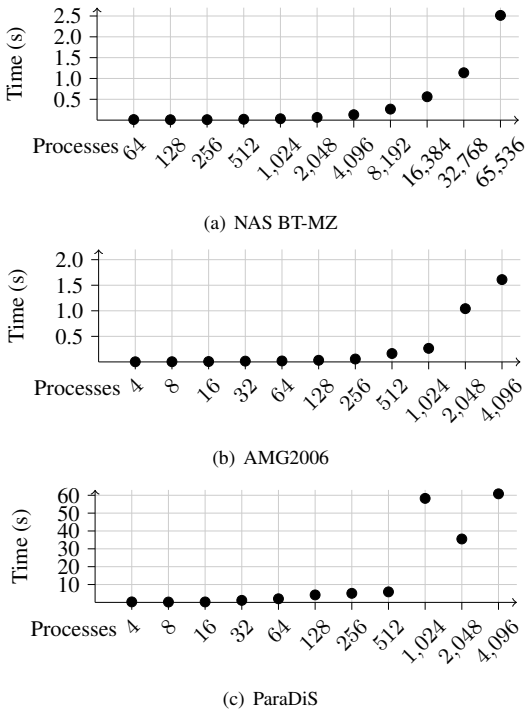
For each application, Table III characterizes the total number of processes and its function pair count on the left. The column *Processes* states the total number of processes, which may include MPI processes, threads, and CUDA streams. On the right side of the table we depict the results of applying our methods. The evaluation time t_{eval} includes the steps of adding each process and its function pairs to the lattice, as well as computing the similarity matrix from the final lattice. For each application, except ParaDiS, the evaluation time is less than 0.1 seconds. This underlines the wide applicability of our approach. For 10 out of the 15 applications, the number of process groups is below ten, even at a scale of 16,384 processes. This result supports our expectation that the number of process groups is often independent of scale. The applications ParaDiS and LULESH exhibit increased numbers of process groups, 74 and 35, respectively. However, lattice construction and similarity computation remain within an acceptable computational overhead for these applications. In the next section, we investigate ParaDiS in detail to highlight the source for the increased number of process groups. Overall, we see the results of our applicability analysis as a strong indicator that our assumptions on process group counts hold for wide ranges of applications.

B. Scaling Study

To analyze the impact of application scale in more detail, we conduct additional measurements with two benchmarks and one application: NAS BT-MZ, AMG2006, and ParaDiS. To conduct measurements on a second hardware platform, we used an Intel Xeon E5-2690 system with 32–128 GB RAM per board. Hyper-Threading has been disabled system-wide. For compilation we used the GNU C Compiler and `-O2` as optimization flag. We execute our process classification approach on one processor core with a fixed clock rate of 2.9 GHz and disabled turbo boost.

Figure 12 depicts the timing for our structural pre-clustering across application scale. Note that process numbers in Figure 12 are plotted on a logarithmic scale.

a) NAS BT-MZ: The first study uses the pseudo application BT-MZ from the NAS Parallel Benchmarks [3]. Figure 12(a) shows the wallclock computation time for our approach at up to



(a) NAS BT-MZ

(b) AMG2006

(c) ParaDiS

Fig. 12. Overheads of our process classification approach across scale.

65,536 processes (4,096 MPI processes plus 61,440 threads). Since BT-MZ exhibits a very regular function call structure across all processes, the number of groups and nodes in the lattice remains constant. The only additional work required for increasing scale is to add more processes and their function pairs into the lattice. Calculation time rises linearly with increasing process count, as we expected for the $\mathcal{O}(p)$ complexity that we derived in Section VI. While the calculation time of 2.5 seconds remains acceptable, increases in scale may motivate the need for a parallel implementation as we discussed before.

b) AMG2006: Figure 12(b) depicts computation times for AMG2006 [8] with up to 4,096 MPI processes. Again, following our time complexity analysis, we observe an almost linear increase in runtime, with a maximum of 1.6 seconds. Different process counts can cause different domain decompositions and varying iterative refinements. In our measurements, AMG2006 calls between 207 and 217 distinct functions and has 436–483 different function pairs. The variation stems from different code paths being taken. Due to these variations AMG2006 exhibits a less regular call structure than BT-MZ. The number of lattice nodes and groups varies as a consequence.

c) ParaDiS: The third study investigates how our approach behaves for the Parallel Dislocation Simulator [5] (ParaDiS). Each ParaDiS process computes on its own domain with an adaptive load balancer that periodically redistributes load between domains.

Figure 12(c) depicts classification time for ParaDiS with up to 4,096 processes. In ParaDiS the function call structure of a process heavily depends on its individual workload. Particularly, for runs with 1,024 or more processes we observe increased lattice node counts as a result. The largest lattice in this study contains 57,814 nodes and 216 groups. The group counts are

acceptable, whereas the number of nodes prevents efficient computation. Computing the similarity matrix of a ParaDiS run using 4,096 process takes more than 60 seconds. Since our approach targets pre-clustering of similar processes as an input for subsequent detailed performance analysis, we consider this run time to be high.

To prevent this high run time and coarsen the resulting groupings, we propose to apply the following approach. We monitor the number of nodes during lattice construction to detect when a lattice grows too large. Then, we use functions—our first similarity measure—instead of function pairs to determine the degree of similarity. This leads to far fewer lattice nodes and a coarsening of the grouping. In our tests, this approach decreases node counts to about 15% of their original size. Additionally, it reduces the computation time for the similarity matrix to 17% of the original time. While we have only begun to explore this approach for reducing computation size and improving groupings for problematic applications, we believe it is promising and plan to explore it and other options further in future work.

X. Conclusions

Performance analysis of HPC applications is a notoriously difficult task and typically involves inspection and comparison of potentially large amounts of performance data. To alleviate this problem, current performance analysis tools perform data reductions. However, the techniques they use to reduce data suffer from lack of scalability, or do not consider program execution structure. Thus, analysts struggle between using inefficient methods for processing performance data, or using methods that can inadvertently omit important performance details.

Our solution to this problem is structural pre-clustering, a fast, scalable, and lightweight approach to group processes of a parallel execution according to their function call structure. While our structural pre-clustering approach provides a basis for comparing processes manually, its key strength lies in its ability to improve the effectiveness of existing performance analysis techniques. Our pre-clustering step enhances existing techniques because it reduces data into intuitive and logical clusterings before applying the more traditional analysis steps.

In this paper, we demonstrated how our structural pre-clustering improves the quality of existing analysis techniques, including time-based clustering, differential profiling, and alignment-based trace comparison. We applied structural pre-clustering to 15 HPC applications on up to 65,536 parallel processes, as well as to two particularly challenging applications that use MPI processes, OpenMP threads, and CUDA streams. In almost all cases, the number of clusters generated by our approach remains low and stable for increasing process counts. We found only one application yielding cluster counts that increase with scale, and for this scenario we developed a strategy to avoid this performance degradation. Overall, our approach yielded small numbers of clusters, less than 10 in most cases for process counts of up to 16,384, and only 22 clusters for our 65,536 process run of FD4.

Acknowledgments

This work has been supported by the NEXTGenIO project that has received funding from the European Union's Horizon2020 Research and Innovation programme under Grant Agreement no. 671951. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-CONF-669728

References

[1] Allinea DDT and MAP User Guide — Viewing Stacks in Parallel. https://www.allinea.com/user-guide/forge/userguide.html, Oct. 2015.

[2] D. C. Arnold, D. H. Ahn, B. R. De Supinski, G. L. Lee, B. P. Miller, and M. Schulz. Stack Trace Analysis for Large Scale Debugging. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[4] H. Brunst and M. Weber. Custom Hot Spot Analysis of HPC Software with the Vampir Performance Tool Suite. In *Proceedings of the 6th International Parallel Tools Workshop*, pages 95–114. September 2012.

[5] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis. Scalable Line Dynamics in ParaDiS. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 19. IEEE Computer Society, 2004.

[6] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, et al. Radiative Signatures of The Relativistic Kelvin-Helmholtz Instability. In *Proc. of SC13: Intl. Conference for HPC, Networking, Storage and Analysis*, 2013.

[7] Parallel Programming and Computing Platform — CUDA — NVIDIA. http://www.nvidia.com/object/cuda_home_new.html, Oct. 2015.

[8] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp. Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 172–181, 2011.

[9] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. 1999.

[10] R. Godin, R. Missaoui, and A. April. Experimental comparison of navigation in a Galois lattice with conventional information retrieval methods. *Intl. Journal of Man-Machine Studies*, pages 747–767, 1993.

[11] J. Gonzalez, J. Gimenez, and J. Labarta. Automatic Detection of Parallel Applications Computation Phases. In *Parallel & Distributed Processing. IPDPS 2009. IEEE International Symposium on*, pages 1–11, 2009.

[12] X. Hu, X. Wei, D. Wang, and P. Li. A Parallel Algorithm to Construct Concept Lattice. In *Fuzzy Systems and Knowledge Discovery, 2007. FSKD 2007. Fourth Intl. Conference on*, pages 119–123, 2007.

[13] Intel Trace Analyzer and Collector. http://software.intel.com/en-us/articles/intel-trace-analyzer/, Oct. 2015.

[14] K. L. Karavanic and B. P. Miller. Experiment Management Support for Performance Tuning. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '97, pages 1–10, 1997.

[15] A. Knüpfer, B. Voigt, W. E. Nagel, and H. Mix. Visualization of Repetitive Patterns in Event Traces. In *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*, PARA'06, pages 430–439, 2007.

[16] K. Mohror and K. L. Karavanic. Evaluating Similarity-based Trace Reduction Techniques for Scalable Performance Analysis. In *Proc. of the Conference on HPC Networking, Storage and Analysis*, SC '09, 2009.

[17] P. Njiwoua and E. M. Nguifo. A Parallel Algorithm to Build Concept Lattice. In *Proceedings of the 4th Groningen International Information Technology Conference for Students*, volume 107, 1997.

[18] P. Ratn, F. Mueller, B. R. de Supinski, and M. Schulz. Preserving Time in Large-Scale Communication Traces. In *Proc. of the 22th Annual Intl. Conference on Supercomputing*, ICS '08, pages 46–55, 2008.

[19] M. Schulz and B. R. de Supinski. Practical Differential Profiling. In *Proceedings of the 13th international Euro-Par conference on Parallel Processing*, Euro-Par'07, pages 97–106, 2007.

[20] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford. Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis. *Scientific Programming*, 2008.

[21] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An Algebra for Cross-Experiment Performance Analysis. In *Proceedings of the Intl. Conference on Parallel Processing*, ICPP '04, pages 63–72, 2004.

[22] D. Van Der Merwe, S. Obiedkov, and D. Kourie. AddIntent: A New Incremental Algorithm for Constructing Concept Lattices. In *Concept Lattices*, pages 372–385. 2004.

[23] M. Weber, R. Brendel, and H. Brunst. Trace File Comparison with a Hierarchical Sequence Alignment Algorithm. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA '12, pages 247–254, July 2012.

[24] M. Weber, K. Mohror, M. Schulz, B. R. de Supinski, H. Brunst, and W. E. Nagel. Alignment-Based Metrics for Trace Comparison. In *Proc. of the 19th Int. Conf. on Parallel Processing*, Euro-Par'13, pages 29–40. 2013.