Clustering Performance Data Efficiently at Massive Scales *

Todd Gamblin*, Bronis R. de Supinski*, Martin Schulz*,
Rob Fowler†, and Daniel A. Reed‡

*Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore, CA, 94550, USA
†Renaissance Computing Institute, University of North Carolina, Chapel Hill, NC, 27599, USA
‡Microsoft Research, One Microsoft Way, Redmond, WA, 98052, USA

{tgamblin, bronis, schulzm}@llnl.gov, rjf@renci.org, daniel.reed@microsoft.com

ABSTRACT

Existing supercomputers have hundreds of thousands of processor cores, and future systems may have hundreds of millions. Developers need detailed performance measurements to tune their applications and to exploit these systems fully. However, extreme scales pose unique challenges for performance-tuning tools, which can generate significant volumes of I/O. Compute-to-I/O ratios have increased drastically as systems have grown, and the I/O systems of large machines can handle the peak load from only a small fraction of cores. Tool developers need efficient techniques to analyze and to reduce performance data from large numbers of cores.

We introduce CAPEK, a novel parallel clustering algorithm that enables *in-situ* analysis of performance data at run time. Our algorithm scales sub-linearly to 131,072 processes, running in less than one second even at that scale, which is fast enough for on-line use in production runs. The CAPEK implementation is fully generic and can be used for many types of analysis. We demonstrate its application to statistical trace sampling. Specifically, we use our algorithm to compute efficiently stratified sampling strategies for traces at run time. We show that such stratification can result in data-volume reduction of up to four orders of magnitude on current large-scale systems, with potential for greater reductions for future extreme-scale systems.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement Techniques; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Clustering*

General Terms

Algorithms, Measurement, Performance

*This work was performed under the auspices of the U.S. Department of Energy, supported by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-422684).

To appear in *ICS'10*. Draft typeset on April 20, 2010.

ICS'10, June 2–4, 2010, Tsukuba, Ibaraki, Japan.

1. INTRODUCTION

Scalability is a key challenge in designing parallel performance tools. Petascale supercomputers have over 100,000 cores, and exascale systems are expected to have hundreds of millions of cores [1, 11, 21]. Thread counts on these systems may reach into the billions.

Programmers need detailed performance information to analyze and to tune application performance, but collecting this information can be a difficult task. For example, load balance is critical for performance at scale. Many scientific codes use adaptive techniques in which per-process load depends on application data. In large physical simulations, particles or other model components may move between processes. Overall, load distributions may evolve. Some applications employ adaptive load-balancing techniques to mitigate these problems, but understanding how load balancing schemes perform and diagnosing their limitations requires the generation of load traces across processes and over time. The size of a performance trace collected from such an application will scale linearly with the process count and simulation length.

While performance data sizes continue to grow, I/O to compute ratios of modern large-scale machines have shrunken. Further, concurrent output from all processes leads to unacceptable performance, and application developers must frequently develop novel I/O strategies to limit contention for shared file-system resources. These limits hamper tool developers even more, because tools compete for resources against the applications that they measure. Too much output from a tool can severely perturb the application, invalidating any measurements it may make.

An ideal performance tool would collect only measurements pertinent to an application's performance. However, tool developers often do not know *a priori* which data are pertinent. Typical trace tools collect per-timestep, per-process data for off-line analysis. As core counts increase, we must perform some or all of this analysis on-line to avoid saturating the I/O system.

This paper's primary contribution is CAPEK, an algorithm for scalable parallel cluster analysis. We use CAPEK to identify processes with similar performance behavior and to group them into equivalence classes. Knowing performance equivalence classes allows tool developers to focus on a representative subset of processes, which reduces data volume considerably. We show that CAPEK scales to 131,072 processes on an IBM BlueGene/P system, and on this system it exhibits a run time suitable for on-line use in production runs.

Our implementation of CAPEK is generic, using C++ templates suitable for use on arbitrary data types and for arbitrary distance metrics. We demonstrate its utility by clustering performance trace

data. Prior work showed that statistical sampling could reduce the volume of performance-trace data by over an order of magnitude on comparatively small systems for performance clusters that are known *a priori* [10, 27]. Using our algorithm, we are able to use clustering information to *stratify* on-line performance traces adaptively, and we achieve data reduction of four orders of magnitude for much larger systems.

The remainder of this paper is organized as follows. In Section 2, we describe existing clustering techniques and their relevance to our approach. We present our parallel clustering algorithm in Section 3. Section 4 briefly describes the *effort model* [9] for parallel performance data, and Sections 5 and 6 describe methods for scalably clustering and stratifying *effort signatures* collected using this model. In Section 7, we present experimental results from a production job on a capability-class BlueGene/P system to demonstrate that our technique can scalably reduce performance-data volume by four orders of magnitude.

2. CLUSTERING TECHNIQUES

Clustering algorithms find groups, or *clusters*, of similar items within data. Users define measures of distance or *dissimilarity* on the data elements to be clustered, and clustering algorithms find groups of objects that minimize this dissimilarity, either optimally or according to some heuristic. While a full taxonomy of clustering algorithms is beyond the scope of this paper, we describe two common clustering approaches in this section: *hierarchical clustering* and *partitioning algorithms*.

2.1 Hierarchical Clustering

Hierarchical or *agglomerative* clustering [17, 30] starts with a set of objects and combines the closest objects into a single cluster. The resulting clustering does not identify distinct groups, but rather a hierarchy of increasingly larger groups with the full data set at the root and individual objects at the leaves. This resulting tree, or *dendrogram*, can be partitioned into groups of arbitrary granularity by *cutting* the tree at a particular level. Hierarchical clustering algorithms have complexity $O(n^2)$ or $O(n^2 log(n))$. Most of this time is spent pre-computing a *dissimilarity matrix* to store distances between each pair of objects.

2.2 Partitioning Algorithms

In this work, we focus on partitioning algorithms. Instead of dividing objects into hierarchical groups, these algorithms divide a set of objects into non-overlapping clusters.

2.2.1 K-Means

The most common partitioning algorithm is the the K-Means method [6, 12, 22, 23]. K-Means clustering takes n objects and k, the number of clusters to find, as input. It then groups the objects into k clusters with the objective of minimizing the *squared error*, the sum of the squared dissimilarities of each object from the *centroid*, or *mean*, of its cluster. Although K-Means does not ensure optimality, the algorithm is fast and typically converges in many fewer than n iterations [2].

K-Means clustering has several limitations. First, it is sensitive to outliers since it attempts to minimize squared dissimilarity within clusters instead of minimizing dissimilarity directly. Second, to compute the means K-Means requires that object metrics support algebraic operations, a property that does not hold for all types of data. Further, the representative of each cluster in a K-Means clustering is its centroid, which is likely not identical to any member of the cluster.

2.2.2 K-Medoids

Due to these limitations, we focus on another set of partitioning techniques, the K-Medoids methods, which do *not* require algebraic operations on the data to exist. Thus, K-Medoids applies even when we can only compute the dissimilarities between objects. Like K-Means, K-Medoids takes the number of clusters k as input, but differs in three ways. First, K-Medoids methods attempt to minimize dissimilarity directly, which reduces sensitivity to outliers. Second, instead of centroids, K-Medoids uses *medoids*, which are objects from the input data, as cluster representatives. Finally, K-Medoids only requires that we formulate a distance measure between input elements (effort traces in our studies). It does not require us to compute a mean.

PAM [18], which is the most basic K-Medoids implementation, has $O(n^2)$ complexity. As in hierarchical clustering approaches, most of its run time is spent computing a dissimilarity matrix. Since PAM is costly for large n, many use CLARA [19], which is a less expensive, sampled variant of K-Medoids. This approach does not cluster an entire data set directly. Instead, it first applies PAM to subsets of the full data, which reduces n and, thus, the run time. It then assigns each object of the original data set to that cluster from the PAM run that has the nearest medoid. CLARA repeats this process several times and selects the clustering with the lowest sum of the dissimilarity of each object from its cluster's medoid. CLARA has complexity $O(n + s^2 S)$, where s is the size of the sample and S is the number of iterations. With $s \leq \sqrt{n}$, CLARA has complexity and storage requirements that are linear in the size of the data set rather than quadratic.

2.2.3 Clustering Criteria

Unlike hierarchical clustering, partitioning schemes require the user to pick the number of clusters, k, in advance. However, the optimal k is often unknown *a priori* since it depends on the specific structure of the data. A commonly used method for overcoming this drawback for data sets where the optimal k is small by running multiple trials with different values of k. We can then assess the quality of the resulting clusters based on a *clustering criterion* [26, 33], which assesses how well dissimilarity is minimized within clusters. We then use the result with the best clustering criterion. We discuss this approach further in Section 3.3.

2.3 Parallel Implementations

Many have studied parallel hierarchical clustering. However, even algorithms for networks of n processors have $O(nlog(n))$ complexity [31]. This is too high for performance tools, as even linear scalability typically prevents their adoption. Although implementations exist for distributed-memory parallel machines with small processor counts, their speedups (25 times with 48 processors on biological data [5] and 9 times on 32 processors for financial data [38]) do not meet the needs of large-scale applications. To our knowledge, no implementations exist for large processor counts.

K-Means and hierarchical clustering are both used to cluster performance data in the PerfExplorer tool [15], but the implementations are sequential and the data analysis is off-line. The Etrusca framework [35] uses K-Means to cluster parallel performance data on-line, but Etrusca is not scalable, as it aggregates data to a single, central machine and applies the sequential K-Means algorithm.

Parallel K-Means implementations have scaled to slightly larger system sizes than hierarchical clustering implementations. A variant of K-Means for image-analysis applications on supercomputers with a hypercube interconnect achieved 50% parallel efficiency on hypercube networks with 64 cores [34]. Another parallel K-Means implementation achieved 90% parallel efficiency on 32 pro-

```
CLARA(X, d, k)
 1  bestDissim ← ∞
 2  for t ← 1 to S
 3  do X' ← RANDOM-SUBSET(X, s)
 4     D ← BUILD-DISSIM-MATRIX(X', d)
 5     (C', M) ← PAM(X', D, k)
 6     C ← ASSIGN-MEDOIDS(X, M, D)
 7     dissim ← TOTAL-DISSIM(C, M, D)
 8     if dissim < bestDissim
 9       then bestDissim ← dissim
10            C_best ← C
11            M_best ← M
12
13  return (C_best, M_best)
```

Figure 1: Sequential CLARA algorithm

cessors for data sets of up to 100,000 elements. A similar algorithm achieved nearly linear speedup on up to 128 processors and up to 10 million objects [7, 8]. However, we know of no results on thousands, let alone a hundred thousand cores, as we target.

Kaufman, *et al.* developed parallel implementations of CLARA for a 10 processor system [14, 16], which we discuss further in Section 3.2. They report good performance but not specific speedups.

3. MAKING PARALLEL CLUSTERING VIABLE FOR PERFORMANCE TOOLS

Not only has previous work on parallel clustering only targeted small processor counts, it has focused on dedicating the processors to the task of clustering very large data sets. In contrast, parallel performance tools must work in large-scale environments in which the processors primarily execute the measured applications. Further, parallel performance data are distributed across the system, so comparison of any two objects usually requires communication. A naive solution could try to aggregate all performance data to one process and cluster it sequentially. In the context of performance tools, the amount of data on each node is moderate, but the total size is large because of the large number of processes. Since the goal of clustering in this scenario is to reduce the amount of data communicated and stored, this approach is unacceptable.

This section describes our Clustering Algorithm with Parallel Extended K-Medoids (CAPEK), which clusters distributed performance data with $O(\frac{n}{P}log(P))$ complexity for P processors. For performance tools, where n is $O(P)$, CAPEK has $O(log(P))$ complexity, making it suitable for low-overhead, run time data analysis.

We base our parallel clustering algorithm on the classic CLARA algorithm [19] (Figure 1). CLARA takes an object set X, a dissimilarity function $d : X \times X \rightarrow \mathbb{R}$ and a number of clusters $k \in \mathbb{N}$ as input. It produces a *clustering*, a set of disjoint clusters $C = C_1, \ldots, C_k \subseteq X$ such that $\bigcup_{i=1}^{k} C_i = X$ and a set of medoids $M = m_1, \ldots, m_k$ such that $m_i \in C_i$. Each m_i is the representative element for cluster C_i.

3.1 CLARA Overview

CLARA begins by selecting s elements of X randomly, and stores these in a subset X'. The algorithm then builds a dissimilarity matrix for the subset and invokes PAM on it, yielding a clustering C' and a set of medoids M. CLARA discards C' and only uses M to generate a clustering C by assigning each element $x_i \in X$ to the cluster $C_j \in C$ that minimizes $d(m_j, x_i)$. That is, it assigns each object to its nearest medoid. It then computes the total intracluster dissimilarity for C by summing the dissimilarity of each x_i and the medoid of its assigned cluster. CLARA repeats this process S times and then returns the clustering with the lowest total dissimilarity.

CLARA has time complexity $O(n + s^2 S)$, which is asymptotically linear because the sampling parameters s and S are constant in the limit. Kaufman *et al.* found that a sample of $s = 40 + 2k$ and a trial count of $S = 5$ work well for very large data sets [19].

3.2 CAPEK: Parallel Sampled Clustering

CAPEK, shown in Figure 2, parallelizes CLARA with little synchronization and low parallel complexity. Like CLARA, it takes a set X of $n \in \mathbb{N}$ objects to be clustered and a distance function $d : X \times X \rightarrow \mathbb{R}$ as input. It may also take a number of clusters k, or the user may omit k and use a clustering criterion to select an ideal k from a range of values instead. We discuss this mode of execution further in 3.3. We assume that the elements of X are distributed over P processes. For simplicity, Figure 2 shows the case where $n = P$ and $k = 3$, although we do not require these restrictions.

CAPEK first generates S arbitrary subsets of X using deterministic pseudorandom number generators with the same seed on each process. Every process generates the full index set of sampled objects for each trial, which implements a global selection process without synchronization. We similarly select a group of S worker processes and collect a different sample set on each. The total number of samples to be aggregated, $s \cdot S$, and the efficiency with which we can aggregate S sets of s objects to S workers bound the complexity of this step. Sample sizes s and S are again asymptotically constant and the efficiency of gathering these data to the worker processes is at most logarithmic in P.

In the third step of our algorithm, each worker runs the PAM partitioning algorithm on its sample of X. As with CLARA, this is $O(s^2)$ and, thus, constant for large systems. Each worker determines a set of M medoids, which we broadcast from the S worker processes to all P processes. The collective broadcast operations can be pipelined, but take worst-case time $S \cdot log(P)$, so they have complexity $O(log(P))$.

The broadcasts ensure that all P processes have the medoid sets found by each PAM trial run. For each trial, each process finds the medoid nearest to its object, which parallelizes CLARA's $O(n)$ ASSIGN-MEDOIDS calls without additional synchronization. Since we have $n = P$ processors, the complexity is $O(Sk) = O(1)$.

After the local medoid calculation, each process has a vector of distances to its objects' nearest medoids. We compute the total dissimilarity for each clustering with a global sum of these vectors using a parallel reduction followed by a broadcast of complexity $O(log(P))$. Each process now has the global dissimilarity sum for each potential clustering. Thus, each process can locally identify the best clustering and assign its objects to the nearest medoids with complexity $O(S)$, or constant time.

When $n = P$, CAPEK has overall complexity $O(log(P))$. Our experiments on large systems show that its most expensive component is the constant, but significant, time required to compute local PAM trials. However, even with this limitation, its run time is sufficiently low for performance tools to perform on-line clusterings at run time without significantly perturbing the application. Nonetheless, CAPEK is more general: if we can determine in constant time which process owns an object from its index, then it has complexity $O(\frac{n}{P}log(P))$. This condition may not hold for more complex data distributions. Poorly balanced distributions of X may incur a performance penalty.

With the parallelization of S independent sample sets, CAPEK follows a similar idea to a prior parallel implementation of CLARA for the 10-processor IBM lCAP 1 system [14, 16]. However, this

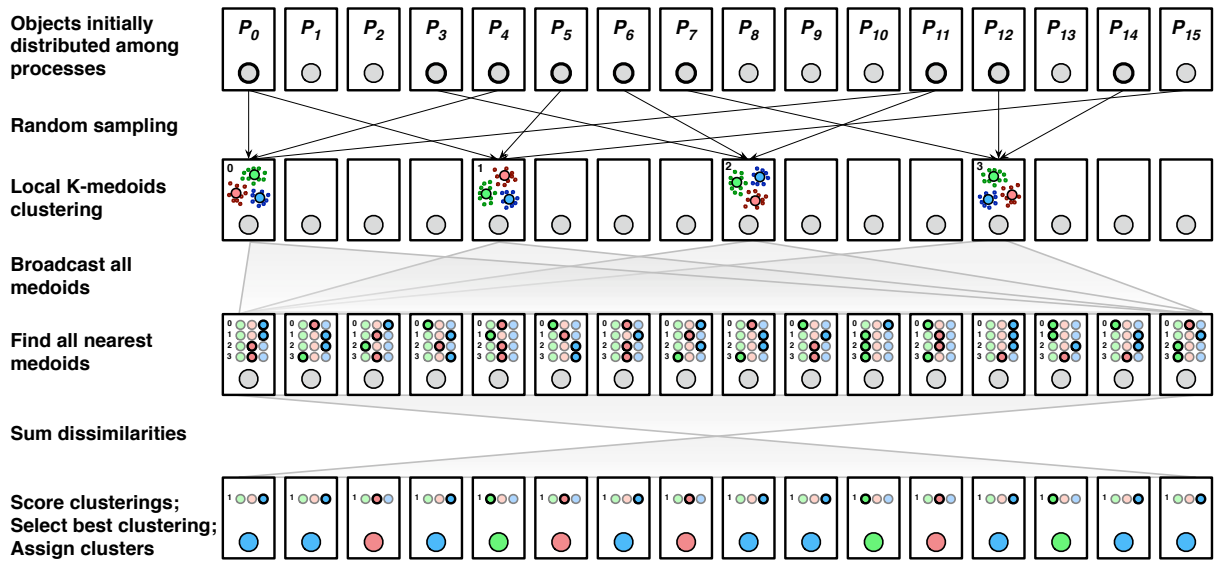


Figure 2: CAPEK, our parallel K-Medoids clustering algorithm

approach required each worker to compute both the $O(s^2)$ PAM trials *and* the $O(n)$ medoid assignment steps locally. While this is acceptable at small scales for which this approach was designed, it fundamentally does not scale to extremely large data sets, as each worker fully traverses a copy of the entire input data. In contrast, our algorithm aggressively targets and parallelizes all $O(n)$ steps and therefore provides significantly better scalability.

We have implemented our algorithm using C++ templates with MPI [29], and X may be any generic data type, as long as it supports special functions for transmission using MPI.

3.3 Parallel Clustering Criteria

CAPEK can be run with either a fixed value for k, or it can use a clustering criterion to select an ideal k. We have implemented k-selection using the Bayesian Information Criterion (BIC) [33, 37]. The BIC is a maximum likelihood model that considers a clustering as a set of spherical gaussians centered at the clustering's medoids. A clustering's BIC score is a function of the likelihood that the clustered data was generated by the model and the number of clusters in the model. The BIC penalizes models with two many clusters, so that the clustering that best describes the data with the *fewest* clusters will have the best score.

The BIC has been applied as part of the *X-Means* partitioning algorithm [33] to generalize K-Means clustering, and the X-Means algorithm has become popular in the literature. Since CAPEK is a sampled K-Medoids algorithm, the accuracy of the gaussian model may be reduced if medoids are slightly off-center within their clusters. However, we have not found this to be an issue in practice.

Using the BIC adds a small constant overhead to CAPEK, but it does not affect CAPEK's worst case complexity. To run CAPEK with the BIC, we conduct additional parallel PAM trials to search for an ideal k (S trials for each value in a range of reasonable k values), which parallelizes the approach used in the X-Means algorithm. If P is large, CAPEK can afford to perform many more PAM trials without affecting its run time. Even for $P = 16$ as in Figure 2, many processes are idle during the K-Medoids clustering step. Increasing the range of k simply adds more trials to this embarassingly parallel step. Since we run these trials on otherwise idle processes, they do not increase the complexity. If $k \ll P$, as holds for many clustering applications, then the total number of sampled objects that we must transfer during the first communication step is still much smaller than the total system size, and the asymptotic communication complexity does not increase.

After computing medoids for all PAM trials, we broadcast them to all processors, just as we would with fixed k. However, instead of measuring total intracluster dissimilarity, we now apply the BIC to each clustering, and we choose a clustering with the minimal BIC score instead of one with minimal dissimilarity. The fixed-k version of CAPEK can be considered a special case of the BIC version that uses intracluster dissimilarity as its clustering criterion instead of the BIC. Parallel implementations of both criteria require only simple $O(log(P))$ reduction operations, so both have the same asymptotic complexity. Evaluating the BOC in parallel costs slightly more than parallel intracluster dissimilarity, as the BIC requires several reductions instead of a single global sum.

4. EFFORT DATA

To demonstrate the application of our algorithm to real performance data, we apply it to data collected using the *Effort Model* of Gamblin *et al.* [9]. This model describes load-balance in single-program, multiple-data (SPMD) applications using two constructs:

Progress loops indicate absolute headway towards some application goal. Each each iteration is a global, synchronous step toward completion. The progress loop of an SPMD simulation is usually its outer (main) loop. For example, each progress step of a climate simulation computes the physics for some known interval of simulated time.

Effort regions are variable-time code regions with possibly data-dependent execution, nested within progress steps. Effort regions may be steps of a convergent algorithm (e.g., the conjugate gradient method) or they may integrate over variable-size, or adaptively-refined partitions. We measure the work required for each progress step by timing effort regions.

To ensure that our instrumentation is lightweight, we collect timings for coarse-grained effort regions, each of which is the dynamic execution path between two significant delimiting events. In this

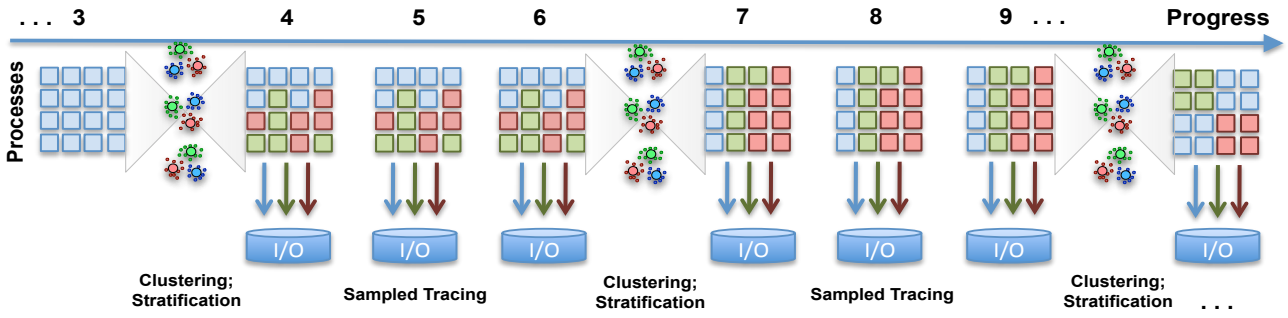


Figure 3: Using clustering to stratify sampled effort traces

work, we use MPI calls as delimiters, though other choices are possible. We use low-overhead, link-level instrumentation techniques, such as PMPI and P^{N}MPI [36], in conjunction with stack-walking tools [32], to monitor effort regions at run time.

Progress steps provide a semantic context in which to compare effort values across processes and over time. Comparison of the same progress step on two processes indicates differences in effort (work) and, thus, in load between the processes. Similarly, comparison of different progress steps of a single process captures load evolution. If effort values increase over time, either the application model is growing or the application is becoming less efficient.

Our work extends this effort analysis. We compare the temporal behavior of processes through *effort traces*, which are effort value vectors over successive time steps. Sections 5 and 6 describe scalable analysis techniques to cluster and to sample these traces.

5. STRATIFIED TRACE SAMPLING

Stratified trace sampling [10] is a technique for scalably gathering performance trace data by reducing I/O bottlenecks. Prior work has shown that off-line clustering can modestly improve the performance of stratified trace sampling, and we demonstrate in Section 7.4 that CAPEK can improve the effectiveness of such techniques dramatically. This section reviews the basic ideas of sampled tracing to provide context for our experiments.

5.1 Sampled Tracing

The AMPL trace library [10] uses population-sampling techniques to select representative subsets of large numbers of processes for tracing. This reduces performance data volume and load on the I/O system of large clusters. AMPL collects effort traces on all processes at run time, and it uses the notion of a *mean trace* as a heuristic to select a representative subset of the full trace data. The user of AMPL specifies confidence and error bounds on how accurately the mean of the subset should estimate the mean of the full data set, and AMPL dynamically adjusts the number of processes that should record effort traces to ensure that these bounds are met. This dynamic adjustment is called *trace sampling*.

Trace sampling is effective because the sample set size depends on the variance of the data. If the variance is low, we only need to collect a small subset of the full data. Further, the minimum sample size n required to sample P processes with AMPL is constant in the limit for a fixed population variance. Given a confidence interval z_α and an absolute (unnormalized) error bound d,

$$n \geq P\left[1 + P\left(\frac{d}{z_\alpha \sigma}\right)^2\right]^{-1} \quad (1)$$

where σ is the standard deviation of the sampled data. In the limit, the sample size n approaches $(z_\alpha \sigma / d)^2$, which is constant, given a fixed standard deviation. Thus, the ratio of sampled processes to total system size is n/P, which *decreases* in the limit.

The AMPL approach has been shown to reduce trace data volumes by up to an order of magnitude on systems of up to 2,000 processes, but does not scale to the hundreds of thousands or more processes on the largest current machines, let alone millions of processes, as will exist on exascale machines.

5.2 On-line stratified sampling

Equation 1 implies that if the variance of the trace data is low enough, then we can monitor an entire population of processes by sampling a small subset. However, if variance is high, then we may be forced to trace too many processes, and this can easily overwhelm the I/O system and perturb monitored applications. We can reduce the needed sample size by using *stratification*, a technique frequently used in statistical surveys. Specifically, if we can group processes into subgroups, or *strata*, each with *lower* variance than the system as a whole, then sample these strata independently, then we can lower the total sample size needed to monitor all processes.

Clustering is a natural way to divide processes into low-variance strata, so we use that approach. Given a clustering C of process traces $P_1 ... P_n$, with clusters $C_1 ... C_k$, cluster means $\bar{P}_1 ... \bar{P}_k$ and standard deviations $\sigma_1 ... \sigma_k$, a stratified sample is more efficient than an unstratified sample if

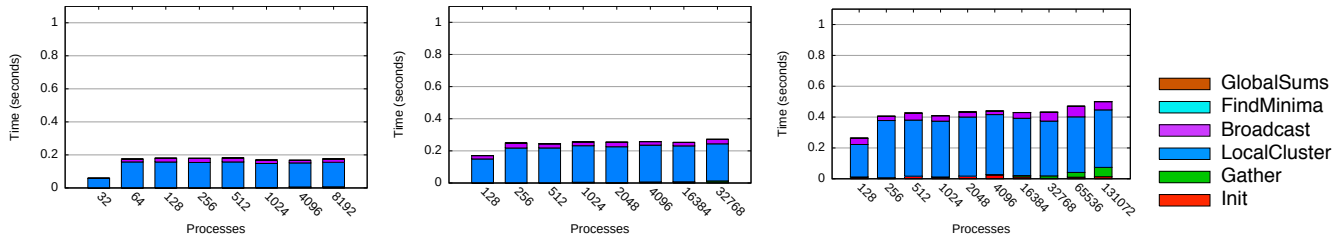
$$\sum_{i=1}^{k} P_i (\bar{P}_i - \bar{P})^2 > \frac{1}{P} \sum_{i=1}^{k} (P - P_i)\sigma_i^2 \quad (2)$$

In other words, a total variance *within* the clusters that is lower than the variance *between* the clusters leads to a more efficient sample.

In prior work with AMPL, we applied sequential clustering postmortem to detect strata in performance data [10]. We then re-ran the simulation using the resulting stratification, and we were able to reduce sample data volume by up to 25% over an un-stratified sample. In this work, we extend the AMPL approach using our *on-line* clustering algorithm to identify performance equivalence classes dynamically. We then adaptively adjust both the sample size and the stratification at run time.

Figure 3 shows a 16 process example that uses our extension to the AMPL approach. The horizontal axis of the figure corresponds to progress steps of the target system. Each process has its own effort trace. All processes are initially members of the same cluster. After three progress steps, we cluster the effort traces collected so far using our clustering algorithm with $k = 3$.

We then use the clustering results to divide the processes into k strata. The figure differentiates the members of the strata by the colors red, blue, and green. We use the AMPL approach to select



(a) 1 process per node **(b)** 2 processes per node **(c)** 4 processes per node

Figure 4: Time to cluster 2-dimensional points on BlueGene/P

the sample size for each stratum. The sampled processes of each stratum then write trace data at each progress step. We re-cluster the data after every three progress steps, which allows adaptation in response to evolving application behaviors.

Dynamic stratification is more effective than our prior static approach because multiple clusterings generated adaptively are a better fit to the application's performance data than the single, static clustering we generated using our previous approach. Further, with dynamic clustering, there is no need to re-run the simulation. This reduces the burden on the user of our tool.

The time required for cluster analysis is a potential concern for on-line analysis, but the scalability and low overhead of CAPEK makes it possible to use this kind of analysis with minimal overhead. In general, we will incur one second of overhead for each invocation of CAPEK. For codes with long progress steps, this should not be a problem even if we cluster once every three time steps, as in our example. In practice, if an application has short progress steps, we can reduce clustering overhead by clustering longer vectors of progress steps less frequently.

6. APPROXIMATE EFFORT DATA

When we decrease clustering frequency, we increase the clustered trace size. Longer traces imply more network traffic and may slow our clustering algorithm, especially the step that gathers random samples to worker processes. This section presents a technique to reduce the volume of a single-process effort trace while preserving key trace characteristics for clustering.

Wavelet transforms [4, 24] are often used in compression algorithms to reduce the entropy of signal data before applying other encodings. Gamblin *et al.* [9] used a parallel wavelet transform to reduce interprocess data volume while lossily preserving effort traces of all process. Although we sample the processes, we can use the wavelet transform to reduce single-process data.

Wavelet transforms have useful properties for trace clustering. First, they are local, unlike techniques such as the Fast Fourier Transform (FFT) [3]. Thresholding or sampling coefficients can result in large, global artifacts with the FFT. The output coefficients of the wavelet transform, however, have temporal locality, and preserve time-varying behavior even with high compression levels. We require this property since we target clusterings of traces that reflect their time evolution as well as the proximity of their means.

We can approximate the trace at a low resolution through the low-frequency coefficients of the wavelet transform. Instead of transfering a full effort trace from each sampled process across the network, we can send and cluster these low-frequency coefficients that serve as an *effort signature*.

Effort signatures have significant advantages. They preserve the locality of the original trace at greatly reduced data volumes. If the values collected in the original trace were steadily increasing or had a significant peak, then low-frequency wavelet coefficients also exhibit these features. We can apply the wavelet transform as many times as the input data can be recursively bisected, and each application halves the amount of data in the low-frequency bands. Thus, we can create effort signatures that are logarithmic in the original trace size. Our results demonstrate that can we use small effort signatures to discover clusterings of similar quality to those found from full trace data.

7. RESULTS

We conducted experiments with our tools on an IBM BlueGene/P (BG/P) system with 40,960 quad-core 850Mhz PowerPC 450 compute nodes. Each node has 2GB RAM (512 MB per core). A 3-D torus network and two tree-structured networks are available for communication between MPI processes. We used the xlC and gcc compilers and IBM's MPI implementation, based on MPICH-2. BG/P supports three modes of execution: *Symmetric Multiprocessing, or SMP mode*, where each node has exactly one MPI process, which may use threads on all 4 cores; *dual mode*, where each node has two MPI processes, each of which may use two cores; and *Virtual Node (VN) mode*, where each MPI process has exactly one thread and the application code must take interrupts to handle communication and I/O events.

While we tested our software in all BG/P run modes, none of our experiments used multithreaded code. Thus, the key differences between the modes are network bandwidth and shared access to the NIC. In SMP mode, one process may use the entire network bandwidth for the node, and the process has exclusive access to the NIC. In DUAL mode two processes must share the bandwidth and the NIC, and in VN mode, the bandwidth and NIC are shared by four processes. Shared NIC access may lead to lower messaging rates for DUAL and VN modes.

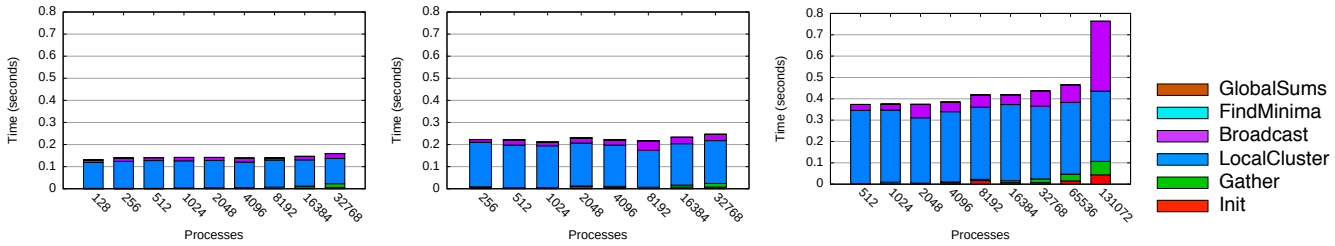
7.1 Scalability of CAPEK

To assess the scalability of CAPEK for clustering performance data, we performed scaling runs with up to 131,072 processes on the Blue Gene/P system with several different types of data.

First, we ran CAPEK on simple two-dimensional point data. To each process, we allocated a uniquely-seeded random number generator. We then generated two double-precision floating point coordinates and clustered the generated points using CAPEK. We used $k = 10$ and the Euclidean distance as our dissimilarity metric.

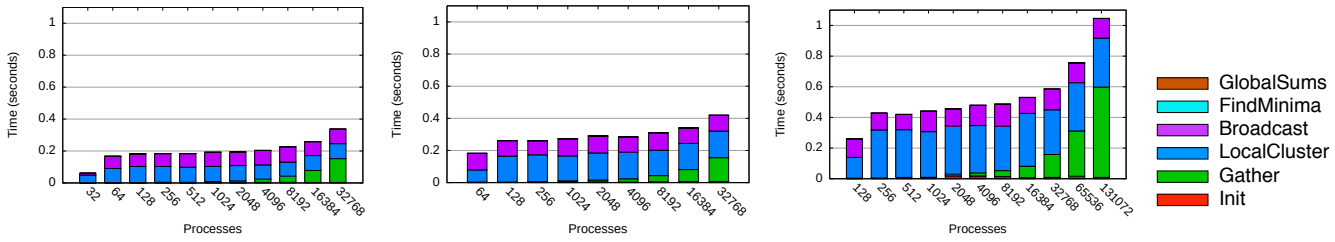
Figure 4 shows that CAPEK runs in slightly less than two tenths of a second in all cases under SMP mode, where the communication to compute ratio of the system is greatest. Performance is similar in dual mode. CAPEK runs in just over two tenths of a second on up to 32,768 cores. We were able to run tests on up to 131,072 cores in VN mode. Performance was slower in VN mode than in SMP or Dual mode, but CAPEK still ran in less than half a second on up to 131,072 cores.

The figure also shows the performance of particular phases of



(a) 1 process per node **(b)** 2 processes per node **(c)** 4 processes per node

Figure 5: Time to cluster 64-element effort traces on BlueGene/P



(a) 1 process per node **(b)** 2 processes per node **(c)** 4 processes per node

Figure 6: Time to cluster 2-dimensional points with experimental BIC for k-selection on BlueGene/P

CAPEK. The local clustering computation dominates in all cases and the trend is sub-linear in the number of processes in the system. At 131,072 cores, the initial gather begins to take more time, which is an artifact of our implementation. We currently use direct, asynchronous `MPI_Isend` operations to perform the gather operation instead of collectives. We are currently investigating alternative implementations using collectives efficiently for the some-to-some gather pattern required by our sampling scheme.

To assess CAPEK's behavior when applied to effort trace data, we generated synthetic effort traces and clustered them in parallel. Again, each process used a uniquely seeded random number generator to create synthetic data, but instead of two-dimensional point data, each process generated 64-element local effort traces. So that the data was not pure noise, we create our trace with six unique generator functions that include a constant-valued trace generator, a monotonically increasing trace generator, three sinusoids varying frequency and amplitude, and one sinusoid with dynamically varying frequency. We also added random noise to all of our trace functions to emulate real application data.

Figure 5 shows our synthetic signature clustering results. As before, we used the Euclidean distance to cluster the data, but traces were longer. For SMP and dual mode, the latency of our algorithm is nearly the same at about 0.2 seconds. At scale, the timings are nearly identical as well, and the constant time for local clustering dominates nearly all run times. However, while our 131,072-process run still finishes in under one second, the time required for the medoid broadcast phase is increasing due to a detail of our implementation. Currently, we use the first S MPI ranks as worker processes, but these are not laid out ideally for a large-scale torus network. By default, contiguous ranks are laid out near each other in the BG/P torus network, and contention for links around the worker processes causes the broadcast overhead we see in our 131,072-process run. We could avoid this overhead by spreading the worker ranks out in the torus network.

As a final test of our algorithm's robustness, we performed scaling runs of CAPEK using the BIC for k-selection. In this variant, instead of supplying a specific k to test, we supplied a maximum k, k_{max}, and CAPEK ran with BIC enabled for $k \in 1..k_{max}$. It then performs $5 \cdot k_{max}$ trials, 5 for each possible value of k. This variant makes use of more cores than the standard version but generates more messages during the gather and broadcast phases.

Figure 6 shows timings for our k-selection variant. Like our standard algorithm, it scales to 131,072 cores (VN mode) and runs in around one second. In SMP and DUAL modes, it runs in approximately 0.4 seconds. However, at scale, particularly in VN-mode, the initial gather phase of our algorithm begins to dominate the runtime. We analyzed this problem and found that the additional time is due to our implementation of random sampling. We currently use Knuth's Algorithm R [20] to generate a set of s samples from n object indices. However, Algorithm R has $O(n)$ time complexity, which dominates CAPEK's run-time at scale.

This problem did not present itself prominently in the fixed-k version of CAPEK because the fixed-k version requires many fewer workers than the BIC version. We currently run this algorithm once per worker sequentially, the overhead of our sampling algorithm is amplified when we use the BIC. We do see that the time required for the gather phase grows linearly with scale in the fixed-k timings (Figure 5), but it does not grow nearly as quickly as it does for the BIC version (Figure 6).

We can eliminate this overhead by generating worker sample sets concurrently using a parallel random number generator such as SPRNG [25]. We leave this implementation for future work. At current scales, our algorithm still runs in approximately one second, even with BIC, though we will likely need to eliminate this overhead for acceptable performance on exascale machines.

7.2 Performance with effort approximations

We conducted experiments to quantify the impact of the additional communication CAPEK's performance due to having more or larger input objects that we observed in our initial experiments with effort traces. Figure 7 shows data for clustering effort traces from 512 to 8,192 elements long on 512 to 65,536 Blue Gene/P cores. Increasing trace size does not affect CAPEK's scalability for sizes less than 131,072 processes. The algorithm exhibits nearly

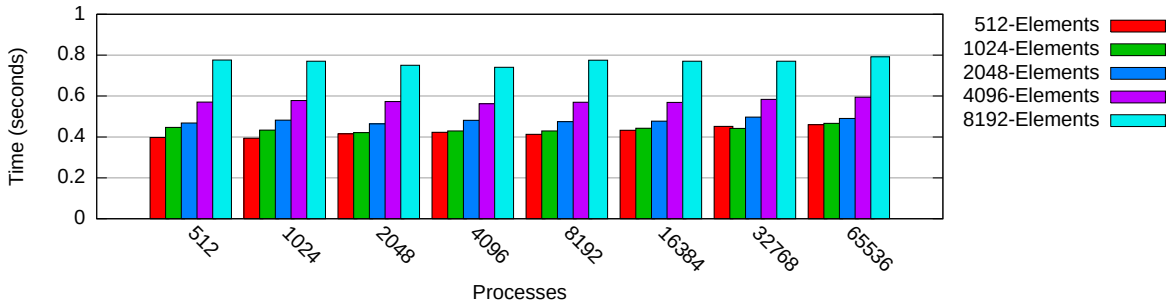
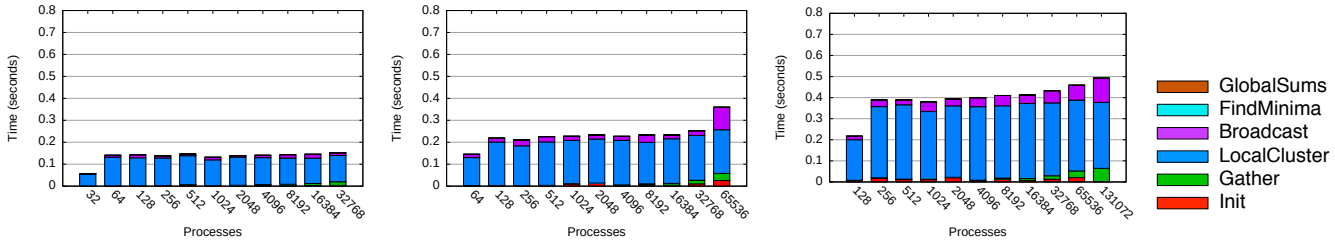


Figure 7: Time to cluster longer effort traces on BlueGene/P



(a) 1 process per node **(b)** 2 processes per node **(c)** 4 processes per node

Figure 8: Time to cluster 16-element effort signatures on BlueGene/P

perfect weak scaling, regardless of the size of the traces clustered.

Trace size does not affect CAPEK's complexity, as it is a multiplicative constant in our worst-case running time. However, we are concerned with this constant for tools that do not reduce data size before using CAPEK, because it can increase clustering latency. As we discussed in Section 6, we can remedy this by clustering the *effort signatures* of larger traces rather than the full trace.

Figure 8 shows the results of clustering 16-element effort *signatures* derived from the 64-element effort *traces* that we clustered in Figure 5. Using this optimization halves CAPEK's run time at 131,072 cores. The congestion that we observed when running at this scale has also been mitigated significantly, and the run time is again sub-linear in the limit.

7.3 Cluster quality

To verify that CAPEK produces high-quality clusterings even with sampling and effort approximations, we ran trials of CAPEK and compared the results to those produced by PAM. We use a formal metric, the *Mirkin Distance* [28, 26], to compare clusterings. Given a clustering C with k clusters, each cluster C_i contains n_i objects, such that $|C_i| = n_i$, and $\sum_{i=1}^{k} n_i = n$. Given two clusters, C_i and C'_i, let the number of points in their intersection be:

$$n_{ii'} = |C_i \cap C_{i'}| \tag{3}$$

Given two clusterings on the same data, C and C', the Mirkin distance between them is:

$$\mathrm{d'}_M(C, C') = \sum_{i=1}^{k} n_i^2 + \sum_{i'=1}^{k} n_{i'}'^2 - 2\sum_{i=1}^{n}\sum_{i'=1}^{k} n_{ii'}^2 \tag{4}$$

The Mirkin distance measures the number of pairs of points that are in the same cluster in C but in different clusters in C'. For our comparisons, we use the normalized Mirkin distance, which is invariant to the data set size and can be used as a measure of error:

$$\mathrm{d}_M(C, C') = \frac{d'_M(C, C')}{n^2} \tag{5}$$

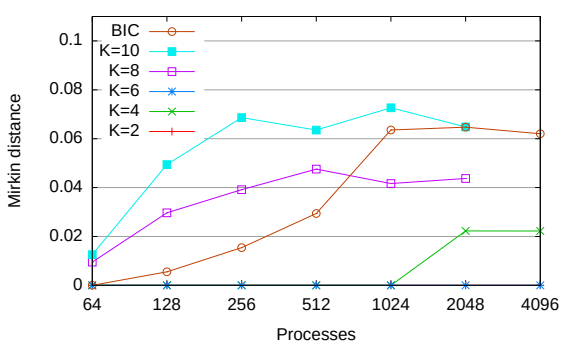
We first ran trials of CAPEK on synthetic effort signature data with 64 to 4,096 processes of Blue Gene/P and even-valued fixed k from 2 to 10. We also performed similar trials using the Bayesian Information Criterion to select a k. We compared sampled clusterings generated using these settings to optimal clusterings generated using the sequential PAM algorithm. As in Section 7.1, our data contained 6 types of signatures. Figure 9a shows the normalized Mirkin distance between the clusterings found by PAM and the clusterings found by CAPEK. For small systems of 64 processes, the normalized Mirkin distance is below .01 for all values of k. CAPEK clusterings incur more error with increasing scale, but the Mirkin distance stays near 5% for $k = 8$ and near 7% for $k = 10$ for 256 or more processes. For $k = 4$, the Mirkin distance is never more than 3%, and for $k = 6$ and $k = 2$ CAPEK and PAM produce identical clusters. Using the BIC to estimate k, the error is very low for small runs, then increases between 512 and 1024 proceses and finally levels off at 10% for $k \geq 10$.

When clustering effort signatures with varying approximation levels, as in Figure 9b, CAPEK's clustering quality is insensitive to the level of effort approximation used for nearly all values of k. For k = 4, we see error rates of 22% because the data actually has 6 clusters. CAPEK *must* find exactly k clusters and splits or merges real clusters into k when the data has a different number of clusters. With small errors in distance introduced by approximation, many groups may have approximately the same dissimilarity but very different memberships. The Mirkin distance is sensitive to these changes. We see that when k is at the natural cluster count of 6 (or above), error introduced by effort signatures is less than 7%.

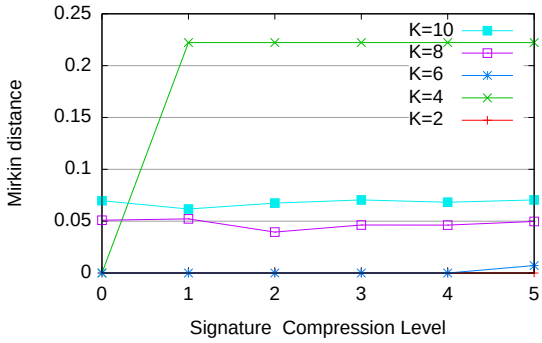
7.4 Data volume

We applied our stratified sampling tool to S3D [13], a Department of Energy gas-dynamics code that solves the Navier-Stokes equations for turbulent combustion problems. In general, the code exhibits uniform performance behavior and achieves near-perfect weak scaling. However, processes within S3D are arranged in a rectangular prism, and those on the edges, faces, and corners may behave differently from internal processes.

We collected effort traces for time spent in the y-direction derivative computation in S3D's solver. We used CAPEK to adaptively cluster traced processes, and we used AMPL to collect a stratified



(a) CAPEK vs. PAM, varying system size



(b) CAPEK with effort approximation vs. PAM

Figure 9: Error relative to PAM with CAPEK

Processors	Clusters							
	1		3		6		9	
	Size	%	Size	%	Size	%	Size	%
4096	4095	99%	5	0.12%	9	0.21%	14	0.34%
8192	8191	99%	5	0.06%	10	0.12%	14	0.17%
16384	16383	99%	6	0.036%	10	0.061%	17	0.10%
32768	Not tested	Not tested	6	0.018%	13	0.040%	18	0.055%
65536	Not tested	Not tested	4	0.006%	10	0.015%	13	0.020%

Table 1: Cost of sampling time spent in S3D's y derivative solver with bounds of 95% confidence, 5% error

sample using the clusters generated by CAPEK. For our traces, we used a custom, uncompressed ASCII trace format with two callsite identifiers and a double-precision floating point number per effort region. We ran CAPEK at synchronous points in S3D and used a separate MPI communicator so that CAPEK did not interfere with S3D communication.

We focused on the y-solver because the naive implementation of AMPL performed very poorly on this region of the code due to high inter-cluster variability resulting from differences between edges, faces, corners, and internal processes. Specifically, with 95% confidence and 5% error, AMPL estimates that nearly 100% of processes must be sampled to estimate the time spent in the y-solver. This is clearly not scalable.

Table 1 shows the sample sizes estimated using AMPL sampling with CAPEK stratification. For these tests, we ran CAPEK with several fixed k values. With one stratum, our results match those for the AMPL technique. For S3D runs of 4,096, 8,192, and 16,384 processes, we must sample nearly all processes to record an "accurate" trace. We did not attempt to single-stratum traces for larger system sizes because of the potentially enormous I/O volume.

Our results are much better when using CAPEK to find strata. With only three strata, CAPEK detects strong equivalence classes among processes. This stratification strategy reduces the intra-cluster variance sufficiently so that we only need to sample 4 processes for an accurate trace of a 65,536-process run. Sample sizes are slightly larger for runs with 6 and 9 strata, but the sizes are still less than 0.5% of the total system size, and our approach reduces the trace storage requirements by over four orders of magnitude for the 65,536-process run. For our largest runs, the total trace size was 36 GB with data reduction; a full trace would consume over 3 TB. Moreover, the sample size is constant in the limit, and the potential reduction of data volume on exascale systems will be far larger.

For S3D, the overhead of our approach is 1-3%, which primarily arises from the MPI instrumentation. CAPEK overhead is negligible because we run the algorithm infrequently. Figure 7 shows that we could cluster larger groups of time steps to keep CAPEK overhead low for codes with shorter (wall clock) time steps.

Overhead from tracing is also negligible because our clustering is very effective at finding strata in the S3D trace data, which allows us to write out very few traces for our largest runs. This is a considerable improvement over our previous work with AMPL tracing, in which trace overhead exceeded 100% for some runs.

8. CONCLUSION

Exascale systems will have hundreds of millions of processors and billions of concurrent tasks. More than ever, application developers for these systems will need detailed measurements to analyze, to understand and to tune their applications' performance. However, even the tools to collect this data must be finely tuned parallel applications at this scale. Scalable on-line analysis techniques are thus critical for the success of exascale systems.

We have introduced CAPEK, a massively scalable parallel partitioning algorithm with $O(\frac{n}{P}log(P))$ complexity, or $O(log(P))$ complexity when $n = P$, as is the case for parallel performance tools. To our knowledge, CAPEK is the only clustering algorithm to have scaled to 131,072 processors cores, and its implementation is fully generic. We have also applied our algorithm to the *effort model* for parallel load balance data and we have shown that CAPEK can efficiently cluster effort traces using *effort signatures* to reduce trace data before clustering. Further, we have applied the Bayesian Information Criterion (BIC) to eliminate the need for users of CAPEK to specify the number of clusters k in advance.

Combining these novel techniques, we have applied our performance trace clustering technique to an existing trace sampling framework to enable efficient on-line stratification of performance data. Finally, we showed that using adaptive stratification, performance trace data from a 65,556-process parallel application may be reduced by up to four orders of magnitude, with potential for more data reduction at larger scales.

Acknowledgments

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. These resources were made available via the Performance Evaluation and Analysis Consortium End Station, a U.S. DOE INCITE project. Neither Contractor, DOE, or the U.S. Government, nor any person acting on their behalf: (a) makes any warranty or representation, express or implied, with respect to the information contained in this document; or (b) assumes any liabilities with respect to the use of, or damages resulting from the use of any information contained in the document.

9. REFERENCES

[1] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, K. Hill, J. Hiller, S. Karp, C. Koelbel, D. Koester, P. Ko, J. Levesque, D. A. Reed, V. Sarkar, R. Schreiber, M. Richards, A. Scarpelli, J. Shalf, A. Snavely, and T. Sterling. ExaScale software study: Software challenges in extreme scale systems-. Technical report, DARPA IPTO, September 14 2009.

[2] D. Arthur and S. Vassilvitskii. How slow is the k-means method? In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 144–153, New York, NY, USA, 2006. ACM.

[3] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[4] I. Daubechies. *Ten Lectures on Wavelets*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1992.

[5] Z. Du and F. Lin. A novel parallelization approach for hierarchical clustering. *Parallel Comput.*, 31(5):523–527, 2005.

[6] E. W. Forgy. Cluster analysis of multivariate data: efficiency *vs.* interpretability of classifications. *Biometrics*, 21:768–769, 1965.

[7] G. Forman and B. Zhang. Distributed data clustering can be efficient and exact. *SIGKDD Explor. Newsl.*, 2(2):34–38, 2000.

[8] G. Forman and B. Zhang. Linear speedup for a parallel non-approximate recasting of centerbased clustering algorithms, including k-means, k-harmonic means, and em. Technical Report HPL-2000-158, HP Laboratories, Palo Alto, CA, December 4 2000.

[9] T. Gamblin, B. R. de Supinski, M. Schulz, R. J. Fowler, and D. A. Reed. Scalable load-balance measurement for SPMD codes. In *Supercomputing 2008 (SC'08)*, pages 46–57, Austin, Texas, November 15-21 2008.

[10] T. Gamblin, R. J. Fowler, and D. A. Reed. Scalable methods for monitoring and detecting behavioral classes in scientific codes. In *Proceedings of the 22nd International Parallel and Distributed Processing Symposium (IPDPS 2008)*, pages 1–12, Miami, FL, April 14-18 2008.

[11] A. Geist and R. Lucas. Major computer science challenges at exascale. *Int. J. High Perform. Comput. Appl.*, 23(4):427–436, 2009.

[12] J. A. Hartigan and M. A. Wong. Algorithm as 136: A K-Means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[13] E. R. Hawkes and J. H. Chen. Direct numerical simulation of hydrogen-enriched lean premixed methane–air flames. *Combustion and Flame*, 138:242–258, 2004.

[14] P. K. Hopke. The application of supercomputer to chemometrics. In E. J. Karjalainen, editor, *Proceedings of the Scientific Computing and Automation (Europe) Conference*, Masstricht, The Netherlands, June 1990. Available from: http://books.google.com/books?id=NtAAaXpJBZcC.

[15] K. A. Huck and A. D. Malony. PerfExplorer: A performance data mining framework for large-scale parallel computing. In *Supercomputing 2005 (SC'05)*, page 41, Seattle, WA, November 12-18 2005.

[16] L. Kaufman, P. K. Hopke, and P. J. Rousseeuw. Using a parallel computer system for statistical resampling methods. *Computational Statistics Quarterly*, 2:129–141, 1988.

[17] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*, chapter 5, pages 199–252. Wiley Series in Probability and Statistics. Wiley-Interscience, 2nd edition, 2005.

[18] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*, chapter 2, pages 68–125. Wiley Series in Probability and Statistics. Wiley-Interscience, 2nd edition, 2005.

[19] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*, chapter 3, pages 126–163. Wiley Series in Probability and Statistics. Wiley-Interscience, 2nd edition, 2005.

[20] D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms, page 144. Addison-Wesley Longman Publishing Co., 3rd edition, 1997.

[21] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. A. Yelick. ExaScale computing study: Technology challenges in achieving exascale systems. Technical report, DARPA IPTO, September 28 2008.

[22] S. P. Lloyd. Least squares quantization in PCM. technical note, Bell Laboratories. *IEEE Transactions on Information Theory*, 28:128–137, 1967, 1982.

[23] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. L. Cam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. Univeristy of California Press, June 21-July 18 1967.

[24] S. Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, 2nd edition, September 1999.

[25] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26:436–461, 2000.

[26] M. Meilă. Comparing clusterings: an axiomatic view. In *Proceedings of the 22nd International Conference on Machine Learning (ICML '05)*, pages 577–584, New York, NY, USA, 2005. ACM.

[27] C. L. Mendes and D. A. Reed. Monitoring large systems *via* statistical sampling. *International Journal of High Performance Computing Applications*, 18(2):267–277, May 2004.

[28] B. Mirkin. *Mathematical Classification and Clustering*. Kluwer Academic Publishers, 1996.

[29] MPI Forum. MPI: A message passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, 1994.

[30] F. Murtagh. *Multidimensional Clustering Algorithms*. Physica-Verlag, 1985.

[31] C. F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21:1313–1325, 1993.

[32] Paradyn Project. *DynStackwalker Programmer's Guide*. Madison, WI, July 13 2007. Version 0.6b. Available from: http://ftp.cs.wisc.edu/pub/paradyn/releases/current_release/doc/stackwalker.pdf.

[33] D. Pelleg and A. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734, 2000. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.3377.

[34] S. Ranka and S. Sahni. Clustering on a hypercube multicomputer. *IEEE Trans. Parallel Distrib. Syst.*, 2(2):129–137, 1991.

[35] P. C. Roth. Etrusca: Event trace reduction using statistical data clustering analysis. Master's thesis, University of Illinois at Urbana-Champaign, 1996.

[36] M. Schulz and B. R. de Supinski. P^NMPI tools: A whole lot greater than the sum of their parts. In *Supercomputing 2007 (SC'07)*, Reno, NV, November 11-16 2007.

[37] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro: Micro's Top Picks from Computer Architecture Conferences*, November-December 2003.

[38] B. Wang, Q. Ding, and I. Rahal. Parallel hierarchical clustering on market basket data. In *ICDMW '08: Proceedings of the 2008 IEEE International Conference on Data Mining Workshops*, pages 526–532, Washington, DC, USA, 2008. IEEE Computer Society.