SCALAEXTRAP: Trace-Based Communication Extrapolation for SPMD Programs

XING WU, FRANK MUELLER, North Carolina State University

Performance modeling for scientific applications is important for assessing potential application performance and systems procurement in high-performance computing (HPC). Recent progress on communication tracing opens up novel opportunities for communication modeling due to its lossless yet scalable trace collection. Estimating the impact of scaling on communication efficiency still remains non-trivial due to execution-time variations and exposure to hardware and software artifacts.

This work contributes a fundamentally novel modeling scheme. We synthetically generate the application trace for large numbers of nodes by extrapolation from a set of smaller traces. We devise an innovative approach for topology extrapolation of single program, multiple data (SPMD) codes with stencil or mesh communication. Experimental results show that the extrapolated traces precisely reflect the communication behavior and the performance characteristics at the target scale, for both strong and weak scaling applications. The extrapolated trace can subsequently be (a) replayed to assess communication requirements before porting an application, (b) transformed to auto-generate communication benchmarks for various target platforms, and (c) analyzed to detect communication inefficiencies and scalability limitations.

To the best of our knowledge, rapidly obtaining the communication behavior of parallel applications at arbitrary scale with the availability of timed replay, yet without actual execution of the application at this scale is without precedence and has the potential to enable otherwise infeasible system simulation at the exascale level.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement Techniques; C.4 [**Performance of Systems**]: Modeling Techniques

General Terms: Experimentation, Tracing, Compression

Additional Key Words and Phrases: Communication, Tacing, Compression, Trace Extrapolation

ACM Reference Format:
Wu, X., Mueller, F. 201?. SCALAEXTRAP: Trace-Based Communication Extrapolation for SPMD Programs. ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 26 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Scalability is one of the main challenges for scientific applications in HPC. A host of automatic tools have been developed by both academia and industry to assist in communication gathering and analysis for MPI-style message passing [Gropp et al. 1996]. Most of these tools either obtain lossless trace information at the price of poor scalability [Nagel et al. 1996] or preserve only aggregated statistical trace information to limit the size of trace files as in mpiP [Vetter and McCracken 2001]. Recent work on communication tracing and time recording made a breakthrough in this realm. SCALATRACE introduced an effective communication trace representation and compression algorithm [Noeth et al. 2009]. It managed to preserve the structure and temporal ordering of events, yet maintains traces in a space-efficient representation. However, SCALATRACE needs to be linked to the original application and executed on a high-performance computing cluster of a *given number*

We would like to thank the Juelich Supercomputing Centre for giving us access to their Blue Gene/P system.
This work was supported in part by NSF grants 0937908 and 0958311.
Authors' address: Department of Computer Science, North Carolina State University, Raleigh, NC 27695-8206.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 0164-0925/YYYY/01-ARTA $10.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

of compute nodes to obtain a trace. Due to the often long application execution times and limited availability of cluster resources for large numbers of nodes, obtaining the trace information of a large-scale parallel application remains costly.

An alternative to obtaining communication traces is to model and predict application behavior [Kerbyson et al. 2001; Kerbyson and Hoisie 2006]. Generally, this approach takes a number of machine and application parameters as input. It utilizes a set of formulae to assess the impact of scaling on the system characteristics and predict performance in terms of wall-clock runtime of an application. Similarly, this approach provides only overall statistics for an application on a particular architecture. Without a detailed application trace, more sophisticated static analysis is impossible. In addition, measuring the system and application performance parameters is also non-trivial given the complexity of supercomputers and large-scale scientific applications.

Contributions: This paper contributes a set of algorithms and techniques to extrapolate full communication traces and execution times of an application at larger scale with information gathered from smaller executions. Since extrapolation is based on analytical processing of smaller traces with mathematical transformations, this approach can be performed on a single workstation, much in contrast to analysis or visualization of large traces in contemporary tools (*e.g.*, VAMPIR Next Generation [Brunst et al. 2005]). It thus enables, for the first time, the instant generation of trace information of an application at arbitrary scale without necessitating time-consuming execution. Specifically, we extrapolate two aspects of the application behavior, namely the (1) communication trace events with parameters and (2) timing information resembling computation. The extrapolation of the communication trace is based on the observation that, in many regular SPMD stencil and mesh codes, communication parameters and communication groups are related to the sizes and dimensions of the communication topology. Thus, extrapolation of communication traces becomes feasible with the detection of communication topologies and the analysis of communication parameters to infer evolving patterns. The extrapolation of timing information involves a process of analytical modeling. In order to mitigate timing fluctuations under scaling, we employ statistical methods.

Our extrapolation methodology is applicable for both strong and weak scaling applications. Weak scaling is typically defined as scaling the problem size and the number of processors at the same rate such that the problem size per processor is fixed. This should imply that the communication patterns generally evolve in a similar manner for both strong and weak scaling. Thus, we hypothesize that the same extrapolation algorithms for patterns and communication end points should apply to both. For communication parameters, such as message sizes and computation times different trends can be observed. But we hypothesize that extrapolation based on curve fitting is still applicable. In this work, we verify these hypotheses by evaluating our extrapolation algorithm with both strong and weak scaling applications.

Our extrapolation methodology follows a trace analysis methodology independent of the tracing infrastructure and works for any of the existing trace formats. Nonetheless, the approach is significantly facilitated by SCALATRACE's compression scheme that preserves application structure with inherent compression that closely resembles the loop structure of an application. In contrast, extrapolation with other trace formats, such as OTF [Knüpfer et al. 2006], would be far more tedious and time/space consuming as structure is neither established across nodes nor retained after binary-level compression.

This trace extrapolation approach has been implemented in the SCALAEXTRAP tool, which we utilize to evaluate our extrapolation approach with both strong and weak scaling benchmarks, including NAS Parallel Benchmark codes [Bailey et al. 1991] and Sweep3D [Wasserman et al. 2000]. We utilize up to 16,384 nodes of a 73,728-node IBM Blue Gene/P supercomputer to generate communication traces for extrapolation and verification. Experiments were performed to assess both the correctness of communication extrapolation and the accuracy of the timing extrapolation. Experimental results demonstrate that our topology detection algorithm is capable of identifying and characterizing stencil/mesh and collective communication patterns. Upon topology detection, the communication trace extrapolation algorithm correctly extrapolates all communication events, pa-

rameters and communication groups at an arbitrary target size for both stencil/mesh point-to-point and collective communication. The experiments also demonstrate that the extrapolation of timing information resembles the running time of the original parallel application. Compared to the running time of the original application, the accuracy of replay times of the corresponding extrapolated trace is, in the majority of cases, higher than 90%, sometimes as high as 98%. Given the difficulty of extrapolating application execution time with only the time information obtained from several small executions, our approach achieves unprecedented accuracy that is sufficient for modeling, procurement and analysis tasks.

Overall, this work explores the potential to extrapolate communication behavior of parallel applications. Several novel algorithms for communication topology detection and communication trace extrapolation are introduced. Experimental results demonstrate that rapid generation of an application's trace information at arbitrary size is entirely possible, which is unprecedented. In contrast to tedious and application-centric model development, our approach opens new opportunities for automatically deriving communication models, facilitating communication analysis and tuning at any scale. Our work further enables system simulation at extreme scale based on a single file, concise communication trace representation. More specifically, HPC simulation tools (*e.g.*, Dimemas or SST [Labarta et al. 1997; Snavely et al. 2002; Rodrigues et al. 2006]), which currently cannot operate at petascale levels, could benefit by utilizing our extrapolated single-file traces that are just 10s of megabytes in size. Benchmark generation is important for cross-platform performance analysis due to its standard and portable source code and the platform-independent nature. Our work enables code generation at extreme scale by providing large traces that are otherwise unavailable. Furthermore, by contributing a set of detection techniques of communication patterns, our work has the potential to enable the generation of flexible and stand-alone programs that can be executed with arbitrary numbers of nodes and any possible input.

This paper is structured as follows. Section 2 summarizes related work on ScalaTrace with respect to its ability to support extrapolation. Section 3 provides a detailed introduction to the algorithms designed for extrapolation. Sections 4 and 5 present the experimental framework and results. Section 6 provides two case studies to demonstrate the potential applications of trace extrapolation. Section 7 contrasts this work with prior research. Section 8 summarizes this work.

2. OVERVIEW OF SCALATRACE

Our work utilizes the publicly available ScalaTrace infrastructure [Noeth et al. 2009]. ScalaTrace is an MPI trace-gathering framework that generates near constant-size communication traces for a parallel application regardless of the number of nodes while preserving structural information and temporal ordering. ScalaTrace utilizes the MPI profiling layer (PMPI) to intercept MPI calls of HPC programs. Extended regular section descriptors (RSDs) are used to record the parameters and information of a single MPI event nested in a loop. Power-RSDs (PRSDs) recursively specify RSDs nested in multiple loops. For example, for the 4-point stencil code shown in Figure 1, *RSD1: <MPI_Irecv, (NORTH, WEST, EAST, SOUTH)>* and *RSD2: <MPI_Isend, (NORTH, WEST, EAST, SOUTH)>* denote the alternating send/receive calls to/from the 4 neighbors, and *PRSD1: < 1000, RSD1, RSD2, MPI_Waitall>* denotes the a loop with 1000 iterations. In the loop's body, RSD1, RSD2, and a following `MPI_Waitall` are called sequentially. During application execution, ScalaTrace performs intra-node compression, which captures the loop structure on-the-fly and represents MPI events in such a compressed manner. Local traces are combined into a single global trace upon application completion, i.e., within the PMPI interposition wrapper for `MPI_Finalize`. The key approaches to achieve near-constant inter-node compression are the location-independent encoding and communication group encoding schemes detailed in the following.

- *Location-independent encoding:* Communication end-points in SPMD programs differ from one node to another. By encoding endpoints *relative* to the index of an MPI task on a node, a location independent denotation is created that describes the behavior of large node sets. In a stencil/mesh

```
neighbor[] = {NORTH, WEST, EAST, SOUTH};
for(i=0; i<1000; i++) {
    for(j=0; j<4; j++) {
        MPI_Irecv(neighbor[j]);
        MPI_Isend(neighbor[j]);
    }
    MPI_Waitall();
}
```

Fig. 1. Sample Stencil Code for RSD and PRSD Generation

topology, only few of such distinct sets/groups tend to exist. Location-independent encoding not only opens up opportunities for inter-node compression to unify endpoints across different computational nodes but also enables extrapolation.

- *Communication group encoding:* Similarity in communication patterns is recognized to succinctly denote sets/groups of nodes with common behavior. In a topological space, a communication group refers to a subset of nodes that have identical communication patterns. With this encoding scheme, a communication group is represented as a *rank list*. Using the EBNF meta-syntax, a *rank list* is represented as $< dimension\ start_rank\ iteration_length\ stride\ \{iteration_length\ stride\} >$, where *dimension* is the dimension of the group, *start_rank* is the rank of the starting node, and the *iteration_length stride* pair is the iteration and stride of the corresponding dimension. As an example, consider the row-major grid topology in Figure 2. The shaded nodes form a communication group. This group is represented as *ranklist* <*2 6 3 5 3 1*>, where the tuple indicates that this communication group is a 2-dimensional area starting at node 6 with 3 iterations of stride 5 in the y-dimension and 3 iterations of stride 1 in the x-dimension, respectively. Since this encoding scheme takes node placement into account, it naturally reflects the spatial characteristics of a communication group.

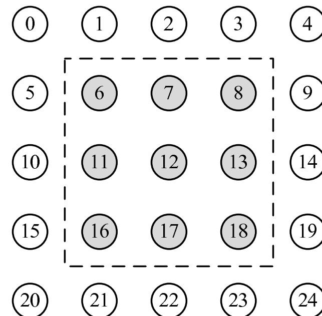


Fig. 2. Ranklist Representation for Communication Group

We exploit these representations as a foundation for extrapolating communication topology.

Besides communication tracing, ScalaTrace also preserves the timing information of a parallel application in a scalable way [Ratn et al. 2008]. Along with the intra-node and inter-node compression processes, “delta” times representing the computation between communication events are recorded and compressed. For the purpose of scalability, delta times of a single MPI function call across multiple loop iterations are not recorded one by one. Instead, histograms with a fixed number of bins for delta times are dynamically constructed to provide a statistical view. Delta times are distinguished by not only the call context of recorded events, but also by their path sequence, which addresses significant variation of delta times caused by path differences, *e.g.*, within entry/exit paths of a loop.

Finally, SCALAREPLAY is a replay engine operating on the application traces generated by SCALATRACE. It interprets the compressed application trace on-the-fly and issues MPI communication calls accordingly. During replay, all MPI calls are triggered over the same number of nodes with their original parameters (e.g., message payload size) but a randomly generated message content. This ensures comparable bandwidth requirements on communication interconnects. SCALAREPLAY emulates computation events in the original application by sleeping so that the communication contention characteristics are maintained during replay. In general, the replay engine can be utilized for rapid prototyping and tuning, as well as to assess communication needs of future platforms for large-scale procurements in conjunction with system simulators (DIMEMAS/SST) [Labarta et al. 1997; Snavely et al. 2002; Rodrigues et al. 2006]. In this work, we use SCALAREPLAY to verify the correctness of extrapolation results, which will be discussed later in this paper.

3. COMMUNICATION EXTRAPOLATION

This work focuses on the extrapolation of communication traces and execution times. The respective design is subsequently implemented in a novel tool, SCALAEXTRAP. The challenge of communication trace extrapolation is to determine how the communication parameters change with node and problem scaling. The main idea is to identify the relationship between communication parameters and the characteristics of the communication topology, *i.e.*, typically the sizes of each dimension. As a simple example, in Figure 2, assume *node 0* communicates with *node 4*, *i.e.*, a node at distance of 4. If we can identify that the topological communication space is a grid consisting of 25 nodes with 5 nodes per row, we know that *node 0* actually communicates with the upper-right node. Therefore, when there are $1024 = 32\times32$ nodes, we can safely infer that *node 0* communicates with *node 31*, which is still the upper-right node.

Characterizing a communication pattern from one or more traces is non-trivial nonetheless. Without the knowledge of a given node assignment scheme and topology, identifying the communication pattern from the communication graph provided by a trace file is equivalent to solving the graph isomorphism problem, which is known to be NP hard [Xu and Subhlok 2008]. Therefore, instead of attempting to find a universal solution, we constrain our work to applications where

(1) nodes execute the same program on different data, i.e., the application follow the SPMD paradigm;
(2) nodes are numbered in a row-major fashion; and
(3) communication is performed in stencil/mesh point-to-point manner or via collectives involving all MPI tasks.

In essence, our communication trace extrapolation algorithm first identifies the nodes at the "corner" of a topological space. It then calculates the sizes of each dimension of the topological space accordingly.

Upon acquiring the topology data, we can perform extrapolation. The extrapolation of a communication trace consists of two tasks. First, we need to match the records corresponding to the same MPI call in the source code across the traces of different node sizes. We will discuss the difficulties involved in this step and our solutions in the following sections. Second, for each MPI call in the source code, we need to determine which MPI processes execute this call and what are the values of the parameters when the application is running at the target scale. For the second task, we represent the rank list and the communication parameters, *e.g.*, the destination rank of `MPI_Send`, as a function of the known topology data and their undetermined coefficients. In order to calculate these coefficients, we correlate multiple traces and construct a set of linear equations. Finally, we employ Gaussian Elimination to solve the set of equations. With the fixed coefficients, we can extrapolate the value of the desired communication parameter by simply substituting the topology data with their values at the desired problem size.

The second aspect of this work concerns the extrapolation of program execution time. In the input trace files, computation time and communication time between (and optionally during) MPI com-

munication events are preserved statistically with histograms. When analyzing the corresponding delta time, scaling trends can be identified across different number of nodes. Therefore, statistical curve fitting methods are utilized to model an evolving trend and extrapolate the execution time to a desired target size. In order to eliminate outliers, we further introduce several confidence coefficients to statistically determine the best extrapolated value under such constraints.

3.1. Topology Identification

Topology identification is the basis of communication trace extrapolation. In order to identify a topology, it is important to find the nodes at the corner or on the boundary of a topological space, which we call *critical nodes*. We devised a three-step approach to identify the communication topology.

(1) We create an adjacency list of communication endpoints for each node and group nodes according to their adjacency lists.
(2) We identify critical nodes by analyzing the adjacency lists.
(3) We calculate the sizes of each dimension (x, y, and z) of the communication topology.

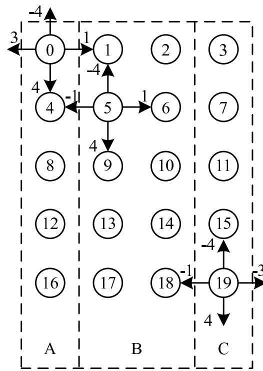


Fig. 3. Topology Detection

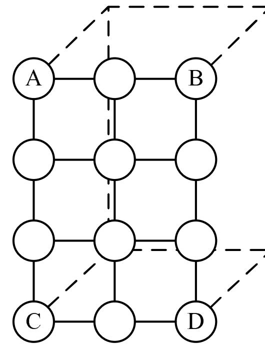


Fig. 4. Boundary Size Calculation

First, our algorithm traverses the input trace to construct communication adjacency lists for each node. According to the relative positions (encodings) of all the communication endpoints of each node, nodes with same endpoint patterns are placed into the same group. Figure 3 illustrates an example of a 2D mesh topology. In this example, nodes on the boundaries communicate with nodes at the opposite side in a wrap-around manner while the internal nodes communicate with their immediate neighbors. Note that wrapping around in the vertical direction does not lead to different endpoint encoding. Therefore, the nodes are divided in to three groups (A, B, and C) with group sizes 5, 10, and 5, respectively.

Next, we analyze the adjacency list of each node to identify the critical nodes. Exploiting the row major constraint, we scan all nodes sequentially to identify loop structures with respect to communication adjacency list patterns. The underlying rationale is that critical nodes define a topology. Between corresponding critical nodes, communication patterns emerge repeatedly. According to the length of a loop structure, the sizes of the groups consist of critical nodes, *i.e.*, *critical groups*, are calculated as

$$\textit{critical group size} = \frac{n}{\textit{length of loop}},$$

where n denotes the number of nodes engaged in MPI communication. For example, in Figure 3, each row has the same group distribution (A B B C) and is thus identified as a single iteration of the

loop structure. Since the length of such a loop iteration is 4, the size of the *critical groups* (group A and C) is *20/4 = 5*. Having obtained the size of the critical groups, we then associate critical nodes with groups by matching sizes of critical groups.

Finally, we calculate the sizes of each dimension. Again exploiting the row-major constraint, in a d-dimensional topological space, the number of nodes at the *d-th* dimension is the total number of nodes. The number of nodes at the *i-th* $(i < d)$ dimension, n_i, is the inclusive range of numbers of nodes between *node 0* (*1st* critical node) and the 2^i*-th* critical node. Once we have determined the number of nodes at each dimension, the boundary size of the *i-th* dimension, s_i, is calculated as

$$s_i = \frac{n_i}{n_{i-1}}$$

For example, in the 3D topology of Figure 4, the number of nodes in the *1st* dimension, $n_1=3$, is the number of nodes between A and B inclusively, the number of nodes in the second dimension, $n_2=12$, is the number of nodes between A and D, and the number of nodes in the third dimension n_3 is the total number of nodes. Hence, we have

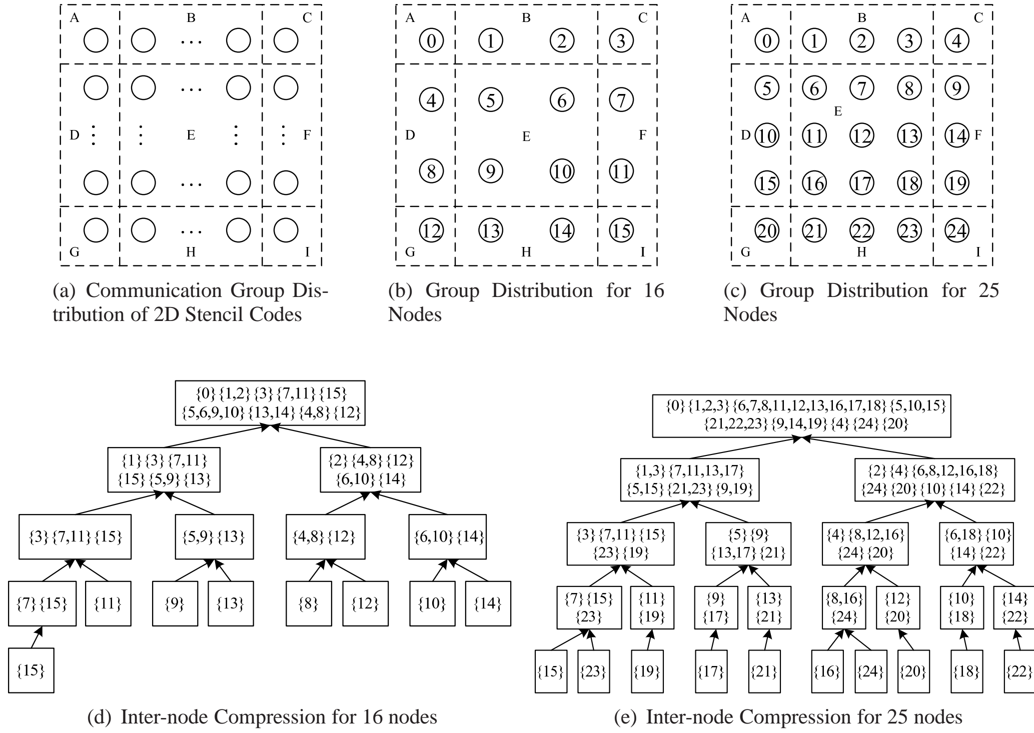
$$\begin{cases} x = s_1 = n_1/n_0 = 3 \\ y = s_2 = n_2/n_1 = 4 \\ z = s_3 = n_3/n_2 \end{cases}$$

3.2. Matching MPI Events for Extrapolation

The extrapolation of a trace is performed one-by-one for each recorded MPI event of the trace. An MPI event is emitted per execution of an MPI function in the source code by the actual values of the input parameters. Therefore, the extrapolation of an MPI event is actually the process of inferring the execution of an MPI function at the target scale from its executions at smaller scales, which are represented as RSDs in the input traces. (In addition, due to the SPMD nature of parallel applications, the extrapolation also involves the prediction of the participants of an MPI event, *i.e.,* the callers of an MPI function in the source code, which will be discussed in Section 3.3.) Therefore, being able to match the RSDs corresponding to calls to the same MPI function originating from source code across traces of different node sizes is the prerequisite of extrapolation.

Due to its structure-preserving representation, SCALATRACE traces are often similar to the source code. In a trace, the queue of RSDs represents the temporal ordering of the MPI events, which in turn reflects the locations of the corresponding MPI function invocations in the source code. Therefore, in most cases, traces of different node sizes are inherently aligned. However, nodes are sometimes partitioned due to differences in their communication patterns and may thus form different communication groups. For example, Figure 5(a) shows the distribution of the communication groups of 2D stencil codes such as Sweep3D. Since the communication behavior is different across groups, SCALATRACE cannot merge the per-node traces but appends them sequentially. Because the inter-node compression are performed with a radix tree and the order of disjoint subsequences of MPI events are not maintained during compression [Noeth et al. 2007], the relative positions of RSDs originating from different communication groups are not necessarily the same in traces of different node sizes. For example, in the final 16-node trace the third group is Group C (both in Figure 5(b) and in the root node of Figure 5(d)) while it is Group E in the 25-node trace (in the root node of Figure 5(e)) . This illustrates how the order of different communication groups are determined along with the radix tree style inter-node reduction (cf. Figures 5(d)+(e)). Clearly, extrapolating by relating RSDs of different communication groups is meaningless.

We utilize the dependence graph to reorder the trace. The dependence graph is a data structure used by SCALATRACE to keep track of disjoint RSD subsequences during the inter-node reduction [Noeth et al. 2009]. If two per-node traces are partially different, a branching point and a merging point will be inserted before the first and after the last non-matching RSDs. We designed a recursive algorithm that traverses the dependence graph in a depth-first manner and topologically sorts each branch in the rank order (Algorithm 1). Our reordering algorithm guarantees that the RSD subsequences corresponding to different communication groups are always organized in ascending



(a) Communication Group Distribution of 2D Stencil Codes

(b) Group Distribution for 16 Nodes

(c) Group Distribution for 25 Nodes

(d) Inter-node Compression for 16 nodes

(e) Inter-node Compression for 25 nodes

Fig. 5. Inter-node Compression and the Positions of Communication Groups for 2D Stencil Codes

order of the rank for the leading nodes of groups. With such an algorithm, we are able to align the communication groups in traces of different node sizes. The extrapolation is subsequently becomes possible.

3.3. Extrapolation of MPI Events

The extrapolation of an MPI event consists of the extrapolation of both communication groups and communication parameters to indicate who communicates and how they communicate. The extrapolation algorithm is based on the observation that, in regular SPMD stencil/mesh codes, *strong scaling* (increasing the number of nodes under a constant input size) linearly increases/decreases the value of communication parameters and the topological sizes. Given several data points, a fitting curve can be constructed to extrapolate the growth rate of the communication parameters and the topology information (the sizes of each dimension) of the communication groups.

Specifically, in an n-dimensional Cartesian space, the coordinates of node X and Y are $(X_1, X_2, ..., X_n)$ and $(Y_1, Y_2, ..., Y_n)$, where X_i and $Y_i \in [0, S_i - 1]$ and S_i is the size of the *i-th* dimension of the topological space $(1 \le i \le n)$. Assuming the locations of node X and Y differ only in the *i-th* dimension, the distance between X and Y in the *i-th* dimension is $d_i = X_i - Y_i$. With the assumption of linear correlation between topology size and communication parameters, $d_i = X_i - Y_i = a_i \times S_i + b_i$, where a_i and b_i are two constants. Furthermore, with the row-major node placement assumption, the rank of an arbitrary node $A(A_1, A_2, ..., A_n)$ is

$$Rank_A = \sum_{i=1}^{n} A_i \prod_{j=1}^{i-1} S_j.$$

Algorithm 1 Aligning the Communication Groups

Require: T_{in}: input trace
Ensure: T_{out}: output trace in which branches (RSD subsequences for communication groups) are ordered by rank

```
1: procedure REORDER_TRACE(T_in)
2:     for iter ← T_in.head, T_in.tail do
3:         if iter is a merging RSD node then
4:             merging_node ← iter
5:             find branching_node: merging_node's matching branching RSD node
6:             REORDER(merging_node, branching_node) ▷ reorder the branches between merging_node and branching_node by rank
7:         end if
8:     end for
9: end procedure

10: procedure REORDER(merging_node, branching_node)
11:     for each branch between merging_node and branching_node do
12:         traverse branch in depth-first order
13:         if m: a merging RSD node is found in branch then
14:             find b: m's matching branching RSD node
15:             REORDER(m, b)                    ▷ recursively reorder the branches
16:         end if
17:     end for
18:     sort the branches between merging_node and branching_node by rank
19:     reorder the branches
20: end procedure
```

Therefore, $d_i{}'$, the rank distance between X and Y, is

$$d_i{}' = (X_i - Y_i) \times \prod_{j=1}^{i-1} S_j = (a_i \times S_i + b_i) \times \prod_{j=1}^{i-1} S_j$$

In general, for two arbitrarily selected nodes M and N, their rank distance d' is the sum of their rank distances in each dimension,

$$\begin{aligned} d' &= d_0{}' + d_1{}' + \ldots + d_n{}' \\ &= \sum_{i=1}^{n} (N_i - M_i) \prod_{j=1}^{i-1} S_j = \sum_{i=1}^{n} (a_i \times S_i + b_i) \prod_{j=1}^{i-1} S_j \\ &= a_n \prod_{j=1}^{n} S_j + \sum_{i=1}^{n-1} (a_i + b_{i+1}) \prod_{j=1}^{i} S_j + b_1 = \sum_{i=0}^{n} c_i \prod_{j=1}^{i} S_j, \end{aligned}$$

where $c_n = a_n$, $c_0 = b_1$, and $c_i = a_i + b_{i+1} (1 \leq i \leq n-1)$.

In order to extrapolate the rank of a communication endpoint (src/dest), which is defined by the rank distance between nodes, we need to identify how the topology information is related to the communication parameter. We construct a set of linear equations to solve c_i (1≤i≤n-1). In general, for an n-dimensional topology, $n+1$ input traces are needed to solve $n+1$ coefficients. We employ Gaussian Elimination to solve the equations. Once the values of $c_i (1 \leq i \leq n-1)$ are determined, a fitting curve for the given parameter is established. In order to extrapolate the same parameter for a larger execution, we utilize the known coefficients and specify the topology information at the target task size. The desired value is then calculated accordingly.

As an example, in a 2D space, the bottom-right node in Figure 6 communicates with its *EAST* neighbor in a wrap-around manner. In order to extrapolate the rank of the communication end-

point, three input traces with dimensions 4×4, 5×5, and 6×6 are used to construct the set of linear equations shown in Figure 7, and $c_2 = 1$, $c_1 = -1$, and $c_0 = 1$ are obtained as the values of the coefficients. To extrapolate a 10×10 mesh, we re-construct the equation with coefficients and topology information assigned. Subsequently, the target value V is calculated as $V = c_2 \times 10 \times 10 + c_1 \times 10 + c_0 = 91$.

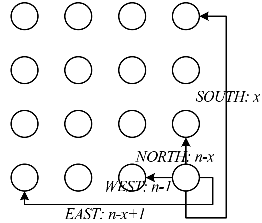


Fig. 6. Generic Representation of Communication Endpoints

$$\begin{cases} c_2 \times 4 \times 4 + c_1 \times 4 + c_0 = 13 \\ c_2 \times 5 \times 5 + c_1 \times 5 + c_0 = 21 \\ c_2 \times 6 \times 6 + c_1 \times 6 + c_0 = 31 \end{cases}$$

Fig. 7. Set of Equations for Communication Endpoint Extrapolation

Besides the communication parameters, communication groups are also extrapolated. The topological space of an application can be partitioned into several communication groups according to the communication endpoint pattern of each node. Under *strong scaling*, partitions tend to retain their position within the topological space but change their sizes for each dimension accordingly. For example, Figure 8 shows the distribution of 9 communication groups of a 2D stencil code. Despite the changing problem size, groups *A*, *C*, *G*, and *I* always represent corner nodes, groups *B*, *D*, *F*, and *H* are always the boundaries, and group *E* contains the remaining (interior) nodes.

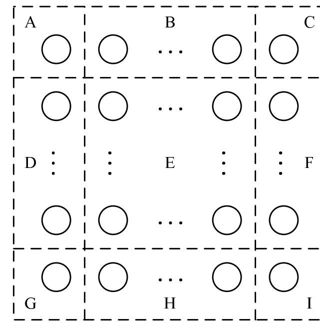
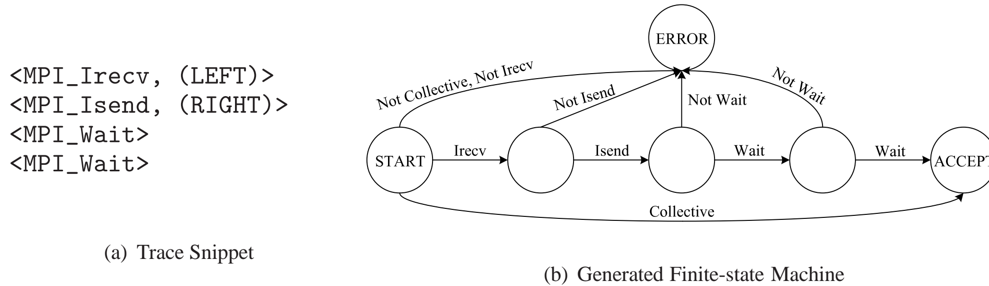


Fig. 8. Distribution of Communication Groups of a 2D Stencil Code

This opens up the opportunity to extrapolate communication groups of the same application at arbitrary size. In order to extrapolate, we represent communication groups as *rank lists*, which effectively specifies the starting node and the dimension sizes of a group. Since the dimension sizes are defined by the distances between nodes (vertices), we again utilize a set of linear equations to establish the relation between the topology information of communication groups and the task sizes. Extrapolation is performed for the *start_rank*, *iteration_length*, and *stride* fields of the rank list. The output rank list reflects the communication group at the target size. For example, for the topology shown in Figure 8, when the total number of nodes is 16, the rank list of group *E*, as defined in Section 2, is *<2 5 2 4 2 1>*, *i.e.,* a 2D space starting from *node 5* with x- and y-dimensions of size 2. Similarly, the rank lists of group *E* at sizes 25 and 36 are *<2 6 3 5 3 1>* and *<2 7 4 6 4 1>*,



(a) Trace Snippet

(b) Generated Finite-state Machine

Fig. 9. A Simple Trace Snippet and the Generated Finite-state Machine

respectively. We can thus construct the set of linear equations for each field in the rank list to derive a generic representation of the rank list as:

$$< 2 \quad x+1 \quad x-2 \quad x \quad x-2 \quad 1 >.$$

Subsequently, assuming that we want to extrapolate for size 10×10, let x be 10, which yields the output rank list $<2\ 11\ 8\ 10\ 8\ 1>$ that precisely matches the *rank list* representation of communication group E at this problem size.

By combining the extrapolation of both communication groups and communication parameters, we are capable of extrapolating the communication trace for a given application at arbitrary topological sizes.

3.4. Lossy Extrapolation

As was discussed in Section 3.2, aligning the matching RSDs across traces of different node sizes is critical for extrapolation. Despite being recognized as one of the existing tracing libraries that provides the best compression, there still exist some applications that SCALATRACE cannot obtain constant-size compression for. In fact, for applications that exhibit new communication patterns only at or beyond a certain node size, lossless yet constant-size compression is hardly possible. To extrapolate traces of such applications, we designed a lossy approach. We attempt to capture and extrapolate the dominating communication pattern by optionally dropping events at three different levels: (1) within an MPI process, (2) among MPI processes of an execution, and (3) across traces of different node sizes.

The intra-node level event filtering is performed against the per-node queue of MPI events. We observe that a number of application traces contain a subsequence of events that embody the dominating communication pattern and comprise a large portion of the trace by repeating multiple times. Based on this observation, a user-provided trace snippet is utilized as the reference to drop events. Typically, such a trace snippet is a sequence of RSDs consisting of tens of MPI events, with the event type (Send, Recv, etc.) and the values of the key parameters of each event. Based on the trace snippet, we automatically generate a Finite-State Machine (FSM) to process the input stream of MPI events. At the beginning, the FSM is initialized to the START state. If the input event is a collective, the FSM directly enters the ACCEPT state. This indicates that all collectives are directly accepted while the FSM is not in the middle of accepting a sequence. If the input is neither a collective nor the first event to be accepted, the FSM enters the ERROR state and the input event is dropped. Once the FSM leaves the START state, it only accepts the next event expected in a sequence. If an unexpected event arrives, the FSM enters the ERROR state and all the pending events are dropped, including the current input if it is not a collective. Finally, if the FSM arrives at the ACCEPT state, the pending events are accepted. These events will not be affected by future ERROR states. Figure 9 shows a simple trace snippet and the FSM generated.

Beyond the intra-node level event filtering, if necessary, we also drop events during the inter-node compression and when aligning the traces of different node sizes for extrapolation. We designed a Longest Common Subsequence (LCS) based approach for the event filtering at these two levels. The LCS problem is to find the longest subsequence that is common to two input sequences, where a subsequence need not be consecutive in either of the original sequences. If a trace is considered as a sequence of MPI events, the LCS of two traces reflects the MPI events that nodes participate in for both traces. We adapted a well-known dynamic programming based LCS algorithm for trace comparison [Bergroth et al. 2000]. In a SCALATRACE trace, the loop structure is preserved and explicitly indicated. As a building block, loop structures should be evaluated in their entirety with the number of MPI events in the loop representing the weight. Therefore, we first enhanced the LCS algorithm to take into account the weight when evaluating how the length of the LCS will be affected by retaining or removing a loop structure. Second, since loops are often nested in the source code and in the trace, we further modified the LCS algorithm so that it can execute in two modes in a recursive manner. In the first pass, this algorithm only calculates the LCS but does not modify the trace. This is required because modifying the inner loop may affect the evaluation of the outer loop. Once the LCS is determined, this algorithm is applied again in the second mode such that any uncommon events are removed.

With these event filtering techniques above, we are able to extrapolate complicated and irregular applications depending on nodes counts such as the NPB MG kernel. We have developed a replay engine that infers receives from sends to replay a histogram-based communication trace [Wu et al. 2011]. In case the event filtering makes the trace incorrect, *e.g.,* when sends and receives mismatch, this algorithm can be adapted to execute on a single machine to correct the trace. Our insight here is that instead of dropping the minor communication events not matched by any other MPI processes due to event filtering, preserving them by manually generating the matching events with probabilistic approaches may be a better solution [Wu et al. 2011].

3.5. Extrapolation of Timing Information

Besides the communication traces, we also extrapolate the timing information of the application. SCALATRACE preserves the "delta" time for each communication event and for the computation between two communication events. For a single MPI function call across multiple loop iterations, *i.e.*, for a RSD, the delta times are recorded in multi-bin histograms. These histograms contain the overall average, minimum, and maximum delta time, the distribution of the delta execution times represented as histogram bins, and the average, minimum, and maximum delta time for each histogram bin. To extrapolate timing information, we utilize curve fitting to capture the variations in trends of the delta times with respect to the number of nodes, *i.e.,* $t=f(n)$, where t is the delta execution time and n is the total number of nodes. Hence, the target delta time t_e is calculated as $t_e = f(n_e)$, where n_e is the total number of nodes at a given problem size. While we can extrapolate only the aggregated average delta time per RSD, to restrain the statistics of delta time, extrapolation is performed for each field of a histogram. Currently, we implemented four statistical models based on curve fitting for each extrapolation. We use a deviation-based metric to determine the best of these models to fit to a given curve.

(1) Constant: This method captures constant time, *i.e.*, $t=f(n)=c$. Before calculating the constant time, the input time t_o with the largest absolute value of deviation is excluded from the input times to mitigate the influence of outliers (which can be caused by either unstable system state or an empty bin). Subsequently, the average value of the remaining input times reflects the constant time c, and d_1=*std. dev./average* is used to evaluate this fitting curve among the remaining values.
(2) Linear: This method captures linearly increasing/decreasing trends, *i.e.*, $t=f(n)=an+b$. We use the least-squares method to fit the curve. In order to avoid mis-classifications, such as a constant time relationship as a linear relationship with a near-zero slope, we define a threshold slope

s_{min}=0.2 such that $\forall a < s_{min}$ t=$f(n)$=b. For curve evaluation, $d_2 = \sqrt{residual}/average$ is used, where *average* refers to the average value of the estimated running times.

(3) Inverse Proportional: This method captures inverse-proportional trends, *i.e.*, t=$f(n)$=k/n. We observe this trend in the NAS Parallel Benchmark IS, where `MPI_Alltoallv` dynamically re-balances the per-node workloads even though the collective workload over all nodes is constant. Let t_i be the input times, n_i be the corresponding number of nodes, and $k_i = t_i \times n_i$. We extrapolate the constant k as the average value of k_i. Again, we exclude the outlier k_o, which has the largest absolute value within the deviation. To evaluate this fitting curve, we calculate the standard deviation of k_i and then divide by the average value of k_i, *i.e.*, d_3=*std. dev./average* is used for comparison.

(4) Inverse Proportional + Constant: This method captures the execution time consisting of an inverse proportional phase and a constant phase, *i.e.*, t=$f(n)$=k/n+c. Instead of directly extrapolating t, we utilize the least-squares method to extrapolate $t' = tn = cn + k$ and use $d_4 = \sqrt{residual}/average$ for the curve evaluation. With an extrapolated c and k, t is subsequently calculated as $t = t'/n = k/n + c$.

Having obtained the deviations for each curve-fitting process, we compare the values to determine the curve that best fits. For a closer approximation, we define a threshold value $d_t = 0.05$, such that if and only if $d_{min} + d_t < d_i$ holds for all d_i other than d_{min} will the corresponding candidate curve be selected as the fitting curve. Otherwise, the extrapolation for the current field is postponed until we have processed all the fields in the same histogram. Since every field in the histogram should have the same variation trend, we finalize the pending extrapolation according to the decisions of the remaining fields.

4. EXPERIMENTAL FRAMEWORK

Our extrapolation methodology for communication traces was implemented as the SCALAEXTRAP tool that generates a synthetic trace for a freely selected number of nodes. The extrapolation is based on traces obtained from application instrumentation with SCALATRACE on a cluster. For both base traces generation and results verification, we use a subset of JUGENE, an IBM Blue Gene/P with 73,728 compute nodes and 294,912 cores, 2 GB memory per node, and the 3D torus and global tree interconnection networks.

The extrapolation process is run on a single workstation and requires only several seconds, irrespective of the target number of nodes for extrapolation. This low overhead is due to the linear time complexity of our algorithm with respect to the total number of MPI function calls in an application. Results from extrapolation are subsequently compared to traces and runtimes of an application at the same scale, where runtimes for extrapolated traces are obtained via SCALAREPLAY (see Section 2).

We conducted extrapolation experiments with the NAS Parallel Benchmark (NPB) suite (version 3.3 for MPI) [Bailey et al. 1991] and Sweep3D [Wasserman et al. 2000]. These benchmarks have either a stencil/mesh communication pattern or collective communication, both of which are applicable to our extrapolation algorithm. Among these benchmarks, IS originally exhibited imperfect compression resulting in non-scalable trace sizes due to its dynamic load re-balancing via workload exchange through the `MPI_Alltoallv` communication collective. In order to utilize our extrapolation techniques, we enhanced SCALATRACE such that minor differences in `MPI_Alltoallv` parameters caused by load re-balancing are eliminated. The communication pattern of CG is another example of a complicated dynamic pattern. In CG, nodes are logically organized in a 2D array. Each node communicates with the nodes in the same row with a power-of-two distance and with the node diagonally symmetric to itself, as indicated in Figure 10. We support such more complicated patterns by allowing programmers to provide plugin functions for compression and extrapolation on a per-parameter basis. The communication trace extrapolation for CG is facilitated by specifying the communication pattern (i.e., the communication end point described by a function) as a plugin. With this plugin, the extrapolation of timing information does not require any extra information.

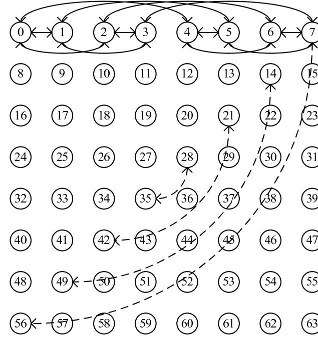


Fig. 10. CG Communication Topology

We report the experimental results for both strong scaling and weak scaling. For the strong scaling experiments, we mostly used class D and E input sizes for the NPB codes. For the weak scaling experiments, we enhanced the input generator to provide weak scaling inputs for selected NPB codes.

5. EXPERIMENTAL RESULTS

Experiments were conducted with respect to two aspects, namely the correctness of communication traces and the accuracy of timing information, both for extrapolations under strong scaling, *i.e.*, when varying the number of nodes. Notice that strong scaling is actually a *harder* problem under extrapolation as it tends to affect communication parameters such as message volume size. In contrast, weak scaling (increasing the number of nodes and problem sizes at the same rate) is easier as it tends to preserve message volumes sizes irrespective of the number of nodes.

5.1. Correctness of Communication Trace Extrapolation

We first evaluated our communication trace extrapolation algorithm with microbenchmarks and the NPB BT, EP, FT, CG, LU, and IS codes. We assessed the ability to retain communication semantics across the extrapolation process for these benchmarks at the target scale. The microbenchmarks perform regular stencil-style/torus-style communication in topological spaces from 1D to 3D. The NPB programs exercise both collective and point-to-point communication patterns. We verified the extrapolation results in multiple ways.

(1) The extrapolated trace file T_{e_0} was compared with the trace file obtained from an actual execution at the same scale T_{target} on a per-event basis (Exp1 in Figure 11).
(2) The extrapolated trace T_{e_0} was replayed such that aggregate statistical metrics about communication events could be compared to those of a corresponding original application run at the same problem size and node size (Exp2 in Figure 11).
(3) After extrapolation, traces T_{e_1}, T_{e_2}, ..., T_{e_i} were collected in a sequence of replays to obtain a fixed point in the trace representation (Exp3 in Figure 11).

First, the per-event analysis of trace files showed that extrapolated MPI parameters and communication groups perfectly matched those of the application trace for all benchmarks except one (Exp1 in Figure 11). In BT, the message volume of non-blocking point-to-point sends and receives *approximates* an inverse-proportional relationship with respect to the number of nodes. However, it diverges slightly from an inverse-proportional approximation for extrapolating the message volume due to integer division (discarding the remainder) inherent to the source code. This inaccuracy is later amplified in the extrapolation process and results in message volumes that are about 13% smaller than the actual ones at a given scaling factor in the worst case. As imprecisions remain localized to certain point-to-point messages, this effect is shown to be contained in that resulting

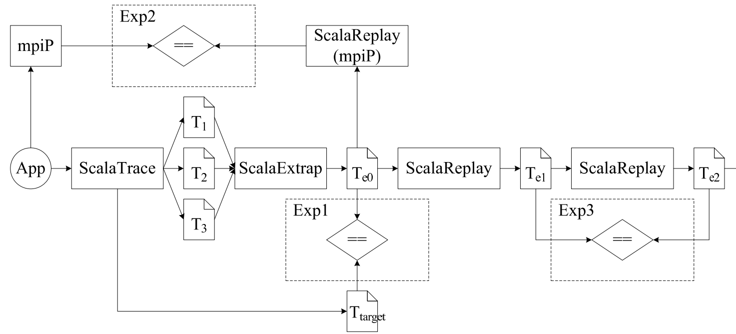


Fig. 11. Correctness of Trace Extrapolation and Replay

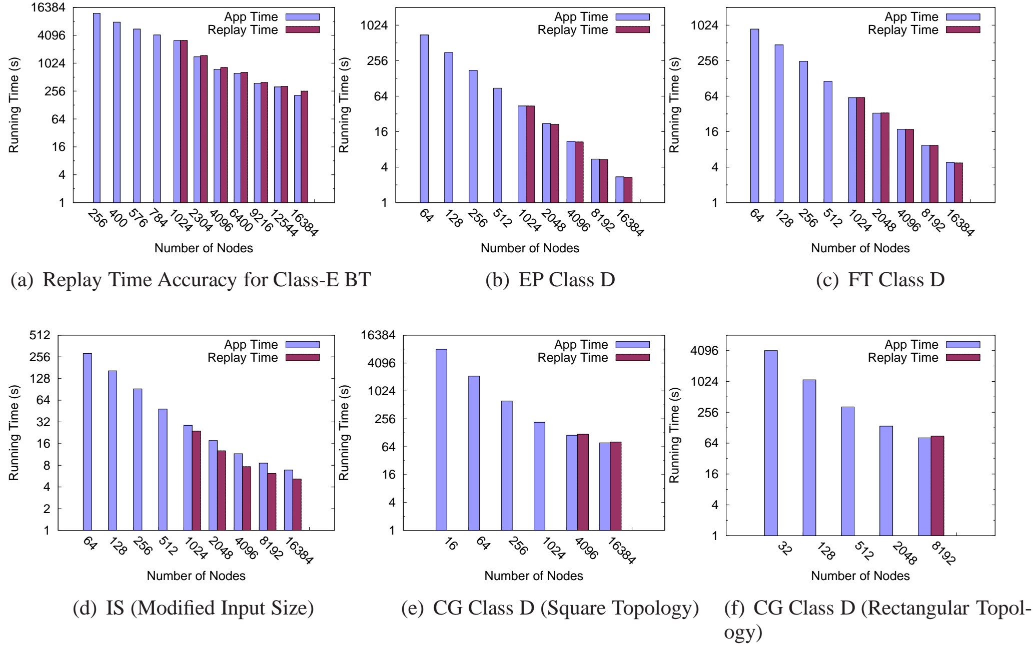
timings are deemed accurate within the considered tolerance range for extrapolation experiments (see timing results below). Such imprecisions have no side-effect on semantic correctness (causal order) of trace events whatsoever. Overall, the results of static trace analysis show that our synthetically generated extrapolation trace is equivalent to the trace obtained from actual execution of the same application at the same scaling level.

Second, we replayed the extrapolated trace T_{e_0} to assess if the MPI communication events are fully captured (see Exp2 in Figure 11). For this experiment, SCALAREPLAY is linked with mpiP [Vetter and McCracken 2001], which yields frequency information of each MPI call distinguished by call site (using dynamic stackwalks). During replay, all MPI function calls recorded in the synthetically generated extrapolation trace were executed with the same number of nodes and their original payload size. For comparison, we instrumented the original application with mpiP and executed it at extrapolated sizes (problem and node sizes). We compared the *Aggregate Sent Message Size* reported by mpiP between the original application and the replayed extrapolated trace. Results show that the total send volumes of these experiments are identical, except for `MPI_Isend` in BT as discussed above. We also compared the total number of MPI calls recorded in the mpiP output files. The results allowed us to verify that the number of communication events in the actual and extrapolated traces match, *i.e.*, the correctness of communication trace extrapolation is preserved.

Third, we evaluated the correctness of SCALAREPLAY by replaying the generated trace file in sequence until a fixed point is reached (see Exp3 in Figure 11). The fixed point approach is a well established mathematical proof method that establishes conversion, in this case of the trace data. In this experiment, instead of instrumenting SCALAREPLAY with mpiP, we interposed MPI calls through SCALATRACE again. As SCALAREPLAY issues MPI function calls, SCALATRACE captures these communication events and generates a trace file for it, just as would be done for any other ordinary MPI application. We start by replaying the extrapolated trace file T_{e_0} and obtain a new trace T_{e_1}. This trace differs from T_{e_0} in that call sites of the original program have been replaced by call sites from SCALAREPLAY. This affects not only stackwalk signatures but also the structure of trace files due to the recursive approach of replaying trace files in place over their internal (PRSD) structure without decompressing it. We then replay trace T_{e_1} to obtain another trace T_{e_2} and so on for T_{e_i}. We then compare pairs of trace files $T_{e_i}, T_{e_{i+1}}$. If two such traces match, a fixed point has been reached. In these experiments, we verified that pairs of trace files, baring syntactical differences, are semantically equivalent to each other. In other words, SCALAREPLAY neither adds nor drops any communication events during replay, *i.e.*, by obtaining a fixed point it was shown that all MPI communication calls are preserved during replay.

5.2. Accuracy of Extrapolated Timings: Timed Replay

We further analyzed the timing information of the extrapolated traces. We report the accuracy of the extrapolated timings for both strong scaling and weak scaling.



(a) Replay Time Accuracy for Class-E BT (b) EP Class D (c) FT Class D

(d) IS (Modified Input Size) (e) CG Class D (Square Topology) (f) CG Class D (Rectangular Topology)

Fig. 12. Replay Time Accuracy for Strong Scaling Benchmarks

Strong Scaling: For this set experiments, we used the NPB BT, EP, FT, CG, and IS codes with a total number of nodes of up to 16,384. For CG, EP, and FT, we used class D input sizes. For BT, class E was used so that a sufficient workload is guaranteed at 16,384 nodes. For IS, we modified the input size to adapt it for 16,384 nodes (the original NPB3.3-MPI provides only class D problem size and supports a maximum of 1024 nodes). These problem sizes and node sizes were decided based on the memory constraints (for some benchmarks, memory constraints compel us to generate the base traces already at large scales, which in turn leaves fewer target sizes for evaluation) and the availability of computational resources to assess the effects and limitations of our timing extrapolation approach.

In this set of experiments, we first generated 4 trace files for each benchmark as the extrapolation basis. From these base traces, an extrapolated trace was constructed next using ScalaExtrap, including extrapolated delta time histograms. We then assess the timing accuracy by replaying the extrapolated traces. During replay, ScalaReplay parses the timing histograms of the computation periods in the trace files. It simulates computation by sleeping to delay the next communication event by the proper amount of time. In this context, the effect of load imbalance is preserved by ScalaTrace. The timing histogram records not only *minimum, maximum, average and standard deviation* values, but also the *frequency* for each timing bin, and these statistics are also extrapolated by ScalaExtrap. During replay, the sleeping time is generated according to these statistics and the unbalanced timing behavior is thus reproduced. Communication is simply replayed with the same extrapolated end points and payload sizes but a random message payload. We do not impose any delays on communication as published results indicate better accuracy with just delays for computation only [Noeth et al. 2009], which we also confirmed. In this experiment, ScalaReplay is linked to neither ScalaTrace nor mpiP to avoid additional overhead caused by the instrumentation layer of these tools. Hence, the output of ScalaReplay in this experiment is the total time to replay a trace. For each extrapolated trace, we run the corresponding application at the same problem size and record its overall execution time for comparison.

Figure 12 depicts the extrapolation accuracy of BT, EP, FT, IS, and CG, respectively, for a varying number of nodes. We show the extrapolation results of CG in separate figures because they have different communication topologies and thus a different extrapolation basis. As shown in Figure 12, the timing extrapolation accuracy is generally higher than 90%, sometimes even higher than 98%, where accuracy is defined as

$$Accuracy = (1 - \frac{|Replay\ Time\ -\ App\ Time|}{App\ Time}) \times 100\%.$$

For BT, we observed slightly lower accuracy when the total number of nodes approaches 16,384. At such sizes the computational workload becomes so small that the influence of non-deterministic factors, such as system overheads or performance fluctuation of MPI collectives caused by different process arrival patterns [Faraj et al. 2007], become dominant. Compared to the other benchmarks, IS shows a constantly lower accuracy (66%-83%). Two reasons may explain this phenomenon: (a) Although IS dynamically rebalances the workload across all nodes, the execution time of the application's sorting algorithm on each process still takes a different amount of time. Hence, collective MPI calls take unpredictable time to synchronize as the arrival times of processes at collectives varies significantly due to load imbalance. Since the degree of imbalance is determined by randomly determined delta times from histograms, it is difficult to predict/extrapolate this behavior. (b) Source code analysis shows that the most computationally intensive code section in IS consists of two phases, namely (i) an inverse-proportional phase (runtime is inverse-proportional to the number of nodes), and (ii) a relatively short constant phase (runtime does not change significantly with node sizes). When the node size is small, the inverse-proportional phase almost solely determines the computation time. As a result, our algorithm fails to uncover a small constant factor that contributes to timing for larger node sizes. SCALAEXTRAP instead treats it as a pure inverse-proportional timing trend. Without the short constant factor in the timing curve, the extrapolated runtime drops slightly faster than the real runtime leading to a constantly shorter replay time. However, since we are able to capture the dominating inverse-proportional timing trend, we still obtained an acceptable timing prediction accuracy.

In large, minor inaccuracies during replay stem from imprecise curve fitting for the extrapolation of computation times. For the simulation of communication duration, SCALAREPLAY depends only on the communication parameters such as end points and payload sizes, which are shown to be correctly extrapolated in Section 5.1. Overall, the extrapolated timing information precisely reflects the runtime of the original application at the target problem size and node size.

Weak Scaling: Weak scaling refers to varying the problem size and the number of processes at the same rate so that the problem size per node stays consistent during scaling [Gustafson 1988]. Among the three factors we have to extrapolate, namely communication topologies, message sizes, and computation times, strong scaling and weak scaling generally do not affect the communication topology in different ways, *i.e.*, the communication patterns often evolve similarly for both strong and weak scaling. Therefore, the communication topology detection and extrapolation algorithms still apply to weak scaling codes. For the other two factors, compared to strong scaling codes, weak scaling codes may exhibit different runtime behavior. For example, due to a constant computational workload per node, the computation times often (but not always) follow a constant trend for weak scaling. In terms of the message sizes, the overall message volume exchanged among all the participating nodes—typically with `MPI_Alltoall` or `MPI_Alltoallv`—often increases linearly (or remain constant) under weak scaling when varying the total number of processes. Nonetheless, the curve fitting approach is still applicable, though different/additional curve fitting algorithms may have to be supplied in practice.

We verified our extrapolation approach with weak scaling codes. We conducted these experiments with the NPB BT, EP, FT, IS and LU codes, and the Sweep3D neutron-transport kernel [Wasserman et al. 2000]. (For other NPB codes, such as CG, weak scaling inputs could be easily be constructed.) Unlike Sweep3D, the NPB codes are originally designed as strong scaling benchmarks. Hence, we manually changed the input to provide weak scaling workloads.

In the first experiment, we verified the correctness of the generated traces with respect to the extrapolated communication topologies and the values of the communication parameters. We applied similar tests, *i.e.*, static trace comparison and mpiP results comparison, for the synthetically generated traces under weak scaling. The results show that the extrapolated traces are able to correctly preserve the communication semantics, and hence demonstrate the applicability of our topology extrapolation algorithm for applications under weak scaling.

In the second experiment, we evaluated the accuracy of the extrapolated timings for weak scaling problems. We used BT, EP, FT, IS, LU, and Sweep3D benchmarks on runs with up to 16,384 MPI processes. We first generated four traces for each benchmark as the extrapolation basis. Because these base traces were obtained from a series of weak scaling executions, the timing information preserved in the traces also reflects the weak scaling trend. We then performed the extrapolation with SCALAEXTRAP and replayed the extrapolated traces to obtain and evaluate the accuracy of the total runtimes against the original runs. In these experiments, we observed that the duration of most of the computational phases remains consistent during scaling. This is because for a given weak scaling input, the per-node problem size is fixed irrespective of the node size. We observed this trend for all benchmarks in this experiment, including the simpler ones such as EP and FT as well as the more complicated ones such as Sweep3D and LU. This observation is consistent with our empirical knowledge about the nature of weak scaling codes. In BT, we also observed a more complicated timing trend for some of the computational phases caused by the 3D layout of the problem. Nevertheless, our curve-fitting approach remains still applicable. Figure 13 depicts the extrapolation accuracy of the benchmarks. Quantitatively, the mean absolute percentage accuracy (as defined in Section 5.2) across all benchmarks and test cases is 92.87%. Among all the tests, IS with 4,096 nodes has a relatively low timing accuracy. IS uses bucket sort to distribute the elements to different nodes and then sorts the local elements within each node. When run with 4,096 nodes with the weak scaling input we provided, IS failed to balance the workload across nodes. Since the base traces for extrapolation were obtained from runs with good load balance, we cannot extrapolate the inconsistent computational duration caused by the unbalanced workload. We thus simply could not reproduce the unbalanced timing behavior during replay. Overall, our timing extrapolation approach is able to accurately predict the runtimes without experiencing additional challenges beyond those observed under strong scaling. With such a high timing accuracy and the proven communication semantics, the extrapolated weak scaling traces can be used for trace-based system simulation or other performance analysis experiments.

5.3. Lossy Extrapolation

Compared to all other benchmarks discussed so far, MG demonstrates the most complicated communication pattern. Overall, MG has a 3D communication topology. All nodes participate in a regular 7-point 3D torus-style communication. However, as a minor communication pattern, nodes at particular positions also communicate to nodes one hop (distance of 2) away in the 3D space. This communication is rank dependent even for those participating nodes, which makes the per-node traces highly divergent and the size of the final trace non-scalable. What is even more challenging is that the non-SPMD communication pattern does not exhibit any regular spacial property in forming the communication groups, i.e., there is little information to be derived from smaller traces about how a node will behave at larger scale. As a result, we cannot do the extrapolation by utilizing a general approach.

To extrapolate MG, we applied the lossy extrapolation approach. Specifically, we applied the snippet-based trace event filtering at the intra-node level to eliminate the minor rank dependent communication events. With events dropped, the generated traces consist of only the dominating regular 3D torus communication and the collectives. When compared across different node sizes, the traces are structurally identical, which indicates a perfect compression. As the result, the extrapolation of MG is largely simplified and becomes equivalent to the extrapolation of the 3D torus micro-benchmark.

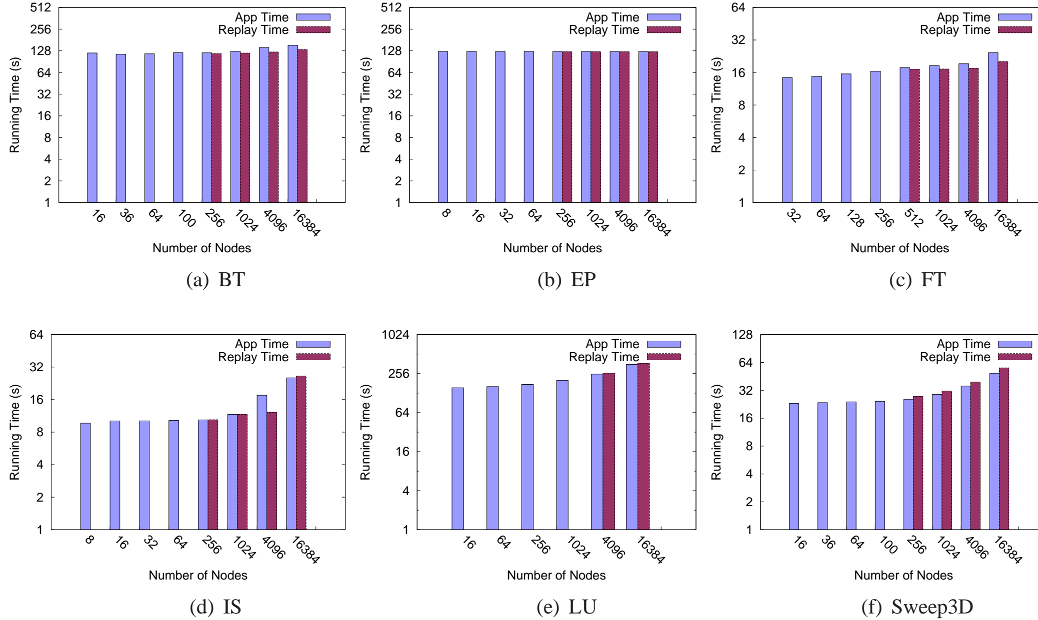


Fig. 13. Replay Time Accuracy for Weak Scaling Benchmarks

We evaluated the lossy extrapolation of MG with both strong scaling and weak scaling configurations. For weak scaling experiments, we changed the input generator to provide weak scaling inputs. We first evaluated the ability of extrapolated traces to preserve communication events. We replayed each extrapolated trace with mpiP instrumentation under SCALAREPLAY. We subsequently compared the generated mpiP results (profile counts) with those obtained by executing the mpiP-instrumented original benchmark over the same number of MPI processes. The experimental results show that for all the extrapolated sizes the number of dropped events is constantly less than 5% of the total number of MPI events, which indicates the communication workload is well preserved. Figure 14 compares the replay times of the extrapolated traces to the runtimes of the original benchmark. Due to inaccurate curve fitting, the replay times of the extrapolated traces are slightly longer than the original runtimes at large scales. Nonetheless, even after increasing the replay times by 5% (given less than 5% of the events were filtered), the extrapolated traces were still able to reflect the total runtime of the original benchmark.

When configured for strong scaling, MG falls into the category of applications that do not follow the dominating communication pattern at smaller scales, which presents a challenge. On the positive side, we are able to preserve and extrapolate all the collectives and the regular 7-point 3D torus communication pattern, the latter of which is measured to be the dominating pattern at smaller scales. However, as the problem size per node decreases, the number of the 3D torus communication events also decreases. Meanwhile, the number of the MPI events corresponding to the minor communication pattern stays constant and starts to domineer at larger scales. For example, with the lossy tracing approach, 96.98% of the `MPI_Send` operations were preserved in the 64-node trace of MG while only 77.56% were preserved in the trace of 1024 nodes. As a result, the extrapolated trace loses its ability to preserve most of the communication workload, even though one of the two overlapping communication patterns is fully captured. Since our trace-based extrapolation is a black box approach relying only on information in the input traces instead of knowledge about the

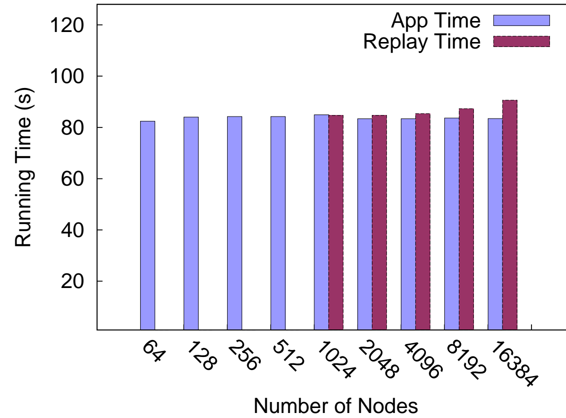


Fig. 14. Timing Accuracy of Lossy Extrapolation of Weak Scaling MG

source code of the application, the extrapolation of the strong scaling for MG is beyond the scope of SCALAEXTRAP's current capabilities.

6. APPLICATIONS OF TRACE EXTRAPOLATION

Extrapolated traces provide the means to correctly predict the communication semantics and timing behavior of the original applications at large scales as demonstrated in our experiments. This capability enables the extrapolated traces to be used for performance analysis. For example, traces can be fed into discrete-event-based simulators, such as DIMEMAS, for performance simulation. They can also be used as the input for visualization tools, such as VAMPIR Next Generation. In this section, we describe two case studies to demonstrate potential use cases of the extrapolated traces. Experiments described in this section were performed on ARC, a cluster with 1728 cores on 108 compute nodes, 32 GB memory per node and both Infiniband and Ethernet interconnects.

6.1. Code Generation from Extrapolated Traces

In the first case study, we use the extrapolated traces to generate parallel benchmarks. The main idea of benchmark generation is to automatically generate 1) MPI events with the same parameter values and temporal ordering, and 2) sleeps that mimic the computation stages in the original application based on the information of the input trace [Wu et al. 2011; Deshpande 2011]. The drawback of the trace-based benchmark generation approach is that the generated code can only be launched with the same number of MPI processes with which the trace was collected. Our extrapolation approach compliments such generation approaches by overcoming precisely this restriction: With the extrapolated trace, the benchmark generator may auto-generate code for arbitrary number of MPI processes that accurately reflects application behavior at that size.

To demonstrate the idea, we generated C+MPI code from the extrapolated Sweep3D traces. In this experiment, we first ran the SCALATRACE-instrumented Sweep3D code at node sizes of 16, 36, 64, and 100, to collect the base traces for extrapolation. We then extrapolated a series of traces for the node sizes of 144, 196, 256, 324, and 400, which were subsequently fed into the benchmark generator to generate C+MPI parallel benchmarks. Figure 15 compares the total execution times of the generated benchmarks to that of the original applications at the same node size. Quantitatively, the mean absolute percentage accuracy (as defined in Section 5.2) across all test cases is 93.76%. With such a high timing accuracy, the generated benchmarks can be used for performance analysis experiments, especially for novel interconnects or cross-platform test. In effect, the auto-generated benchmark serves as a substitute for the original application. This is particularly beneficial when

the original application's source code was classified since the auto-generated benchmark also obfuscated the original code can be release without restrictions.

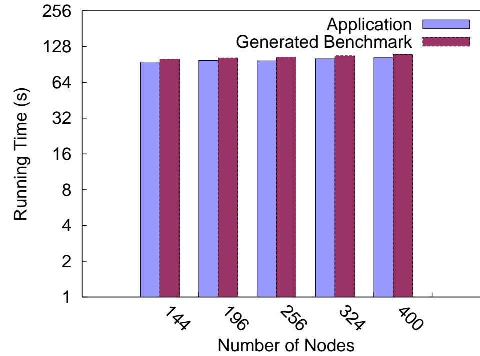


Fig. 15. Timing Accuracy for the Benchmarks Generated from the Extrapolated Traces

6.2. Performance Experiments with Extrapolated Traces

The second case study utilizes the extrapolated trace to analyze the impact of computational speedup on the overall performance. Computational speedup can be achieved in multiple ways. For example, application developers can optimize performance by overlapping communication and computation. They may also manually or automatically parallelize their code to take advantage of the compute power of the multi-core/many-core architectures. A current trend in high-performance computing is to supplement general-purpose CPUs with more special-purpose computational accelerators (e.g., GPUs). Unfortunately, it is neither trivial to predict how fast a parallel application will run once accelerated nor to port a parallel application to an accelerated architecture.

To readily assess the effect of computational speedup before implementation, application developers can perform a quick what-if analysis by modifying the application trace. A unique feature of SCALATRACE is that the collected trace is concise and structure-preserving. This opens the door to manual inspection and modification of traces for performance analysis. Since the extrapolated trace is equivalent to the trace obtained from a real execution, it shares the same merits (and shortcomings) as the original code.

As an example, we assess the impact of computational speedup on the overall performance of Sweep3D with the extrapolated Sweep3D trace. In this experiment, we used the Ethernet interconnect on ARC, which is slower than Infiniband. We first extrapolated the Sweep3D trace of 400 MPI processes from smaller traces. We then changed the computation times and message volumes recorded in the trace to simulate different expected improvements due to (a) hardware acceleration and (b) fluctuations in the communication workload. Total execution time was measured to reflect the change of the overall performance. Results are shown in Figure 16. The x-axis indicates the length of the generated computational phases in percentage of the original delta time. The different curves correspond to different message volumes. We observe an interesting phenomenon: Reducing the computation times, *i.e.,*"accelerating" the computational stages, does *not always* lead to a better overall performance. For example, when the message volume is increased from 1 to 2 and then 4 times of the actual values, the best overall performance is achieved when the sleep time is set to be 10%, 20%, and 40% of the original, respectively. This implies a speedup for 10x, 5x, and 2.5x for the computational parts, respectively. Particularly, when the message volume is set to 8 times of the actual value, the optimal sleep time is 180% of the original computation times, which indicates that, instead of trying to accelerate, application developers actually should slow down the computation stages to achieve the best overall performance. To understand this puzzling behavior, note

that Sweep3D is a stencil code consisting almost exclusively of synchronous point-to-point communication operations. Shortening the sleep times increases the contention in the network, which in turn offsets the time saved with computational speedup. Moreover, the larger the message volume, the heavier the network congestion would be. As a result, the optimal sleep time increases with the message volume.

In prior work, Hoisie *et al.* studied the Sweep3D code through performance modeling [Hoisie et al. 1999]. With their model, the authors showed that the single-node efficiency, rather than the inter-processor communication performance, is the dominant performance bottleneck. The apparent discrepancy is caused by the fact that the impact of communication performance is evaluated from different aspects. In Hoisie's work, the time of a single communication stage is modeled as $T_{msg} = t_0 + N_{msg}/B$, where t_0 is the latency and overhead, and B is the bandwidth in the LogGP model. Subsequently, they fixed the bandwidth and modified the latency t_0 from 10 μs to 0.1 μs and showed that decreasing the MPI latency by a factor of 100 reduces runtime by only less than a factor of two. Our work studies a different aspect. We show that the available bandwidth B is reduced significantly with increasing congestion. When the message size N_{msg} is large (as is the case in Sweep3D), the change of bandwidth dominates the change of the communication time T_{msg} and thus substantially decreases overall performance.

We should note that the experimental result presented in Figure 16 is both application-specific and platform-specific. Yet, with the application trace, what-if analysis on application performance can easily be performed without the need to implement a parallel skeleton algorithm or even port the original application. Besides, with the trace-based extrapolation, performance analysis can be performed even without the large-scale input data set or the necessary hardware, *e.g.,* without purchasing a large number of GPUs and deploying them over compute nodes.

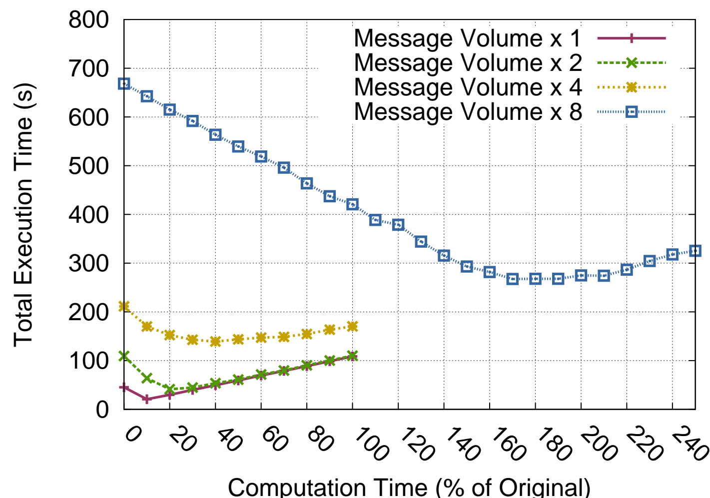


Fig. 16. The Impact of Computational Speedup on the Overall Performance

7. RELATED WORK

SCALATRACE is an MPI trace-gathering framework that generates near constant-size communication traces for a parallel application regardless of the number of nodes while preserving structural information and temporal ordering [Noeth et al. 2009; Ratn et al. 2008] (see Section 2. Our extrapolation work builds on the trace representation of SCALATRACE.

Xu *et al.* construct coordinated performance skeletons to estimate application execution time in new hardware environments [Xu et al. 2008; Xu and Subhlok 2008]. They detect dominant communication topologies by comparing an application communication matrix against a predefined set (library) of reference patterns. In this work, complicated communication patterns, such as the NAS benchmark CG, are handled by manually provided specifications of the new patterns. Moreover, the graph spectrum analysis and graph isomorphism tests utilized in this work lack scalability in terms of time complexity and thus limit the applicability of this work at large sizes. Most significantly, their work does not capture all communication events.

Zhai *et al.* collect MPI communication traces and extract application communication patterns through program slicing [Zhai et al. 2009]. This work utilizes a set of source code analysis techniques to build a program slice that only contains the variables and code sections related to MPI events, and then executes the program slice to acquire communication traces. While removing the computation in the original application enables a fast and cheap trace collection, it also causes the loss of temporal information that is essential for characterizing the application runtime behavior. In addition, the lack of trace compression limits its feasibility for large-scale application tracing. Based on the FACT framework, Zhai *et al.* employ a deterministic replay technique to predict the sequential computation time of one process in a parallel application on a target platform [Zhai et al. 2010]. The main idea is to use the information recorded in the trace to simulate the execution result of MPI calls when there is actually only one MPI process, and utilize the deterministic data replay to simulate the runtime of the computation phases on the target platform. While this approach manages to predict the computation time, it fails to capture the communication related effects. In addition, this work focuses on cross-platform performance prediction but cannot predict the application performance on a cluster that is larger than the available host platform.

DIMEMAS is a discrete-event-based network performance simulator that uses PARAVER traces as input [Pillet et al. 1995]. It simulates the application behavior on the target platform with specified processor counts and network latency. However, DIMEMAS simulations are infeasible for peta-/exascale simulations due to a lack of hardware resources to generate the input trace and the sheer size of traditional application traces. Hermanns *et al.* presented a method for verifying hypotheses on causality between distant performance phenomena in large-scale MPI Applications [Hermanns et al. 2009]. By manipulating the application trace, replaying, and performing wait-state analysis on the simulated event trace, this approach facilitates the what-if analysis for hypothesized performance problems. Our work, in contrast, focuses on the trace extrapolation for larger platforms that applications have not yet been ported to or even future platforms (exascale). The extrapolated traces can subsequently be used for various purposes that a normal trace might be used for, such as 1) deterministic replay with SCALAREPLAY, 2) the input trace for simulators (DIMEMAS/SST) for performance prediction, and 3) benchmark generation for performance experiments, as described in Section 6. Ronsse *et al.* presented $Rolt^{MT}$, an extension of ROLT (Reconstruction of Lamport Timestamps) for message passing systems [Ronsse and Kranzlmueller 1998]. In $Rolt^{MT}$, send events related with a promiscuous receive event are attached with Lamport timestamps incremented by a positive number. Since only timestamps for these events are recorded in the trace, the trace size and the program perturbation caused by tracing are minimized. The recorded Lamport timestamps are then used in replay with additional synchronizations to allow a deterministic replay of programs with non-deterministic receives.

Preissl *et al.* extract communication patterns, *i.e.*, the recurring communication event sets, from MPI traces [Preissl et al. 2008]. They first search for repeating occurrences of identical events in the trace of each individual process and then iteratively grow them into global patterns. The output of this algorithm can be used to identify potential bottlenecks in parallel applications. Preissl *et al.* further utilize the detected communication patterns to automate source code transformations such as automatic introduction of MPI collectives [Preissl et al. 2008]. Our method, in contrast, focuses on the spatial aspect of communication events, *i.e.*, the identification and extrapolation of communication topology.

Eckert and Nutt [Eckert and Nutt 1996; 1994] extrapolate traces of parallel shared-memory applications. They take as input the traces collected on an existing architecture and extrapolate them to a target platform with different architectural parameters, without re-executing the original application. This work analyzes the causal event stream. It focuses on the correctness of the extrapolated trace given the existence of program-level non-determinism, *e.g.*, the interleaving of events or modifications in the actual set of events caused by moving the trace across different architectures. In contrast, our work is based on deterministic application execution. We also preserve the causal ordering of communication events but our focus is on the communication behavior at arbitrary problem sizes.

Performance modeling has traditionally taken the approach of algorithmic analysis, often combined with tedious source code inspection and hardware modeling for floating-point operations per second, memory hierarchy analysis from caches over buses to main memory and interconnect topology, latency and bandwidth considerations. In particular, Kerbyson *et al.* present a predictive performance and scalability model of a large-scale multidimensional hydrodynamics code [Kerbyson et al. 2001]. This model takes application, system, and mapping parameters as input to match the application with a target system. It utilizes a multitude of formulae to characterize and predict the performance of a scientific application. Snavely *et al.* model and predict application performance by 1) characterizing a system with machine profiles, namely single processor performance and network latency and bandwidth, 2) collecting the operations in an application to generate application signatures, and 3) mapping signatures to profiles to characterize performance [Snavely et al. 2002; Bailey and Snavely 2005]. Gruber *et al.* describe PatternTool, an interactive tool for creating scalable hierarchical graphs to define the communication patterns and control flows of a parallel algorithm graphically [Gruber et al. 1996]. The created performance model can be used as the input to PAPS (Performance Prediction of Parallel Systems), a simulator that can be parameterized for different computer systems for what-if performance analysis. Ïpek *et al.* follow a completely different approach by utilizing artificial neural networks (ANNs) to predict the performance when application configuration varies [Ïpek et al. 2006]. This approach employs repeated sampling of a small number of points in the design space that are statistically determined through SimPoint [Sherwood et al. 2002]. Only these points are then simulated and results are utilized to teach the ANNs, which are subsequently utilized to predict the performance for other design points. In contrast, our work explores the potential of extrapolating the application runtime according to its evolving trend across increasing problem sizes. Since this method requires neither measurement of performance metrics nor intense computation, it provides a simple and highly efficient approach to study the effect of scaling across a large numbers of compute nodes. In contrast to all of the above approaches, our SCALAEXTRAP does not just simulate communication behavior at scale but allows such behavior to be observed in practice through replaying on a target platform with large numbers of nodes, even if the corresponding application itself has not been ported yet.

8. CONCLUSION

Scalability is one of the main challenges of scientific applications in HPC. Advanced communication tracing techniques achieve lossless trace collection, preserve event ordering and encapsulate time in a scalable fashion. However, estimating the impact of scaling on communication efficiency is still non-trivial due to execution time variations and exposure to hardware and software artifacts.

This work contributes a set of algorithms and analysis techniques to extrapolate communication traces and execution times of an application at large scale with information gathered from smaller executions. The extrapolation of communication traces depends on an analytical method to characterize the communication topology of an application. Based on the observation that problem scaling increases/decreases communication parameters and topology at a certain rate, we utilize a set of linear equations to capture the relation between communication traces for changing number of nodes and extrapolate communication traces accordingly. For the extrapolation of communication traces, the detection of communication topology is non-trivial but also vital. We currently focus on stencil/mesh topology with nodes arranged in a row-major fashion. While a large amount of parallel applications fall into this category, we observed more complex communication topologies that are

hard to detect with a generic approach, which limits the applicability of this work. In future work, user plugins will be supported so that the extrapolation of complicated and unique communication patterns can be facilitated by user-supplied information.

For the extrapolation of timing information, we utilize curve fitting approaches to model trends in delta times over traces with varying number of nodes. Statistical methods are further employed to mitigate timing fluctuations under scaling. We currently capture four categories of the most commonly seen timing trends. However, the prediction of more complicated timing trends, including detection of combinations of currently supported timing trends, may require more sophisticated algorithms.

Experiments were conducted using an implementation through our SCALAEXTRAP tool and with the NAS Parallel Benchmark suite and Sweep3D. We utilized up to 16,384 nodes of a 73,728-node IBM Blue Gene/P. Experimental results show that our algorithm is capable of extrapolating stencil/mesh and collective communication patterns for both strong scaling and weak scaling configurations. Extrapolation of timing information is further shown to provide good accuracy.

We believe that extrapolation of communication traces for parallel applications at arbitrary scale is without precedence. Without porting applications, communication events can be replayed and analyzed in a timed manner at scale. This has the potential to enable otherwise infeasible system simulation at the exascale level.

REFERENCES

BAILEY, D. AND SNAVELY, A. 2005. Performance modeling: Understanding the present and predicting the future. In *Euro-Par Conference*.

BAILEY, D. H., BARSZCZ, E., BARTON, J. T., BROWNING, D. S., CARTER, R. L., DAGUM, D., FATOOHI, R. A., FREDERICKSON, P. O., LASINSKI, T. A., SCHREIBER, R. S., SIMON, H. D., VENKATAKRISHNAN, V., AND WEERATUNGA, S. K. 1991. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications 5,* 3, 63–73.

BERGROTH, L., HAKONEN, H., AND RAITA, T. 2000. A survey of longest common subsequence algorithms. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*. IEEE Computer Society, Washington, DC, USA, 39–.

BRUNST, H., KRANZLMÜLLER, D., AND NAGEL, W. 2005. Tools for Scalable Parallel Program Analysis - Vampir NG and DeWiz. *The International Series in Engineering and Computer Science, Distributed and Parallel Systems 777*, 92–102.

DESHPANDE, V. 2011. Automatic Generation of Complete Communication Skeletons from Traces. M.S. thesis, North Carolina State University, Raleigh, NC, USA.

ECKERT, Z. AND NUTT, G. 1996. Trace extrapolation for parallel programs on shared memory multiprocessors. Tech. Rep. TR CU-CS-804-96, Department of Computer Science, University of Colorado at Boulder, Boulder, CO.

ECKERT, Z. K. F. AND NUTT, G. J. 1994. Parallel program trace extrapolation. In *International Conference on Parallel Processing*. 103–107.

FARAJ, A., PATARASUK, P., AND YUAN, X. 2007. A study of process arrival patterns for MPI collective operations. In *International Conference on Supercomputing*. 168–179.

GROPP, W., LUSK, E., DOSS, N., AND SKJELLUM, A. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing 22,* 6, 789–828.

GRUBER, B., HARING, G., KRANZLMUELLER, D., AND VOLKERT, J. 1996. Parallel programming with capse – a case study. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on 0*, 0130.

GUSTAFSON, J. L. 1988. Reevaluating Amdahl's law. *Communications of the ACM 31,* 5, 532–533.

HERMANNS, M.-A., GEIMER, M., WOLF, F., AND WYLIE, B. J. N. 2009. Verifying causality between distant performance phenomena in large-scale mpi applications. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE Computer Society, Washington, DC, USA, 78–84.

HOISIE, A., LUBECK, O. M., AND WASSERMAN, H. J. 1999. Performance analysis of wavefront algorithms on very-large scale distributed systems. In *Workshop on Wide Area Networks and High Performance Computing*. Springer-Verlag, London, UK, 171–187.

ÏPEK, E., MCKEE, S. A., CARUANA, R., DE SUPINSKI, B. R., AND SCHULZ, M. 2006. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 195–206.

KERBYSON, D., ALME, H., HOISIE, A., PETRINI, F., WASSERMAN, H., AND GITTINGS, M. 2001. Predictive performance and scalability modeling of a large-scale application. In *Supercomputing*.

KERBYSON, D. J. AND HOISIE, A. 2006. Performance modeling of the blue gene architecture. In *JVA '06: Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing*. 252–259.

KNÜPFER, A., BRENDEL, R., BRUNST, H., MIX, H., AND NAGEL, W. E. 2006. Introducing the open trace format (OTF). In *International Conference on Computational Science*. 526–533.

LABARTA, J., GIRONA, S., AND CORTES, T. 1997. Analyzing scheduling policies using dimemas. *Parallel Computing 23,* 1-2, 23–34.

NAGEL, W. E., ARNOLD, A., WEBER, M., HOPPE, H. C., AND SOLCHENBACH, K. 1996. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer 12,* 1, 69–80.

NOETH, M., MUELLER, F., SCHULZ, M., AND DE SUPINSKI, B. R. 2007. Scalable compression and replay of communication traces in massively parallel environments. In *International Parallel and Distributed Processing Symposium*.

NOETH, M., MUELLER, F., SCHULZ, M., AND DE SUPINSKI, B. R. 2009. Scalatrace: Scalable compression and replay of communication traces in high performance computing. *Journal of Parallel Distributed Computing 69,* 8, 969–710.

PILLET, V., LABARTA, J., CORTES, T., AND GIRONA, S. 1995. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*. Transputer and Occam Engineering Series, vol. 44. 17–31.

PREISSL, R., KÖCKERBAUER, T., SCHULZ, M., KRANZLMÜLLER, D., SUPINSKI, B. R. D., AND QUINLAN, D. J. 2008. Detecting patterns in mpi communication traces. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*. IEEE Computer Society, Washington, DC, USA, 230–237.

PREISSL, R., SCHULZ, M., KRANZLMÜLLER, D., SUPINSKI, B. R., AND QUINLAN, D. J. 2008. Using mpi communication patterns to guide source code transformations. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part III*. Springer-Verlag, Berlin, Heidelberg, 253–260.

RATN, P., MUELLER, F., DE SUPINSKI, B. R., AND SCHULZ, M. 2008. Preserving time in large-scale communication traces. In *International Conference on Supercomputing*. 46–55.

RODRIGUES, A. F., MURPHY, R. C., KOGGE, P., AND UNDERWOOD, K. D. 2006. The structural simulation toolkit: exploring novel architectures. In *Poster at the 2006 ACM/IEEE Conference on Supercomputing*. 157.

RONSSE, M. AND KRANZLMUELLER, D. 1998. Roltmp-replay of lamport timestamps for message passing systems. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on 0*, 0087.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. 45–57.

SNAVELY, A., CARRINGTON, L., WOLTER, N., LABARTA, J., BADIA, R., AND PURKAYASTHA, A. 2002. A framework for performance modeling and prediction. In *Supercomputing*.

VETTER, J. AND MCCRACKEN, M. 2001. Statistical scalability analysis of communication operations in distributed applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

WASSERMAN, H., HOISIE, A., AND LUBECK, O. 2000. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *The International Journal of High Performance Computing Applications 14*, 330–346.

WU, X., MUELLER, F., AND PAKIN, S. 2011. Automatic generation of executable communication specifications from parallel applications. In *Proceedings of the international conference on Supercomputing*. ICS '11. ACM, New York, NY, USA, 12–21.

WU, X., VIJAYAKUMAR, K., MUELLER, F., MA, X., AND ROTH, P. C. 2011. Probabilistic communication and i/o tracing with deterministic replay at scale. In *ICPP*.

XU, Q., PRITHIVATHI, R., SUBHLOK, J., AND ZHENG, R. 2008. Logicalization of mpi communication traces. Tech. Rep. UH-CS-08-07, Dept. of Computer Science, University of Houston.

XU, Q. AND SUBHLOK, J. 2008. Construction and evaluation of coordinated performance skeletons. In *International Conference on High Performance Computing*. 73–86.

ZHAI, J., CHEN, W., AND ZHENG, W. 2010. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 305–314.

ZHAI, J., SHENG, T., HE, J., CHEN, W., AND ZHENG, W. 2009. Fact: fast communication trace collection for parallel applications through program slicing. In *Supercomputing*. 1–12.