ScalaExtrap: Trace-Based Communication Extrapolation for SPMD Programs [*]

Xing Wu
North Carolina State University
xwu3@ncsu.edu

Frank Mueller
North Carolina State University
mueller@cs.ncsu.edu

Abstract

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. 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 D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.4.8 [*Operating Systems*]: Performance—Modeling and prediction

General Terms Measurement, Performance

Keywords High-Performance Computing, Message Passing, Tracing, Performance Prediction

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 [7]. Most of these tools either obtain lossless trace information at the price of poor scalability [13] or preserve only aggregated statistical trace information to limit the size of trace files as in mpiP [22]. Recent work on communication tracing and time recording made a breakthrough in this realm. ScalaTrace introduced an effective communication trace representation and compression algorithm [14]. 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 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 [9, 10]. 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 [3]). 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. Such extrapolations are facilitated by ScalaTrace's compression scheme

[*] 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 grant 0937908.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0119-0/11/02. . . $10.00

that preserves application structure. In contrast, extrapolation with other trace formats, such as OTF [11], 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 a microbenchmark and several NAS Parallel Benchmark codes [1]. 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, parameters 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 [12, 19, 21]), 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 discusses the limitations of this work and uncovers future challenges. 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 [14]. 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.

```
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();
}
```

Figure 1. Sample Stencil Code for RSD and PRSD Generation

- *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 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.

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 [18]. Along with the intra-node and inter-node compression processes,

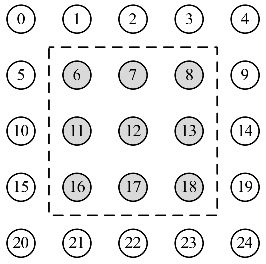


Figure 2. Ranklist Representation for Communication Group

"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) [12, 19, 21]. 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 $1024 = 32 \times 32$ 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 [24]. Therefore, instead of attempting to find a universal solution, we constrain our work to applications where

1. nodes are numbered in a row-major fashion and
2. 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 represent 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. Since the set of linear equations is constructed with the matching values of a communication parameter across traces of different node sizes, we further assume that

1. target applications are SPMD programs and
2. communication traces are compressed perfectly at both intra-node level and inter-node level so that the traces obtained from different node sizes are structurally identical (which only becomes feasible due to ScalaTrace's structure-preserving trace compression).

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 communication 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.

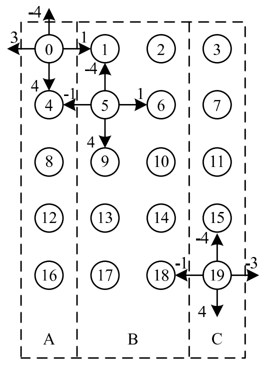


Figure 3. Topology Detection

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

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

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.

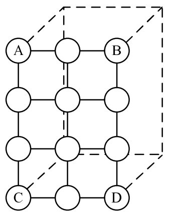


Figure 4. Boundary Size Calculation

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 Extrapolation of Communication Traces

The extrapolation of communication traces 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

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

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,

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

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

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 $c_i (1 \le i \le n - 1)$ 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 5 communicates with its *EAST* neighbor in a wrap-around manner. In order to extrapolate the rank of the communication endpoint, three input traces with dimensions 4×4, 5×5, and 6×6 are used to construct the set of linear equations shown in Figure 6, 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 $V = c_2 \times 10 \times 10 + c_1 \times 10 + c_0 = 91$.

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 7 shows the distribution of 9 communication groups of a 2D stencil code. Despite the changing problem size,

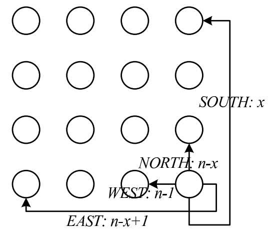


Figure 5. Generic Representation of Communication Endpoints

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

Figure 6. Set of Equations for Communication Endpoint Extrapolation

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.

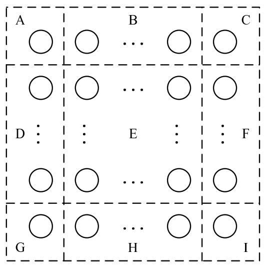


Figure 7. Distribution of Communication Groups of a 2D Stencil Code

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 7, 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>*, 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.

3.3 Handling Dynamic Load Balancing

The extrapolation of communication traces requires the input traces to be structurally identical so that the corresponding RSDs can be matched across different traces. Being structurally identical requires the traces to have the same number of RSDs and the matching RSDs should be generated by the same MPI event in the original application, though the values of the communication parameters and the loop information can be different.

However, the NPB code IS (Integer Sort) [1] results in structurally different traces due to an inherent dynamic workload balancing scheme. Specifically, in IS, each node first sorts their local elements into buckets, then calls MPI_Alltoallv to distribute the buckets to different nodes. Since the bucket sizes are determined by the randomly generated elements, the message sizes of MPI_Alltoallv are different in each iteration both within and across nodes. In addition, to create different program behavior, IS changes the values of several elements every iteration, so the parameters for MPI_Alltoallv keep changing across timesteps. As a result, the compression of MPI_Alltoallv fails for both intra- and inter-node phases, which leads to structurally different traces.

In fact, MPI_Alltoallv supports different message volumes between different node pairs (in contrast to MPI_Alltoall). Compression based on identical parameter values across nodes rarely succeeds. Therefore, we decided to improve *ScalaTrace* to trade precision for a higher degree of abstraction that allows more aggressive trace compression and ultimately enables extrapolation. We observe that the total volume of data exchanged by MPI_Alltoallv across all nodes for each synchronous call represents the topology boundary data. This boundary data volume is constant across all nodes. Thus, we aggregate mismatched parameters, such as message payload sizes, across all nodes. We subsequently record the average value (per node) instead of actual parameter values, which diverges only slightly from the average value. Note that this improvement is not just a customization for extrapolation of IS but rather results in improved compression of *any* application with similar usage patterns of MPI_Alltoallv. Given a constant overall input data size, the message size of MPI_Alltoallv in IS follows an inverse-proportion relationship relative to the number of nodes. (A similar behavior is also observed for FT.) We consequently enhanced ScalaTrace with the ability to detect and handle curve fitting for inverse-proportional relations, *e.g.,* for message sizes. Utilizing this methodology, we are able to obtain perfectly compressed communication traces for IS, which enables extrapolation through ScalaExtrap.

3.4 Handling Unique Communication Patterns

While the above extrapolation algorithm applies to stencil/mesh topologies, which characterize communication of a large number of parallel applications, we also observed a more complex communication pattern that required explicit communication information for extrapolation. As an example, consider the communication topology of NPB CG in Figure 8. The primary communication pattern is repeated across each row. For a single node, both the amount and the distance of endpoints changes with the number of nodes. Specifically, when there are a total of $n = x^2 (x = 1, 2, 4, ...)$ nodes, each node will have lgx communication endpoints with distance

$$d = (-1)^{\lfloor \frac{r}{2^i} \rfloor} \times 2^i,$$

where $0 \leq i \leq lgx$ and r is the rank of the node. Moreover, there is a secondary communication pattern along the diagonals in which nodes at symmetric positions communicate.

In order to detect the communication topology of CG, we improved the topology detection algorithm so that communications originating from different locations in the source code are differen-

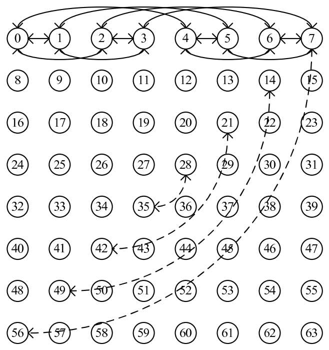


Figure 8. CG Communication Topology

tiated by their distinct call stacks. This enables us to handle applications with multiple interleaved communication topologies. Without the interference of diagonal communication, the loop detection process is able to identify the primary row-wise communication pattern and thus calculate the boundary sizes $x = y = \sqrt{n}$. The communication pattern of CG, however, is not linearly correlated with the topology sizes (which is beyond the scope of this work). We thus manually provide information to facilitate the extrapolation. In future work, we plan to enhance ScalaExtrap to support user-plugins that specify communication patterns. With this functionality, unique communication patterns can be analyzed by ScalaExtrap exploiting its extrapolation capability.

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 rebalances 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. We performed experiments with (a) one MPI task per node and (b) one per core. We report the results for the former because this configuration provides more memory per MPI task, which enables larger scale runs. Nonetheless, configurations (a) and (b) show equally accurate and correct results.

The extrapolation process is run on a single workstation and requires only 1 or 2 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) with class D and E input sizes [1]. We report our extrapolation results for BT, EP, FT, CG, LU, and IS. 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 as explained in Section 3.3. The communication trace extrapolation for CG is facilitated by manually specifying the communication pattern as a plugin function (see Section 3.4). The extrapolation of timing information does not require any extra information.

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 9).
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 9).
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 9).

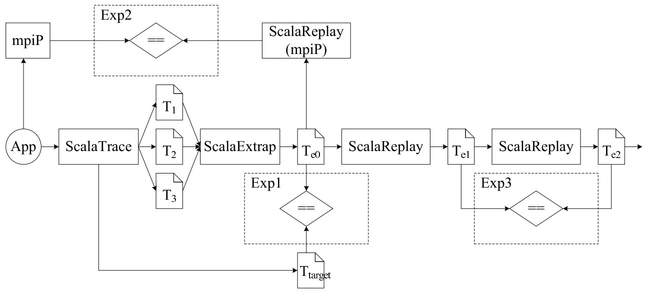


Figure 9. Correctness of Trace Extrapolation and Replay

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 9). 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 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.

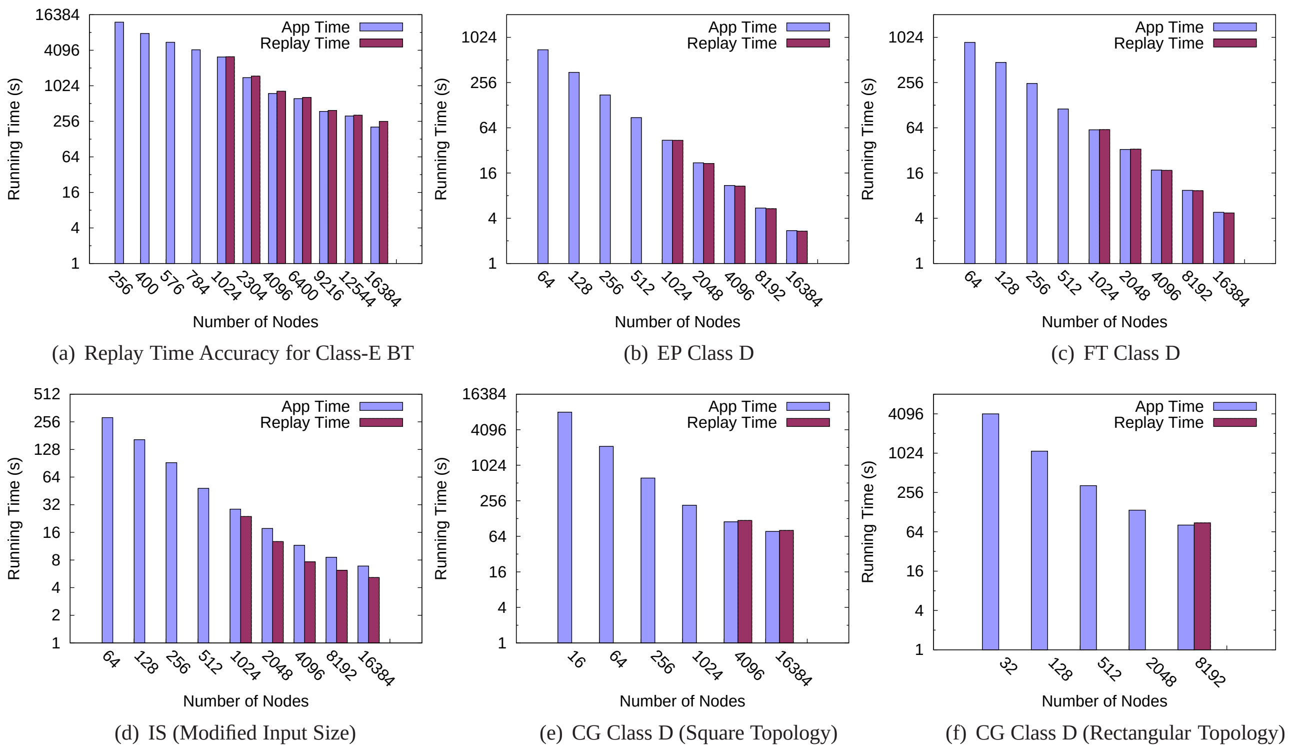
Second, we replayed the extrapolated trace T_{e_0} to assess if the MPI communication events are fully captured (see Exp2 in Figure 9). For this experiment, ScalaReplay is linked with mpiP [22], which yields frequency information of each MPI call distinguished by call site (using dynamic stackwalks). 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 9). 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 extrapolated traces for 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,



(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)

Figure 10. Replay Time Accuracy for Benchmarks

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 *min, max, avg, std.dev* 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 [14], 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 10 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 10, the timing extrapolation accuracy is generally higher than 90%, sometimes even higher than 98%, where accuracy is defined as

$$Accuracy = \frac{|Replay\ Time\ -\ App\ Time|}{App\ Time}.$$

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 [6], 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.

6. Discussion and Future Work

This work explored the extrapolation of the communication behavior of parallel applications, which is unprecedented. For the extrapolation of communication traces, the detection of communication topology is vital but non-trivial. 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 thus limits the applicability of this work. In future work, we plan to support user plugins so that the extrapolation of complicated and unique communication patterns can be facilitated by user-supplied information.

Extrapolation of timing information of parallel applications is another objective of this work. Currently, we capture four categories of the most commonly seen timing trend. However, the prediction of more complicated timing trends, *e.g.*, the detection of the combination of multiple types of timing trends, may require more sophisticated algorithms.

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 [14, 18] (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 [23, 24]. 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 [26]. 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 [25]. 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 [15]. 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. 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 then be either replayed with ScalaReplay (former case) or used as the input trace for simulators (Dimemas/SST) in the latter case for performance prediction.

Preissl *et al.* extract communication patterns, *i.e.*, the recurring communication event sets, from MPI traces [16]. 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 [17]. Our method, in contrast, focuses on the spatial aspect of communication events, *i.e.*, the identification and extrapolation of communication topology.

Eckert and Nutt [4, 5] 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 [9]. 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 [2, 21]. İpek *et al.* follow a completely different approach by utilizing artificial neural networks (ANNs) to predict the performance when application configuration varies [8]. This approach employs repeated sampling of a small number of points in the design space that are statistically determined through SimPoint [20]. 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 between traces and extrapolate communication traces accordingly. 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. Experiments were conducted using an implementation through our ScalaExtrap tool and with the NAS Parallel Benchmark suite. 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. 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

[1] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[2] D.H. Bailey and A. Snavely. Performance modeling: Understanding the present and predicting the future. In *Euro-Par Conference*, August 2005.

[3] H. Brunst, D. Kranzlmüller, and W. Nagel. 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, 2005.

[4] Z. Eckert and G. Nutt. Trace extrapolation for parallel programs on shared memory multiprocessors. Technical Report TR CU-CS-804-96, Department of Computer Science, University of Colorado at Boulder, Boulder, CO, 1996.

[5] Zulah K. F. Eckert and Gary J. Nutt. Parallel program trace extrapolation. In *International Conference on Parallel Processing*, pages 103–107, 1994.

[6] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. A study of process arrival patterns for MPI collective operations. In *International Conference on Supercomputing*, pages 168–179, June 2007.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[8] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. 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*, pages 195–206, 2006.

[9] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Supercomputing*, November 2001.

[10] Darren J. Kerbyson and Adolfy Hoisie. Performance modeling of the blue gene architecture. In *JVA '06: Proceedings of the IEEE John Vincent Atanasoff 2006 International Symposium on Modern Computing*, pages 252–259, 2006.

[11] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel. Introducing the open trace format (OTF). In *International Conference on Computational Science*, pages 526–533, May 2006.

[12] Jesús Labarta, Sergi Girona, and Toni Cortes. Analyzing scheduling policies using dimemas. *Parallel Computing*, 23(1-2):23–34, 1997.

[13] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, 1996.

[14] M. Noeth, F. Mueller, M. Schulz, and B. R. de Supinski. Scalatrace: Scalable compression and replay of communication traces in high performance computing. *Journal of Parallel Distributed Computing*, 69(8):969–710, August 2009.

[15] V. Pillet, J. Labarta, T. Cortes, and S. Girona. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44 of *Transputer and Occam Engineering*, pages 17–31, April 1995.

[16] Robert Preissl, Thomas Köckerbauer, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. Detecting patterns in mpi communication traces. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 230–237, Washington, DC, USA, 2008. IEEE Computer Society.

[17] Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. Supinski, and Daniel J. Quinlan. Using mpi communication patterns to guide source code transformations. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part III*, pages 253–260, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] P. Ratn, F. Mueller, Bronis R. de Supinski, and M. Schulz. Preserving time in large-scale communication traces. In *International Conference on Supercomputing*, pages 46–55, June 2008.

[19] Arun F Rodrigues, Richard C Murphy, Peter Kogge, and Keith D Underwood. The structural simulation toolkit: exploring novel architectures. In *Poster at the 2006 ACM/IEEE Conference on Supercomputing*, page 157, 2006.

[20] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, 2002.

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

[22] J. Vetter and M. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2001.

[23] Qiang Xu, Ravi Prithivathi, Jaspal Subhlok, and Rong Zheng. Logicalization of mpi communication traces. Technical Report UH-CS-08-07, Dept. of Computer Science, University of Houston, 2008.

[24] Qiang Xu and Jaspal Subhlok. Construction and evaluation of coordinated performance skeletons. In *International Conference on High Performance Computing*, pages 73–86, 2008.

[25] J. Zhai, W. Chen, and W. Zheng. Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 305–314, 2010.

[26] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng. Fact: fast communication trace collection for parallel applications through program slicing. In *Supercomputing*, pages 1–12, 2009.