Logicalization of Communication Traces from Parallel Execution

Qiang Xu*†, Jaspal Subhlok†, Rong Zheng† and Sara Voss‡†
{*qiang.xu@cggveritas.com, jaspal@uh.edu, rzheng@uh.edu, sarasvoss@gmail.com*}

*CGGVeritas, 10300 Town Park Drive, Houston, TX 77072
†Department of Computer Science, University of Houston, Houston, TX 77204
‡Coe College, Cedar Rapids, Iowa 52402

***Abstract*—Communication traces are integral to performance modeling and analysis of parallel programs. However, execution on a large number of nodes results in a large trace volume that is cumbersome and expensive to analyze. This paper presents an automatic framework to convert all process traces corresponding to the parallel execution of an SPMD MPI program into a single logical trace. First, the application communication matrix is generated from process traces. Next, topology identification is performed based on the underlying communication structure and independent of the way ranks (or numbers) are assigned to processes. Finally, message exchanges between physical processes are converted into logical message exchanges that represent similar message exchanges across all processes, resulting in a trace volume reduction approximately equal to the number of processes executing the application. This logicalization framework has been implemented and the results report on its performance and effectiveness.**

[1]

I. Introduction

Execution and communication traces are central to performance analysis and performance modeling of parallel applications. However, for long running applications on moderate to large number of nodes, even relatively coarse grained communication traces can be very long and their analysis prohibitively expensive. Fortunately high performance scientific applications are dominated by *stencil computations* where the computations, and hence the traces, are similar for all processes and the communication corresponds to a fixed *stencil* (or template) where each process communicates with a specific set of processes relative to a well defined global communication pattern, such as a 2-D torus or a structured tree. A study of DoD and DoE HPC codes at Los Alamos National Labs [9] and NAS benchmarks [14] shows that the communication in most of these codes is dominated by a single low degree stencil pattern. These characteristics expose the possibility of combining all processor traces into a single representative trace - in the same way as a single SPMD program represents a family of processes. We define a *logical trace* as a single aggregate trace that represents the execution and communication of a parallel program, with all message sends and receives to/from a logical neighbor in terms of a global communication topology (e.g a 2-D torus). We define *logicalization* as the process of converting a family of physical process traces into a single logical trace. This paper presents a framework for automatic construction of a logical program trace from the collection of physical process traces generated by an execution of an MPI message passing parallel application.

[1]Appears in 2009 IEEE International Symposium on Workload Characterization

For illustration, consider the following sections of traces from a message exchange between 4 processes in a 1-dimensional ring topology.

Process 0	**Process 1**	**Process 2**	**Process 3**
...	...	...	...
snd(P1,...)	*snd(P2,...)*	*snd(P3,...)*	*snd(P0,...)*
rcv(P3,...)	*rcv(P0,...)*	*rcv(P1,...)*	*rcv(P2,...)*
...	...	...	...

These can be summarized as the following logical trace:

Program
...
snd(P_R,...)
rcv(P_L,...)
...

where P_L and P_R are the logical left and right neighbor, respectively, for each process in a 1-D ring topology.

Beside reducing the trace size by a factor of, approximately, the number of processes, logicalization also captures the parallel structure of the application. Note that logicalization is orthogonal to *trace compression*, which is based on discovering repeating patterns within a single logical or physical trace. This paper focuses on trace logicalization, although trace compression and logicalization are often employed together, as in the context of this work, the construction of performance skeletons, discussed in Section II.

The logicalization framework has been developed for MPI programs and proceeds as follows. The application is linked with the PMPI library so that all message exchanges are recorded in a trace file during execution. Summary information consisting of the number of messages and bytes exchanged between process pairs is recorded and converted to a binary *application communication matrix* that identifies process pairs with significant message traffic. This matrix is then analyzed to determine the application level communication topology.

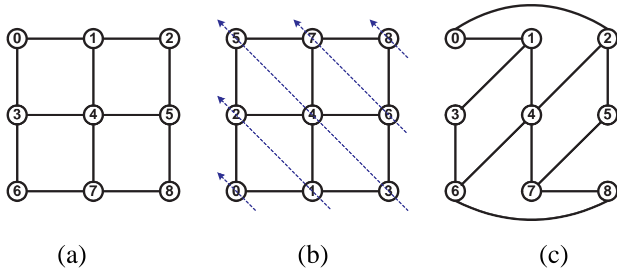

Fig. 1. Isomorphic graphs generated with different process numberings for a simple 2-D grid topology

Once this global topology is determined, a representative process trace is analyzed in detail and transformed into a logical program trace with endpoints of all message sends and receives converted from physical process numbers to relative logical process numbers in terms of the application level communication topology.

The key algorithmic challenge in this work is the identification of the application communication topology from the application communication matrix which represents the inter-process communication graph. The communication topology is easy to identify if the processes are assigned numbers (or ranks) in a well defined order, but is a much harder problem in general. This is illustrated with a very simple example in Figure 1. The figure shows 9 executing processes within a 2D grid communication topology. In Figure 1(a) the processes are assigned numbers in row major order in terms of the underlying 2D grid. However, if the processes were numbered diagonally as indicated in Figure 1(b), the communication graph with process nodes laid out in row major order would appear as Figure 1(c). Clearly, the underlying topology is easy to identify in the scenario represented in Figure 1(a) by a pattern matching approach but much harder when process numbering follows an unknown or arbitrary order, a simple instance of which is the scenario represented in Figure 1(c). The state of the art in identifying communication topologies assumes that a simple known numbering scheme is followed [9].

Identifying the underlying topology from a communication graph in general (i.e., without assuming any numbering scheme) is difficult for two reasons. First, establishing if a given communication graph matches a given topology is equivalent to solving the well known *graph isomorphism* problem for which no polynomial algorithms exist. (It is not known if it is NP-complete). Further, there are many different types of topologies (different stencils on graph/torus, trees, etc.) and many instantiations within each topology type (e.g., different number and sizes of dimensions). A naive method would require solving the graph isomorphism problem for each instance of each candidate topology, which is computationally infeasible. Our approach has the following main steps:

1) *Identification of candidate topologies:* Simple tests exist that can eliminate the possibility that a given topology could be a match for a given application communication matrix. For example, a 2D torus stencil topology is possible only if all processes have 4 communicating neighbors. Hence if there is any process with more or less than 4 communicating neighbors, this topology is eliminated. Of course, not every topology where all processes have 4 neighbors is a torus. In our method, a series of such tests are applied, from simplest to more complex as a decision tree, to reduce the set of possible candidate topologies. The tests are based on matching the following: *number of nodes and edges, sorted list of node degrees, and graph spectrum represented by the eigenvalues of the adjacency matrix*. Typically very few candidate topologies are left after this step, but even if it is just one, a match is not proved.
2) *Exact topology match:* This involves proving that the application communication graph is isomorphic to the corresponding reference topology graph. While graph isomorphism is known to be a difficult problem with no known polynomial solution, practical algorithms exist which can solve the problem efficiently for many scenarios [12]. Also, the size of the problem to be solved is relatively modest as it corresponds to the number of processes. We employ the VF2 graph matching algorithm from *VFlib2* library [2], [4] to establish an exact match with a candidate reference topology.

After the communication topology is established, a representative process trace is transformed to a logical program trace. The paper describes the design and implementation of the logicalization framework, along with experimental results and discussion.

II. Motivation and Context

The results developed in this paper have broad applicability in performance modeling, workload characterization, and debugging of message passing parallel programs. The specific context and usage of this research is construction of application performance skeletons for performance prediction. A performance skeleton is a short running program that captures the fundamental computation and communication characteristics of an application. Monitored execution of a performance skeleton in a new environment (e.g, different number of nodes, different communication library, or different network sharing) is employed to rapidly estimate the performance of the application (and the type of workload the application represents) in a new environment. The basic procedure for construction of performance skeletons consists of collection and compression of application traces followed by the generation of an executable program that recreates the core application behavior. The steps in the skeleton construction procedure are outlined in Figure 2. A framework for scalable skeleton construction has been developed and the effectiveness of performance skeletons for performance prediction have been evaluated [19], [13].

The highlighted logicalization step in Figure 2 is the focus of this paper and discussed in detail in the following sections. Table I presents summary results from the *combined*

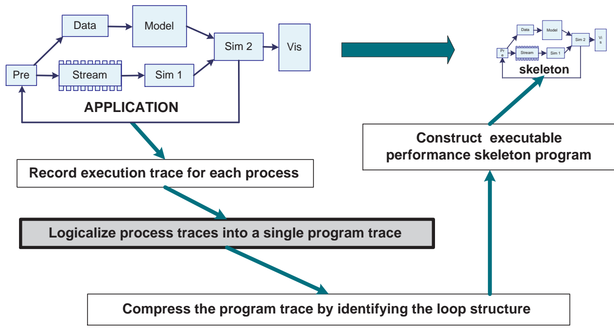

Fig. 2. Skeleton construction procedure

logicalization and compression phases for 16 process NAS benchmarks. The trace length is measured as the number of trace records (or lines), each representing one MPI operation. The logical trace is approximately the same size as a single process trace, hence the compression achieved in logicalization equals the number of processes. The compression ratio presented in Table I is the ratio of the size of full logical trace to the final compressed logical trace. Clearly the approach is effective in reducing a family of raw MPI traces to a short single compressed logical trace. The compression procedure and performance results are presented in detail in [20].

Benchmark Name	Raw Trace Length Per Process	Compressed Logical Trace Length	Compression Ratio
BT B/C	17106	44	388.77
SP B/C	26888	89	302.11
CG B/C	41954	10	41954
MG B	8909	590	15.1
MG C	10047	648	15.5
LU B	203048	63	3222.98
LU C	323048	63	5127.75
Average	71695	165	1815.39

TABLE I
COMBINED TRACE LOGICALIZATION AND COMPRESSION RESULTS

III. RELATED WORK

The importance of MPI traces in program analysis and visualization is clear from the popularity of tools like Vampir [1] and Jumpshot [18]. Several tools have been developed to perform statistical analysis of MPI communication behavior to summarize the execution behavior, an example being [15]. The idea of communication/adjacency matrix for trace analysis for parallel programs was introduced in [6], [7]. They used communication matrices to discover the logical topology employed in MPI and PVM applications to develop a parallel program debugger that exploits topological information. Our work can be considered a generalization of the approach with efficient detection of isomorphism.

Perhaps the work closest to this paper is the scalable trace compression presented in Noeth et. al. [10]. They perform task (or process) level compression on-the-fly, followed by consolidation of compressed traces that they refer to as inter-node compression. An important difference is that we perform logicalization (or inter-node compression) first on process traces, and subsequently perform trace compression. Since trace compression is performed only on a single logical program trace in our approach, we can use more effective compression procedures even if they are more expensive [19], [20]. Also, the logicalization procedure we employ is more general, independent of process numbering, and evaluated more thoroughly. On the other hand, our scheme as presented does require recording of the trace for every process which is a concern for scalability. This is discussed further in Section VI.

We have borrowed part of our pattern identification methodology from Kerbyson et.al. [9]. In this work, a point-to-point communication matrix is developed from application execution and the degree of *match* with a set of predefined communication templates, representing regularly occurring communication patterns in scientific applications, is measured. Their method assumes that the nodes are numbered in a “reasonable” way, e.g., along the rows or columns for a 2-D grid. The basic goal of our approach to topology identification is similar. However, more complex processing steps, that include eigenvalue and graph isomorphism computations, are necessary to identify communication patterns with no assumptions about the numbering of processes. Also, the motivation in this work is to understand the communication patterns in an application while our goal is to convert a suite of process traces into a single program trace.

An important limitation of the current framework is that the input trace must represent a relatively static communication pattern. This is known to be the case for many scientific applications [16], [9], [14]. A parallel application can have phases that show different communication behavior. Identification of phases at different levels of granularity has been studied in related work, such as [3], [11], [8].

IV. LOGICALIZATION METHODOLOGY

The trace logicalization procedure has the following main steps:

1) Generation of a binary application communication matrix from application process traces.
2) Identification of the application communication topology from the communication matrix.
3) Generation of a single logical program trace from a selected physical trace and topology information.

Each of these steps is presented separately in this section. The central assumption in our current implementation is that there is a dominant regular communication pattern in the trace being processed, else no topology is identified.

A. Generation of application communication matrix

For generation of a physical trace, the target MPI application is linked with the PMPI library, which allows lightweight recording of communication operations through user provided functions. During execution a trace file is generated for each process. Attributes recorded for each MPI call include the type

of call, the rank of the source/destination process, and the number of bytes transferred, along with timing information. This information is utilized to generate a *full application communication matrix*. The matrix records the total data transferred between each pair of processes involved in the execution.

The next objective is to convert the full communication matrix to a *binary application communication matrix*, where pairs of processes with *significant communication* are represented by 1 and pairs of processes with no communication or very low communication are represented by 0. This is achieved by threshold based *communication filtering*. Most parallel scientific applications show a distinct dominant communication pattern, typically a simple stencil. However, occasional minor communication is sometimes recorded between other processes. This can be inherent in the algorithm or due to other reasons, such as distribution and collection of data at the beginning and end of execution. Very low level communication (in terms of number of calls and volume of data exchanged) is eliminated as the procedure focuses on capturing, and later recreating, the dominant application characteristics. A heuristic threshold of 5% of average communication was used for the generation of the binary communication matrix from the full communication matrix in our experiments.

As a simple example, the full communication matrix and the binary communication matrix for the 8 process NAS MG benchmark are shown in Table II. Filtering was necessary for discovering the main communication pattern in some sizes of the MG benchmark, but not for any of the other benchmarks. The full communication matrix and the binary communication matrix after filtering for 16-process MG benchmark are shown in Table III. The volume of the communication not associated with the main communication pattern was around 0.5% compared to the main communication pattern.

This filtering step makes the overall logicalization process lossy, when it is invoked. An accuracy measure in the framework quantifies the extent of low volume communication that is eliminated. However, the impact of eliminating a small communication step on the end application of this work, such as performance modeling, is difficult to judge and beyond the scope of this paper. However, it has been analyzed with an end to end performance prediction framework that employs logicalization [19].

B. Topology identification

The procedure in the previous section yields a binary application communication matrix. We now present a procedure to determine if this application communication matrix represents an instance of a topology such as a stencil, tree or another pattern that is part of a reference library. This is a challenging problem for two major reasons. First, each type of pattern has numerous instances. For example, a 2-D grid is an instance of a grid pattern, and it represents a different $X \times Y$ grid for every distinct pair (X, Y). The number of instances can be large even when the total number of processes $X \cdot Y$ is fixed because of different factorizations. Thus, the question of *which graph to match with* is a non-trivial one. Second, for a graph representing a topology, numbering of vertices is not unique. Different numberings of graph vertices correspond to different permutations of the rows and columns of its adjacency matrix. A simple comparison of an application communication matrix with a communication matrix corresponding to a reference topology is not sufficient and *isomorphism* between the graphs represented by the matrices must be established. Two graphs G and H with graph vertices $V_n = \{1, 2, ..., n\}$ are said to be isomorphic if there is a permutation p of V_n such that $\{u, v\}$ is in the set of graph edges $E(G)$ iff $\{p(u), p(v)\}$ is in the set of graph edges $E(H)$. No polynomial algorithm exists to determine if two graphs are isomorphic although the problem is not known to be NP-complete.

The approach taken to topology identification attempts to minimize the use of potentially expensive graph isomorphism analysis. It consists of two phases, i) determining candidate patterns in the reference library and ii) determining an exact matching to a reference topology.

The first step is to reduce candidate patterns based on invariants that must hold for isomorphic graphs. It is widely believed that there is no simple-to-calculate complete graph invariant, i.e., there are simple-to-calculate invariants that hold across all isomorphic graphs, but there also exist non-isomorphic graphs which have the same set of invariants. Hence, graph invariants cannot be employed to establish isomorphism, but they can be utilized to narrow down the reference patterns that are potential matches. To compute candidate reference patterns, the following attributes of an application graph $G(V, E)$ are examined:

1) Number of vertices $|V|$
2) Number of edges $|E|$
3) Node degree in descending order
4) Graph spectrum, $\lambda(G)$, the set of eigenvalues of the adjacency matrix.

Clearly, the number of edges and vertices, as well as the list of nodes in ascending order of node degree, must be identical for isomorphic graphs. It is also known that the set of eigenvalues of the adjacency matrices of isomorphic graphs are identical. Hence, if any of these quantities do not match for a pair of graphs, they cannot be isomorphic.

Clearly the first three quantities are very simple to compute. The complexity for computing $\lambda(G)$ is $O(n^3)$ using the Gauss-Jordan reduction. Several solver packages exist for computing eigenvalues for sparse matrices [5]. In determining the candidate reference patterns, we adopt a decision tree based approach that eliminates most patterns efficiently by employing invariants in increasing order of computational complexity.

1) *Nodes, edges and prime factors* Let the number of vertices of the graph to be matched G be $|V| = n$. Let m be the maximum dimension of the Euclidean structures - graphs, tori and stencils based on them - in the reference library. The first step is to factorize n into products of the form $n_0 \cdot n_1 \cdots n_m$ [21]. This prime factor analysis is

KBytes	P0	P1	P2	P3	P4	P5	P6	P7
P0	0	141	144	0	148	0	0	0
P1	141	0	0	144	0	148	0	0
P2	144	0	0	141	0	0	148	0
P3	0	144	141	0	0	0	0	148
P4	148	0	0	0	0	141	144	0
P5	148	0	0	0	141	0	0	144
P6	0	0	148	0	144	0	0	141
P7	0	0	0	148	0	144	0	141

$\Longrightarrow$

	P0	P1	P2	P3	P4	P5	P6	P7
P0	0	1	1	0	1	0	0	0
P1	1	0	0	1	0	1	0	0
P2	1	0	0	1	0	0	1	0
P3	0	1	1	0	0	0	0	1
P4	1	0	0	0	0	1	1	0
P5	1	0	0	0	1	0	0	1
P6	0	0	1	0	1	0	0	1
P7	0	0	0	1	0	1	0	1

TABLE II
FULL APPLICATION COMMUNICATION MATRIX (TRAFFIC IN KBYTES/SEC) AND BINARY APPLICATION COMMUNICATION MATRIX FOR 8 PROCESS MG BENCHMARK

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
P2	89.09	0	0	87.09	0	0	91.19	0	0	0	0	0	0	0	91.19	0
P3	0	89.09	87.09	0	0	0	0	91.19	0	0	0	0	0	0	0	91.19
P4	91.21	0	0	0	0	87.09	89.10	0	91.23	0	0	0	***0.46***	0	0	0
P5	0	91.21	0	0	87.09	0	0	89.10	0	91.23	0	0	0	***0.46***	0	0

$\Downarrow$

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
P2	1	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0
P3	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	1
P4	1	0	0	0	0	1	1	0	1	0	0	0	***0***	0	0	0
P5	0	1	0	0	1	0	0	1	0	1	0	0	0	***0***	0	0

TABLE III
SECTIONS OF FULL AND BINARY COMMUNICATION MATRIX (TRAFFIC IN KBYTES/SEC) FOR 16-PROCESS MG BENCHMARK. THE HIGHLIGHTED ENTRIES ARE SMALL VALUES THAT ARE ELIMINATED BY FILTERING.

applied for the Euclidean patterns to arrive at a subset of reference graphs $S_1 = \{G_1^1, G_2^1, \ldots, G_{l_1}^1\}$, which have the same number of edges as G.

2) *Degree ordering* In the second step, we order the vertex degrees of $g \in S_1$ in descending order and eliminate those with different sequences. Let the resulting subset be $S_2 = \{G_1^2, G_2^2, \ldots, G_{l_2}^2\}$.
3) *Computing graph spectrum* As the final step, we compute the graph spectrum of $\lambda(G)$ and $\lambda(g)$, $\forall g \in S_2$ and eliminate those with different graph spectrum. Let the resulting subset be $S_3 = \{G_1^3, G_2^3, \ldots, G_{l_3}^3\}$.

By the end of this procedure, we may be left with a single candidate topology but cannot conclude that it is the matched topology. The invariants employed can eliminate a pattern from consideration but do not guarantee a match.

We apply a graph isomorphism algorithm to determine whether application communication graph G matches with any of the graphs in the candidate set S_3. While there are no known polynomial algorithms for graph isomorphism, efficient and practical solution approaches exist. We chose the VFLib 2.0 graph matching library [2], developed at the University of Naples "Federico II". VFLib2 implements the VF2 graph matching algorithm along with a few other algorithms including Schmidt-Druffel algorithm and Ullmann's algorithm. We chose VFlib2 library, in part, because of the ease of integration with C++ programs. Evaluation studies show that the VF2 algorithm can solve a graph isomorphism problem of thousands of nodes in less than a minute [2]. A comparison of different graph isomorphism algorithms is given in [12].

The VF2 algorithm takes a bottom-up approach [4]. It tries to extend an existing mapping of nodes and edges until a full mapping is reached, starting from the empty mapping. This is equivalent to a depth-first search in the tree of all possible permutations where branches that cannot lead to a feasible solution are pruned early.

C. Generation of logical execution trace

The logical communication trace of an application execution is similar to the physical communication trace generated at an execution node, except that all communication events refer to neighbors in a logical topology instead of a physical process number (or rank). For presenting the logical trace generation procedure, we assume that the application communication topology has been established.

We first define the set of *maximal communication* processes for a communication topology, as the set of processes that have all possible communication neighbors within the pattern. For fully symmetrical communication patterns, e.g 2D torus or All-All, all processes are maximal communication processes. However, that is not the case for asymmetrical communication patterns. For example, for a 2D grid pattern, all processes *except* the processes on the perimeter of the grid pattern (i.e. first and last rows and columns) are maximal communication processes as the interior processes have 4 neighbors, while the perimeter processes have 2 or 3 neighbors.

For every application communication topology D with a maximum of k communicating neighbors, we define $D_1, D_2, D_3, ...D_k$ as the set of logical neighbor processes/directions. For illustration, for a 2D grid or torus

structure, $k = 4$ and D_1, D_2, D_3, D_4 intuitively represent $North$, $East$, $South$, $West$ neighbors respectively.

We now describe the process of generating the logical communication trace from physical traces and a known application communication topology D with maximum communication degree k.

1) Identify one maximal communication process, say P_0
2) Let $i_1, i_2, i_3, ...i_k$ be the ranks of the processes P_0 communicates with.
3) Rewrite the trace of P_0 by replacing all references to $i_1, i_2, i_3, ...i_k$ in communication operations with corresponding references to $D_1, D_2, D_3, ...D_k$, respectively. This is the logical trace.

The logical trace represents the entire program execution and is interpreted with the corresponding communication topology. If a neighbor does not exist for a particular direction for a process number, (e.g., the $North$ neighbor for a process in top row of a grid), corresponding communication does not exist either. Finally, all collective operations are retained from the physical trace to the logical trace - collective operations are already global logical operations across the executing processes in the current *MPI COMMWORLD* communicator and no change is needed.

A section of physical and corresponding logical traces for the 16-process BT benchmark are shown in Table IV. Note that the directions are labeled as North, South, etc. for illustration and are actually indices in a general topology matrix.

Communication outside the main topology: Any communication operation in the physical trace that references a process rank not in the established topology is not included in the logical trace as corresponding operations do not exist across the parallel application. In fact, whenever communication filtering discussed in section III is applied, such local communication is present in the physical trace for some of the processes. In our implementation we record the fraction of communication that falls in this category and report it as an inaccuracy in this approach.

V. Experiments and results

The framework for application pattern identification and trace logicalization has been implemented. Experiments were conducted with MPI NAS benchmarks EP, MG, SP, BT, LU, CG, and FT executing with up to 128 processes on a cluster. We discuss the results for the benchmarks executing on 4, 8(9), 16 , 32(36) , 64 and 128(121) processes. (Some benchmarks run only on perfect square numbers of processes.)

A. Topology identification

A full application communication matrix was generated for each program and then converted to a binary communication matrix based on the discussion in Section III. The FT benchmark and EP benchmark showed no point-to-point communication and hence an empty communication matrix. The EP benchmark indeed has no communication. The FT benchmark only has collective All-All communication which implies that the physical trace is essentially the logical trace. We will not discuss the results from the FT benchmark any further.

The matching procedure was then applied to the benchmarks. The reference library employed for comparison initially consisted of the following patterns:

- Grids: Any number of dimensions
- Torus: Any number of dimensions
- Common stencils (6pt, 8pt) on 2D/3D meshes
- All to All
- Binary Tree

Note that the topologies listed are abstract and represent all sizes and dimensions. Also, it is fairly straightforward to add a new topology to the library. Hypercubes are not listed as they are special cases of a torus or a grid configuration. Also, the CG benchmark originally did not match any topology in the reference library. The topology of CG (a 3 point stencil) was manually analyzed and added to the library. The results presented include this addition.

The matching procedure consists of 3 distinct steps based on the description in Section IV-B, i.e., *Simple Tests* to eliminate most topologies, *Graph Spectrum Test* based on computing eigenvalues, and finally *Isomorphism Test* to establish a topology.

Table V lists the topologies that remain as candidates after each of the tests is applied, along with the final established topology. We discovered that many topologies in our abstract lists are themselves isomorphic to each other. In Table V every unique topology is in **boldface**. All topologies *not* in boldface and listed below one in boldface are isomorphic to the boldface topology above them. Note that BT and SP benchmarks have identical communication graphs and topologies and are listed together.

We make the following observations from Table V:

- All benchmarks in our test suite were matched correctly, although CG was matched only when a custom stencil was added. The topology of SP and BT is a 6 point stencil on a 2D grid (i.e. NE and SW neighbors in addition to N E W S), and for LU, CG, and MG, the topology is a grid or a torus. In fact, MG has a hypercube structure up to size 64 which is a special case of a grid/torus.
- The simple tests that we listed are very effective in reducing the set of candidate patterns. In all cases a very small set of candidate patterns were left after these tests were employed.
- The graph spectrum test was also effective, and in fact, eliminated all candidates except for the final correct topology. However, since the isomorphism test also must be employed, its value is unclear.

Traces for the benchmark programs were converted to logical traces. For all benchmarks except MG, each communication call in the trace was directly mapped to a logical call within the program's communication topology implying perfect "accuracy". In the case of MG, a low volume of communication could not be mapped to the application topology, as a consequence of filtering discussed in Section IV-A, and

PHYSICAL TRACE	***LOGICAL TRACE***
....	....
MPI_Isend(... **1**, MPI_DOUBLE, 480, ...)	MPI_Isend(...**EAST**, MPI_DOUBLE, 480, ...)
MPI_Irecv(... **3**, MPI_DOUBLE, 480, ...)	MPI_Irecv(...**WEST**, MPI_DOUBLE, 480, ...)
MPI_Wait() */* wait for Isend */*	MPI_Wait() */* wait for Isend */*
MPI_Wait() */* wait for Irecv */*	MPI_Wait() */* wait for Irecv */*
....	....
MPI_Isend(... **4**, MPI_DOUBLE, 480, ...)	MPI_Isend(...**SOUTH**, MPI_DOUBLE, 480, ...)
MPI_Irecv(...**12**, MPI_DOUBLE, 480, ...)	MPI_Irecv(...**NORTH**, MPI_DOUBLE, 480, ...)
MPI_Wait() */* wait for Isend */*	MPI_Wait() */* wait for Isend */*
MPI_Wait() */* wait for Irecv */*	MPI_Wait() */* wait for Irecv */*
....	....
MPI_Isend(... **7**, MPI_DOUBLE, 480, ...)	MPI_Isend(...**SOUTHWEST**, MPI_DOUBLE, 480, ...)
MPI_Irecv(...**13**, MPI_DOUBLE, 480, ...)	MPI_Irecv(...**NORTHEAST**, MPI_DOUBLE, 480, ...)
MPI_Wait() */* wait for Isend */*	MPI_Wait() */* wait for Isend */*
MPI_Wait() */* wait for Irecv */*	MPI_Wait() */* wait for Irecv */*
....	....

TABLE IV

SECTIONS OF A SAMPLE PHYSICAL TRACE (LEFT) AND CORRESPONDING LOGICAL TRACE (RIGHT) FOR THE BT BENCHMARK. THE TRACE IS SANITIZED FOR THE PURPOSE OF ILLUSTRATION.

Code	#P	Simple Tests	Graph Spectrum	Isomorphism
BT SP	9	**3×3 6-p stencil**	**3×3 6-p stencil**	**3×3 6-p stencil**
	16	**4×4 6-p stencil**	**4×4 6-p stencil**	**4×4 6-p stencil**
	36	**6×6 6-p stencil** 4×3×3 torus **2×2×3×3 torus**	**6×6 6-p stencil**	**6×6 6-p stencil**
	64	**8×8 6-p stencil** **2×2×2×2×2×2 grid** 4×2×2×2×2 torus 4×4×2×2 torus 4×4×4 torus	**8×8 6-p stencil**	**8×8 6-p stencil**
	121	**11×11 6-p stencil**	**11×11 6-p stencil**	**11×11 6-p stencil**
LU	8	**4×2 grid** CG stencil	**4×2 grid** CG stencil	**4×2 grid** CG stencil
	16	**4×4 grid**	**4×4 grid**	**4×4 grid**
	32	**8×4 grid**	**8×4 grid**	**8×4 grid**
	64	**8×8 grid**	**8×8 grid**	**8×8 grid**
	128	**16×8 grid**	**16×8 grid**	**16×8 grid**
CG	8	**4×2 grid** CG stencil	**4×2 grid** CG stencil	**4×2 grid** CG stencil
	16	**CG stencil** **8×2 grid**	**CG stencil**	**CG stencil**
	32	**CG stencil** **8×2×2 grid**	**CG stencil**	**CG stencil**
	64	**CG stencil** **16×2×2 grid**	**CG stencil**	**CG stencil**
	128	**CG stencil** **16×2×2×2 grid**	**CG stencil**	**CG stencil**
MG	8	**2×2×2 grid** 4×2 torus	**2×2×2 grid** 4×2 torus	**2×2×2 grid** 4×2 torus
	16	**2×2×2×2 grid** 4×2×2 torus 4×4 torus	**2×2×2×2 grid** 4×2×2 torus 4×4 torus	**2×2×2×2 grid** 4×2×2 torus 4×4 torus
	32	**2×2×2×2×2 grid** 4×2×2×2 torus 4×4×2 torus	**2×2×2×2×2 grid** 4×2×2×2 torus 4×4×2 torus	**2×2×2×2×2 grid** 4×2×2×2 torus 4×4×2 torus
	64	**2×2×2×2×2×2 grid** 4×2×2×2×2 torus 4×4×2×2 torus 4×4×4 torus **8×8 6-p stencil**	**2×2×2×2×2×2 grid** 4×2×2×2×2 torus 4×4×2×2 torus 4×4×4 torus	**2×2×2×2×2×2 grid** 4×2×2×2×2 torus 4×4×2×2 torus 4×4×4 torus
	128	**8×2×2×2×2 torus** 8×4×2×2 torus 8×4×4 torus	**8×2×2×2×2 torus** 8×4×2×2 torus 8×4×4 torus	**8×2×2×2×2 torus** 8×4×2×2 torus 8×4×4 torus

TABLE V

IDENTIFICATION OF COMMUNICATION TOPOLOGIES. EACH UNIQUE TOPOLOGY IS IN BOLDFACE. TOPOLOGIES NOT IN BOLDFACE ARE ISOMORPHIC TO THE BOLDFACE TOPOLOGY ABOVE THEM.

was discarded.

B. Performance

1) Processing time for NAS benchmarks: The sizes of the traces for the NAS benchmark programs and the total time to logicalize them is listed in Table VI. A trace record corresponds to a traced MPI call. Since tracing employed is fairly lightweight, trace sizes are modest and the tracing overhead within 1% of the execution time for all the benchmark programs. The longest trace was just under 200K records and around 10 MBytes per process for the 128 process LU benchmark. The processing times are measured on an ordinary PC - a 1.86 GHz Pentium M with 1GB RAM. Processing times are fairly low with a maximum of 134 seconds for the aforementioned LU benchmark. The processing time tracked the total number of lines in the trace almost linearly as plotted in Figure 3.

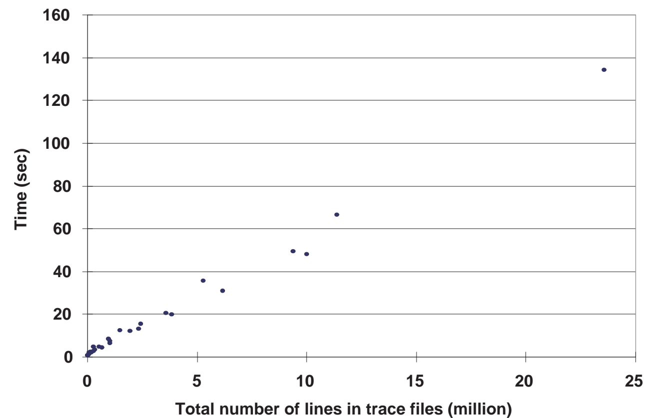

Fig. 3. Total length of all process communication traces and the logicalization time

The processing time is dominated by the construction of the

Name	4 processes		8/9 processes		16 processes		32/36 processes		64 processes		121/128 processes	
	Trace Length Records (Size)	Time (secs)	Trace Length Records (Size)	Time (secs)	Trace Length Records (Size)	Time (secs)	Trace Length Records (Size)	Time (secs)	Trace Length Records (Size)	Time (secs)	Trace Length Records (Size)	Time (secs)
BT	2278 (90 KB)	0.63	12282 (490 KB)	1.73	17106 (731 KB)	2.64	26754 (1081 KB)	8.35	36402 (1459 KB)	13.19	50874 (2106 KB)	30.76
SP	12452 (533 KB)	1.39	19670 (824 KB)	2.09	26888 (1147 KB)	4.14	41324 (17543 KB)	12.55	55760 (2365 KB)	20.34	77414 (3365 KB)	49.16
CG	5042 (186 KB)	0.91	41954 (1599 KB)	3.31	41954 (1667 KB)	4.52	59964 (2376 KB)	11.94	59964 (2376 KB)	19.89	77978 (3224 KB)	47.89
LU	2338 (95 KB)	0.69	152294 (6661 KB)	6.43	203048 (9185 KB)	15.39	203048 (9186 KB)	35.46	203048 (9088 KB)	66.28	203048 (9433 KB)	134.30
MG	1433 (57 KB)	0.73	8867 (403 KB)	1.98	8909 (373 KB)	2.48	8951 (374 KB)	4.56	8953 (373 KB)	4.75	9035 (386 KB)	7.33

TABLE VI
TRACE SIZE (PER PROCESS) AND PROCESSING TIME FOR LOGICALIZATION

communication matrix as that is the only step that analyzes the trace from each process, even though the actual processing on each trace entry is minimal. The only tests in the framework that are potentially computationally expensive are the graph spectrum test and the graph isomorphism test. The processing time for them is plotted in Figure 4. We observe that graph spectrum testing time is under one second for every case, and graph isomorphism testing time cannot be observed on this graph as it is in millisecond range in every case. An important reason for the low overhead of graph spectrum and graph isomorphism tests is that they had to be applied on very few candidate topologies (often 1 or 2) as simple tests discussed earlier were extremely effective in reducing the number of candidate topologies. The simple tests also executed in negligible time.

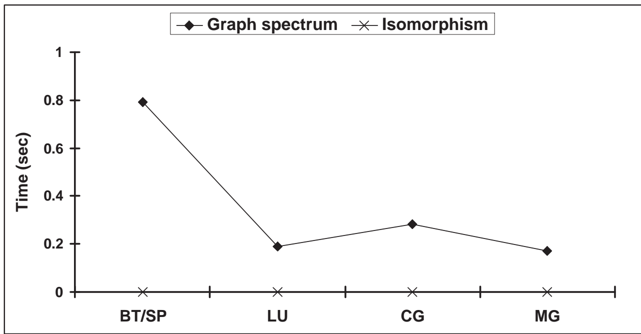

Fig. 4. Graph spectrum and graph isomorphism processing time for 121/128 processes for different NAS benchmarks

2) Scalability analysis with synthetic data: The results noted above for the NAS benchmarks are limited to 128 processes, and virtually all the processes were numbered “nicely” along the axes for grid/torus topologies. In this section, we analyze potential performance issues as we scale to larger graphs and encounter cases with irregular process numbering. We have already noted that the the matrix construction time is linear, and hence predictable, and simple tests are extremely fast. Potential performance issues may be encountered in 1) Graph spectrum tests with $O(N^3)$ complexity and 2) Graph isomorphism tests which are based on a non-polynomial heuristic. We investigate the performance of these tests further with synthetic data.

The results for graph spectrum computation on an ordered and an unordered 2D grid, and an ordered and an unordered 6-point stencil pattern on a 2D grid, are plotted in Figure 5. In the unordered case, 2/3rd of graph nodes were arbitrarily renumbered after starting with a row major order. The computation times are within 70 seconds for up to 1000 nodes or processes, but increase rapidly from a number of 500 to 1000. Hence this test may not be sufficiently efficient for larger scenarios with 1000s of processes.

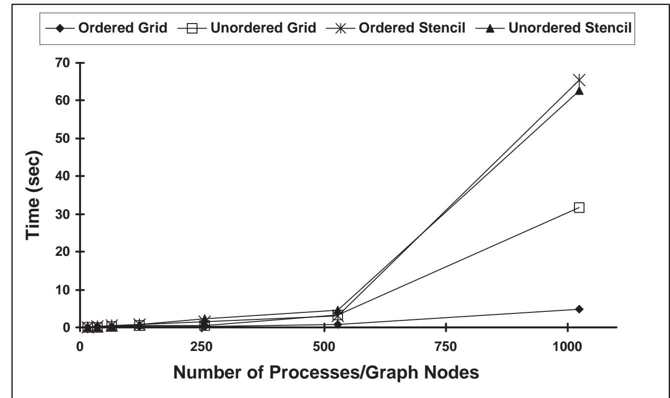

Fig. 5. Performance of the graph spectrum test for isomorphism for different topologies with and without randomness in numbering of processes

We investigate the performance of the VF2 graph matching algorithm further with synthetic ordered and unordered grids, tori, stencils, and binary trees. In the ordered cases, the nodes of the grids, tori, and stencils are numbered in row major order. The binary tree nodes are numbered in level order. The unordered (or randomized) cases have a percent of the nodes arbitrarily renumbered from the ordered numbering. For illustration, say the node numbered 12, which would normally be in the second row in an 8x8 grid, is renumbered, say, as node 47, and vice versa.

The effect of process numbering for 2D grids is shown in Figure 6. The figure shows the processing time for ordered numbering and randomness degrees of 25, 50, 75, and 99 (maximum). The processing time does increase with the degree of randomness, but is within a few seconds even for maximum

randomness up to a graph size around 16K nodes.

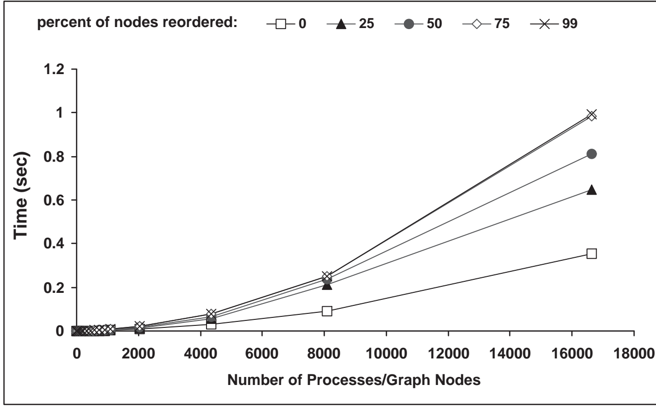

Fig. 6. Performance of VF2 graph matching algorithm for 2D grid topology with varying degree of randomness in numbering of processes

The performance results of employing the VF2 graph matching algorithm on various unordered topologies with maximum randomness are plotted in Figure 7. The processing times are very low for almost all topologies up to 1000 nodes and modest for around 16,000 nodes. A key exception is the 6-point stencil on a 2D torus, for which the processing for over 1000 nodes could not be completed for several hours and the experiment was abandoned, so no data is shown in the graph. However, the processing time for the 6-point stencil on a 2D torus *without* any randomness was dramatically lower: less than a second for up to 16,000 nodes. Also note in Figure 7 that the processing time for 6-point stencil on a 2D grid (instead of torus) is within a second. We speculate that the reasons are related to the type of heuristics employed for graph isomorphism. This represents a potential limitation, but is a problem only for very irregular numbering of nodes. We believe that it is important to allow all possible numbering of nodes, as we cannot predict what numbering an application may follow. However, we do not expect any application to follow completely random numbering that was used in this stress test. Overall, we conclude that the methodology is effective for 1000s of nodes for any numbering of processes that is likely to be encountered in practice. A detailed analysis of the performance of different graph matching algorithms and available packages is presented in [17].

VI. Discussion and future work

The performance of the topology identification procedure generally scales well to thousands of processes. One exception is that the execution time of polynomial eigenvalue computation for the graph spectrum test increases rapidly beyond 1000 processes. Also, the marginal value of the graph spectrum test is limited as the number of candidate patterns after simple tests is very low and can be directly tested for isomorphism. Hence we conclude that the graph spectrum test can be removed from the procedure.

Another scalability concern is due to the fact that our implementation is based on storing a trace file for every

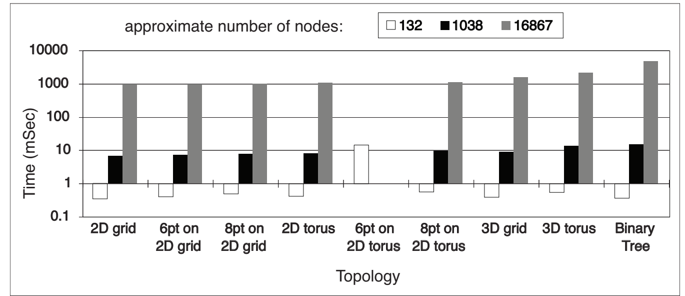

Fig. 7. Performance of VF2 graph matching algorithm for different topologies with maximum randomness

process, and therefore, the storage space required can be very large. This was not encountered in our experiments since the scale was relatively small, and also because trace information needed for our objectives is low. The situation can be mitigated by simple lossless per trace compression during recording. In particular, storing each unique trace record in a hash table and replacing every instance of it with a fixed size token leads to significant compression, around a factor of 100 for our traces. Elements of this approach are similar to [10] as discussed in Section III but we propose it as a preprocessing step rather than the final compression step. This component is part of our framework but not implemented in an on-the-fly fashion at the present time.

Our topology detection procedure implicitly assumes a single dominant communication pattern for the duration of the application. While this is often the case, an application can also have multiple phases where the computation and communication pattern changes across phases. Separating the phases is the subject of other research, such as [8]. Our framework can be applied piecewise if the phases are separated in the trace. Applications can also exhibit multiple concurrent patterns. e.g., a grid and a tree. Addressing such scenarios requires solving the *subgraph isomorphism* problem which is known to be NP-complete and more challenging to solve heuristically. We are currently investigating heuristics to solve the problem in practical scenarios.

VII. Conclusions

Application communication traces are at the core of performance analysis and performance modeling of communicating parallel programs. However, when execution is on a large number of nodes, the size of the traces is a hindrance to their effective usage. Further it is difficult to meaningfully analyze hundreds of traces, each representing execution on one node of a system. This paper presents a framework to automatically construct a single logical trace that is representative of the overall parallel execution when the communication pattern is a regular stencil. The approach is based on identifying the communication topology of the application and converting all point-to-point communication calls between physical processes to logical calls representing the global communication pattern. The methodology is independent of the numbering of processes in the system. The key contribution is an algorithmic

framework to identify the global communication topology from distributed message exchange data that is effective and efficient.

Results presented show that the procedure was successful and efficient for the NAS benchmark suite. Detailed analysis of the performance data shows that the execution time of trace logicalization is likely to be modest for realistic scenarios. The key steps of the framework were analyzed in detail. In particular, for the purpose of identifying the communication topology of an application, a suite of simple tests were found to be very effective, the performance of the graph isomorphism test was adequate for practical scenarios, while the graph spectrum test was found to have limited value and scalability. The paper lays the foundation for a new approach to summarization and reduction of message passing traces that is powerful and likely to be enhanced by future research.

VIII. Acknowledgments

Support for this work was provided by the National Science Foundation under Award No. SCI-0453498, CNS-0410797 and ACI-0234328. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

We would also like to thank Ravi Prithivathi for his contribution to this work.

References

[1] H. Brunst, H.-C. Hoppe, W. E. Nagel, and M. Winkler. Performance optimization for large scale computing: The scalable VAMPIR approach. In *International Conference on Computational Science (2)*, pages 751–760, 2001.

[2] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the VF graph matching algorithm. In *Proc. of the 10th ICIAP*, volume 2, pages 1038–1041. IEEE Computer Society Press, 1999.

[3] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, New Orleans, LA, September 2003.

[4] P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *The 3rd IAPR-TC15 Workshop on Graph-based Representations*, 2001.

[5] V. Hernandez, J. E. Roman, A. Tomas, and V. Vidal. A survey of software for sparse eigenvalue problems. Technical Report STR-6, Universidad Politécnica de Valencia, 2006. Available at http://www.grycap.upv.es/slepc.

[6] S. Huband and C. McDonald. Debugging parallel programs using incomplete information. In *1st IEEE Computer Society International Workshop on Cluster Computing*, pages 278–286, 1999.

[7] S. Huband and C. McDonald. A preliminary topological debugger for MPI programs. In *1st International Symposium on Cluster Computing and the Grid (CCGRID 2001)*, page p. 422, 2001.

[8] J.Gonzalez, J. Gimenez, and J. Labarta. Automatic detection of parallel applications computation phases. In *Proceedings of 23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

[9] D. Kerbyson and K. Barker. Automatic identification of application communication patterns via templates. In *18th International Conference on Parallel and Distributed Computing Systems*, Las Vegas, NV, September 2005.

[10] M. Noeth, F. Mueller, M. Schulz, and B. de Supinski. Scalable compression and replay of communication traces in massively parallel environments. In *21th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, April 2007.

[11] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *International Symposium on Computer Architecture (ISCA)*, June 2003.

[12] J. Singler. Graph isomorphism implementation in LEDA 5.1. http://www.algorithmic-solutions.de/bilder/graph_iso.pdf.

[13] J. Subhlok and Q. Xu. Automatic construction of coordinated performance skeletons. In *NGS 2008: NSF Next Generation Software Program Workshop (Appears in Proceedings of IPDPS 2008)*, Miami, FL, April 2008.

[14] T. Tabe and Q. Stout. The use of the MPI communication library in the NAS Parallel Benchmark. Technical Report CSE-TR-386-99, University of Michigan, Nov 1999.

[15] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *2001 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'01)*, pages 123–132, 2001.

[16] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *J. Parallel Distrib. Comput.*, 63(9):853–865, 2003.

[17] S. Voss and J. Subhlok. Performance of general graph isomorphism algorithms. Technical Report UH-CS-09-07, University of Houston, Aug 2009.

[18] C. E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk, and W. Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.

[19] Q. Xu and J. Subhlok. Construction and evaluation of coordinated performance skeletons. In *The 15th annual IEEE International Conference on High Performance Computing (HiPC 2008)*, Bangalore, India, Dec 2008.

[20] Q. Xu and J. Subhlok. Efficient discovery of loop nests in communication traces of parallel programs. Technical Report UH-CS-08-08, University of Houston, May 2008.

[21] B. Yorgey. Generating multiset partitions. *The Monad.Reader*, (8):5–20, Sept 2007.