``Logicalization'' of MPI Communication Traces

Qiang Xu, Ravi Prithivathi, Jaspal Subhlok, Rong Zheng

Computer Science Department
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

Technical Report Number UH-CS-08-07
May 26, 2008

Keywords: Trace compression, Graphic isomorphism, Performance analysis and modeling

Abstract

Communication traces are integral to performance 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 a single parallel execution of an SPMD MPI message passing program into a single logical program trace. The approach first identifies the communication topology of the application from the application communication matrix computed from the traces. Topology identification is based on the application communication structure and independent of the way ranks (or numbers) are assigned to processes. Once the application topology is identified, point-to-point communication between processes is converted into logical communication that represents similar communication across all processes executing the application. This logicalization framework has been implemented and the results with NAS benchmarks show that it is efficient and effective. The procedure is part of a system to automatically generate performance skeletons that are short running programs mimicing core computation and communication characteristics of an application.

"Logicalization" of MPI Communication Traces

Qiang Xu Ravi Prithivathi Jaspal Subhlok Rong Zheng
University of Houston, Department of Computer Science, Houston, TX 77204

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. The approach first computes the application communication matrix from process traces. Topology identification, which includes graph spectrum analysis and graph isomorphism test, is based on the underlying communication structure and independent of the way ranks (or numbers) are assigned to processes. Once the application topology is identified, message exchanges between physical processes are converted into logical communication that represents similar message exchanges across all processes executing the application. This logicalization framework has been implemented and the performance is analyzed. The results with NAS benchmarks show that it is efficient and effective.

Index Terms
Trace Compression, Graph isomorphism, Performance analysis and modeling

1 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, as high performance scientific applications are generally SPMD programs, in most cases, the traces for different processes are similar to each other and the communication between processes is associated with a well defined global communication pattern. A study of DoD and DoE HPC codes at Los Alamos National Labs [7] and analysis of NAS benchmarks [14] shows that an overwhelming majority of these codes have a single low degree stencil as the dominant communication pattern. These characteristics expose the possibility of combining all processor traces into a single *logical program trace* that represents the aggregate execution of the program - in the same way as an SPMD program represents a family of processes that typically execute on different nodes. This paper presents a framework for automatic construction of a logical program trace from the collection of physical process traces of an execution of a message passing parallel application.

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,...)*
...	*...*	*...*	*...*

The above physical trace can be summarized as the following logical trace:

Program

...

snd(P_R*,...)*

rcv(P_L*,...)*

...

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

The basic goal of the research presented in this paper is to automatically logicalize traces of message passing MPI parallel programs. Beside reducing the trace size by a factor equal to the number of processes, the logical program trace captures the parallel structure of the application. Note that this logicalization is orthogonal to single trace compression, which is based on discovering temporal repeating patterns in traces. For instance, if the trace for **Process 0** above consisted of the sequence:

"... snd(P1,...) rcv(P3,...) snd(P1,...) rcv(P3,...) snd(P1,...) rcv(P3,...) snd(P1,...) rcv(P3,...) ..."

then single trace compression can identify the repeating sequences and reduce the trace to the following representation:

".... [snd(P1,...) rcv(P7,...)] ^{4}*....."*

Single trace compression can be applied to individual process traces as illustrated above, as well as to the logical trace of an application. We will refer to single trace compression simply as *trace compression* in the rest of this paper, as distinct from logicalization. This paper focuses on trace logicalization, although trace compression and logicalization are likely to be employed together.

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 during execution. This matrix is then analyzed to determine the application level communication topology. Once this global topology is determined, a representative process trace is analyzed in detail and transformed into a logical program trace where all message sends and receives are to/from a logical neighbor in terms of a logical communication topology (e.g a torus or a grid) instead of a physical process rank.

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 with 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 with respect to the underlying 2D grid pattern as indicated in Figure 1(b), the communication graph with process nodes laid out in row major order

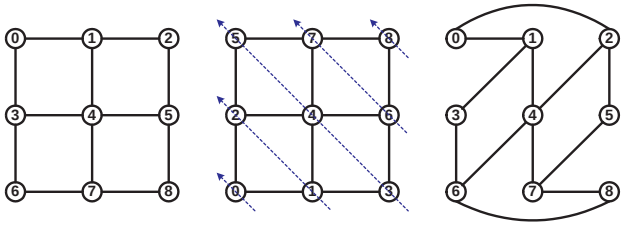

Figure 1. 2D grid topology with row major and diagonal process numbering

would appear as Figure 1(c). Clearly, the underlying 2D grid 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 relatively 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 [7].

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 even for a fixed number of nodes). A naive method would require solving the graph isomorphism problem for each instance of each candidate topology, which is computationally infeasible.

Our approach to identifying the communication topology of an application represented by a communication matrix 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 between the application communication matrix (or graph) and the invariants of a topology: *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, often just one. However, it is still not proved if any of the remaining candidate topologies is actually the application communication topology.

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 [10]. Also, the size of the problem to be solved is relatively modest as the number of processes is likely to be at most in 100s. We employ the VF2 graph

matching algorithm from *VFlib2* library [2, 3] to test for isomorphism between the graphs represented by the application communication matrix and each remaining candidate reference topology to establish the final application communication topology.

The tracing required for logicalization procedure is very low overhead in computation time and volume as only high level message passing calls are recorded. The analysis required for each process trace is minimal - only the collection of gross communication data, such as the number of messages and bytes exchanged. Detailed processing is limited to a single representative process trace that is transformed to a logical program trace. The paper describes the design and implementation of the logicalization framework. Experimental results are discussed, along with the limitations and possible extensions.

2 Motivation and context: Trace compression for performance skeletons

The results developed in this paper have broad applicability in performance analysis and modeling. In this section we discuss the specific context and usage of this research, which is construction of application performance skeletons for performance prediction. A performance skeleton is a short running program that recreates the dominant computation and communication behavior of the original application execution. 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 it represents in the 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 procedure for skeleton construction and the effectiveness of skeletons for performance prediction are discussed in [12, 11, 15, 19]. The research presented in this paper is part of an improved scalable and efficient skeleton construction procedure, which is sketched in Figure 2 and detailed in [19].

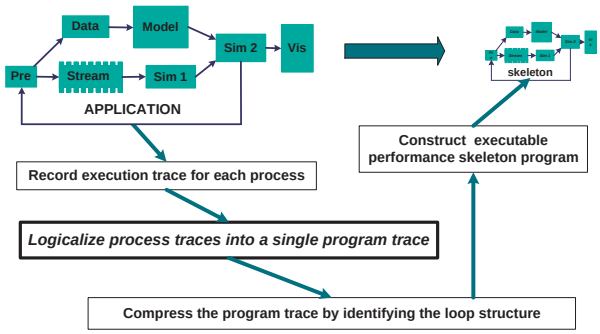

Figure 2. Skeleton construction

The highlighted logicalization step in Figure 2 is the focus of this paper and discussed in detail in the following sections. Table 1 presents summary results from the *combined* 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 1 is the ratio of the size of full logical (or single process) 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.

Table 1. Compression results for NAS programs. Trace length in number of records.

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

3 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 [16]. The idea of communication/adjacency matrix for trace analysis for parallel programs was introduced in [5, 6]. 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. In contrast to collecting summary information from a trace, the goal of the work presented in this paper is to identify program wide communication topology and combine a set of per-process traces into a single program trace to streamline and speedup subsequent trace processing.

Perhaps the work closest to this paper is the scalable trace compression presented in Noeth et. al. [8]. They perform task (or process) level compression on the fly, followed by consolidation of compressed traces that they refer to as inter-node compression. The central difference is that we perform logicalization (or inter-node compression) first on process traces, and subsequently perform conventional compression only on a single logical program trace. In our view, this offers two major advantages. First, process traces are analyzed only to determine the type and size of message exchanges, not to compress them. This reduces the overhead considerably. Second, consolidation of a large number of compressed traces, which is challenging, is not needed. The work presented in [8] performs process trace compression on the fly, hence full trace need not be recorded. We will revisit the tradeoffs involved in Section 6.

We have borrowed part of our pattern identification methodology from Kerbyson et.al. [7] but our algorithm for identifying communication patterns represents a significant improvement. In [7], the authors first develop the real unweighted point-to-point communication matrix from an application execution and then measure the degree of *match* with a set of predefined communication template matrices representing regularly occurring communication patterns in scientific applications. Their method assumes that the nodes are numbered in a "reasonable" way, e.g., along the rows or columns for a 2-dimensional 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 [7, 13] 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.

A compiler assisted approach to identification of MPI patterns is presented in [9]. Our work is most relevant when the application communication follows a relatively static pattern and this is known to be the case for most scientific applications [17, 7, 14]. Finally, this work is in the context of development and usage of performance skeletons that are presented in [11, 15, 19].

4 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 application 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 is that there is a dominant regular communication pattern in the trace being processed, else no topology is identified. Extension to traces that have multiple phases of different patterns, or multiple concurrent patterns, are discussed in Section 6.

4.1 Generation of application communication matrix

For generation of a physical trace, the 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.

Samples from beginning and end of the trace recorded on Node 0 for NAS BT Class S benchmark running on 4 Nodes are illustrated in Table 2. The complete trace consists of 2278 MPI calls. The table shows the raw trace along with a listing of the calls with key parameters. The first call is a broadcast of a single integer rooted at rank 0. The 4th call is an integer receive of 360 doubles from the process with rank 1. The trace displayed here only lists the communication calls, which are interleaved with computation sections.

The next step is the generation of a *full application communication matrix*. The matrix records the total data transferred between each pair of processes involved in the execution. In most SPMD applications, some entries in this matrix represent a large amount of data transfer while many entries are zero, implying that there was no communication between the corresponding pair of processes. The full communication matrix is then converted to a *binary application communication matrix*, where communicating pairs of processes are represented by 1 and non-communicating pairs of processes are represented by 0. As a simple example, the full communication matrix and the binary communication matrix for 8 process NAS MG benchmark are shown in Table 3.

Table 2. A sample execution trace

#Generating Logfile	
Node=0 #939220507ss#0#939220507	
1. 2#1#3220724724#1#28#0#134#0#939220509#939220509	[*MPI_Bcast(...1, MPI_INT, 0,...)*]
2. 2#2#136373224#1#27#0#134#0#939220509#939220509	[*MPI_Bcast(...1, MPI_DOUBLE, 0,...)*]
3. 2#3#135838396#3#28#0#134#0#939220509#939220509	[*MPI_Bcast(...3, MPI_INT, 0,...)*]
4. 7#1#135789088#360#27#1#3000#138#153016848#0#939220509#939220509	[*MPI_Irecv(... 1, MPI_DOUBLE, 360, ...)*]
5. 7#2#135786208#360#27#1#2000#138#153017012#0#939220509#939220509	[*MPI_Irecv(... 1, MPI_DOUBLE, 360, ...)*]
6. 7#3#135794848#360#27#2#5000#138#153017176#0#939220509#939220509	[*MPI_Irecv(... 2, MPI_DOUBLE, 360, ...)*]
7. 7#4#135791968#360#27#2#4000#138#153017340#0#939220509#939220509	[*MPI_Irecv(... 2, MPI_DOUBLE, 360, ...)*]
8. 7#5#135800608#360#27#3#6000#138#153017504#0#939220509#939220509	[*MPI_Irecv(... 3, MPI_DOUBLE, 360, ...)*]
9. 7#6#135797728#360#27#3#7000#138#153017668#0#939220509#939220509	[*MPI_Irecv(... 3, MPI_DOUBLE, 360, ...)*]
10. 9#1#135812616#360#27#1#2000#138#153002824#0#939220509#939220509	[*MPI_Isend(... 1, MPI_DOUBLE, 360, ...)*]
11. 9#2#135809736#360#27#1#3000#138#153002964#0#939220509#939220509	[*MPI_Isend(... 1, MPI_DOUBLE, 360, ...)*]
12. 9#3#135818376#360#27#2#4000#138#153003104#0#939220509#939220509	[*MPI_Isend(... 2, MPI_DOUBLE, 360, ...)*]
13. 9#4#135815496#360#27#2#5000#138#153003244#0#939220509#939220509	[*MPI_Isend(... 2, MPI_DOUBLE, 360, ...)*]
14. 9#5#135824136#360#27#3#7000#138#153003384#0#939220509#939220509	[*MPI_Isend(... 3, MPI_DOUBLE, 360, ...)*]
15. 9#6#135821256#360#27#3#6000#138#153003524#0#939220509#939220509	[*MPI_Isend(... 3, MPI_DOUBLE, 360, ...)*]
16. 22#1#12#153016848#153017012#153017176#153017340#153017504 #153017668#153002824#153002964#153003104#153003244#153003384 #153003524#0#939220509#939220513	[*MPI_Waitall(...)*]
17. 9#7#135786208#1470#27#1#3000#136#153003524#0#939220513#939220513	[*MPI_Isend(... 1, MPI_DOUBLE, 1470, ...)*]
18. 7#7#135809736#1470#27#1#3003#136#153017668#0#939220513#939220513	[*MPI_Irecv(... 1, MPI_DOUBLE, 1470,...)*]
19. 21#1#153003524#0##939220513#939220513	[*MPI_Wait(...)*]
20. 21#2#153017668#0##939220513#939220513	[*MPI_Wait(...)*]
....	....
2277. 3#1#3220724688#3220724696#1#27#100#0#134#0#939220642#939220642	[*MPI_Reduce(...1, MPI_DOUBLE, MPI_MAX, ...)*]
2278. 1#2#91#0#939220642#939220642	[*MPI_Barrier*]
#Finished writing logfile for node=0#939220642#939220646	

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 not considered central to construction of performance skeletons and similar relative performance modeling applications. A filtering step removes matrix entries corresponding to such very low level communication based on a threshold. A heuristic threshold of 5% was used in our experiments. The best possible threshold value can only be determined by far more extensive experimentaions than was feasible. However, our qualitative observation is that that the main pattern is generally dominating, hence most reasonable threshold values would suffice.

Table 3. Full Application Communication Matrix (traffic in KBytes/sec) and Binary Application Communication Matrix for 8 process MG benchmark

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

This step was not relevant for our example codes, except that it was critical for discovering the main communication pattern for the MG benchmark for 16 and higher numbers of processes. The full communication matrix and the binary communication matrix after filtering for 16-process MG benchmark are shown in Table 4. The volume of the communication not associated with the main communication pattern was around 0.5% compared to the main communication pattern. An accuracy measure is introduced in the framework that quantifies the extent of low volume communication that is lost in the logicalization process.

Table 4. 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.

	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
P0	0	87.09	89.09	0	91.19	0	0	0	0	0	0	0	91.19	0	0	0
P1	87.09	0	0	89.09	0	91.19	0	0	0	0	0	0	0	91.19	0	0
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
P6	0	0	91.21	0	89.10	0	0	87.09	0	0	91.23	0	0	0	***0.46***	0
P7	0	0	0	91.21	0	89.10	87.09	0	0	0	0	91.23	0	0	0	***0.46***
P8	0	0	0	0	91.19	0	0	0	0	87.09	89.09	0	91.19	0	0	0
P9	0	0	0	0	0	91.19	0	0	87.09	0	0	89.09	0	91.19	0	0
P10	0	0	0	0	0	0	91.19	0	89.09	0	0	87.09	0	0	91.19	0
P11	0	0	0	0	0	0	0	91.19	8	89.09	87.09	0	0	0	0	91.19
P12	91.23	0	0	0	***0.46***	0	0	0	91.21	0	0	0	0	87.10	89.10	0
P13	0	91.23	0	0	0	***0.46***	0	0	0	91.21	0	0	87.09	0	0	89.10
P14	0	0	91.23	0	0	0	***0.46***	0	0	0	91.21	0	89.10	0	0	87.09
P15	0	0	0	91.23	0	0	0	***0.46***	0	0	0	91.21	0	89.10	87.10	0

⇓

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

4.2 Identification of application communication topology

The procedure in the previous section yields a binary application communication matrix. In this section we present a procedure to determine if this application communication matrix represents an instance of a topology such as a grid, torus, tree or another stencil that is part of a reference communication ma-

trix library. This is a challenging problem for two reasons. First, reference patterns in the library are normally stored as abstract types. Each type of pattern can have 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 a graph correspond to different permutations of the rows and columns of its adjacency matrix. Naive comparison of the adjacency matrix corresponding to an application communication matrix with a reference graph corresponding to an identical topology but different ordering of vertices will lead to a mismatch. Therefore efficient algorithms need to be devised to eliminate the patterns that can be easily proved to *not* be the topology corresponding to an application communication matrix, and to verify if an application communication matrix indeed corresponds to a reference pattern independent of vertex numbering.

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 proven to be NP-complete. (On the other hand, the subgraph isomorphism problem, i.e., finding if a graph is isomorphic to a subgraph of another graph, is known to be NP-complete.) The problem of identifying the topology from an application communication matrix is essentially one of determining if an instance of a reference pattern exists whose graph is isomorphic to the application communication graph represented by the matrix.

The problem of identifying the communication topology can be divided into two sub-problems, namely, i) determining candidate patterns in the reference library and ii) determining an exact matching reference topology. Note that determining whether the application matrix *contains* a set of reference patterns is equivalent to the harder subgraph isomorphism problem that is left to future work.

Identification of candidate patterns

Although there is no general polynomial solution to graph isomorphism, relatively efficient practical solutions exist. However, enumerating through all the reference patterns and checking for isomorphism is clearly infeasible in practice. 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.

It is known that the above parameters are invariants with respect to permutation of the adjacency matrix. 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 [4]. 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.

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 $n_0 \cdot n_1 \cdots n_m$ [20][1]. Clearly, these are the only patterns that can possibly match. For each such product form as well as other patterns (e.g. binary trees), we compute the edge count of reference patterns. Through elimination, we have the subset of reference graphs as $S_1 = \{G_1^1, G_2^1, \ldots, G_{l_1}^1\}$, which have the same number of edges as G.

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 $S_2 = \{G_1^2, G_2^2, \ldots, G_{l_2}^2\}$.

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, it is likely that we are left with a single candidate topology. However, one cannot yet conclude that it is the matched topology. The invariants employed can eliminate a pattern from consideration but do not guarantee a match.

Exact matching to establish topology

We apply graph isomorphism algorithms 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, practically efficient 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 [10].

The VF2 algorithm takes a bottom-up approach [3]. 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. In addition to a binary value indicating whether a match is found, the VF2 algorithm also outputs two arrays containing the labels of nodes paired by the matching algorithm.

[1] Graphs of lower dimension i can be represented by setting $i = i + 1 = \cdots = 1$

4.3 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 and matches a known reference pattern.

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, in a linear array pattern, all processes *except* the endpoints of the array are maximal communication processes. The reason is that interior processes communicate with 2 neighbors, while the endpoints communicate with just one neighbor. Similarly, 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.

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 logical communication trace from physical traces and a known application communication topology D with maximum communication degree k.

1. Identify any 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. The trace is interpreted in connection with the corresponding communication topology. Some of the communication operations are relevant only for some processes in the pattern for asymmetrical patterns. If the logical trace is used to generate a physical trace for a given process rank, actual communication is generated subject to the condition that the corresponding neighbor exists in the topology (e.g., the $North$ neighbor does not exist for the top row of processes in a 2D grid, and hence that communication is not valid). Finally, all collective operations are unchanged from the physical trace to the logical trace - collective operations are already global logical operations across the executing processes and no change is needed. There is an implicit assumption that they apply to *MPI COMMWORLD* communicator and we discuss handling of multiple communicators in the discussion in this paper.

A section of physical and corresponding logical traces for the 16 process BT benchmark are shown in Table 5. Note that the directions are labeled as North, South, etc. for simplicity and will, in fact, be indices in a general topology matrix.

Communication outside the main topology Any communication operation that references a process rank that is not in the established topology is not included in the logical trace. The reason is that

Table 5. Sections of a sample physical trace and corresponding logical trace for BT benchmark

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* */
....	....

corresponding operations do not exist across the parallel application. In fact, whenever communication filtering discussed in section 3 is applied, such local communication is expected in the physical trace. In our implementation we record the fraction of communication that falls in this category as that represents an inaccuracy in this approach.

5 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.)

5.1 Identification of communication topology

A full application communication matrix was generated for each program and then converted to a binary communication matrix based on the discussion in Section 3. Filtering discussed in that section was necessary to generate a binary matrix only for MG benchmark for sizes higher than 16 processes

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 them any further.

The matching procedure was then applied to the remaining 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 for size 16 is illustrated in Figure 3. Subsequently, this pattern was manually added to the library. The results presented include this addition.

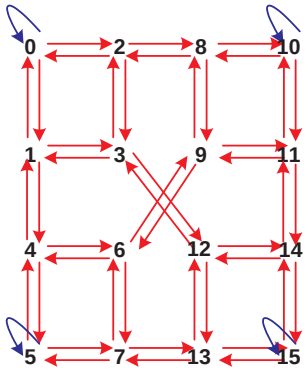

Figure 3. CG communication pattern

The matching procedure can be considered to consist of 3 distinct steps based on the description in Section 4.2.

1. **Simple Tests:** Finding all possible sizes of grid/tori based on prime factors of the number of processes N, and then matching the number of edges and the degree ordered sequence of nodes.
2. **Graph Spectrum Test:** Based on computing eigenvalues.
3. **Isomorphism Test:** Applies graph isomorphism to establish a topology.

Table 7 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 7 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 7:

- All benchmarks in our test suite were matched correctly, although CG was matched only when a custom stencil was added. SP and BT are 6 point stencils while LU, CG, and MG are grids or 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 very effective, and in fact, eliminated all candidates except for the final correct topology.

5.2 Generation of logical traces

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. In the case of MG, some processes performed communication that could not be mapped to the application topology and was discarded. However, the fraction of such calls was low. As a typical example, for the case of MG benchmark running on 32 nodes, only 5.6% percent of the communication calls carrying 2.7% percent of the traffic were not included in the logical trace. Hence the accuracy metric we employ shows perfect accuracy for all benchmarks except for a modest inaccuracy for the MG benchmark.

These logical traces were compressed by identifying repeating loops and deployed to construct performance skeletons [19]. The summary results from combined and compression, along with the procedure for skeleton construction, are outlined in Section 2. However, a detailed discussion is beyond the scope of this paper.

5.3 Performance

Processing time for NAS benchmarks

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 6. Trace Size (per process) and processing time for logicalization

The sizes of the traces for the NAS benchmark programs and the total time to logicalize them is listed in Table 6. A trace record corresponds to a traced MPI call. Since tracing is fairly lightweight, trace sizes are modest and the tracing overhead is low; within 1% of the execution time for all the benchmark programs. The longest trace was around 200K records and around 10 MBytes per process for 128 process LU benchmark. The processing times are computed 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 above mentioned

LU benchmark. The processing time tracked the total number of lines in the trace almost linearly as plotted in Figure 4.

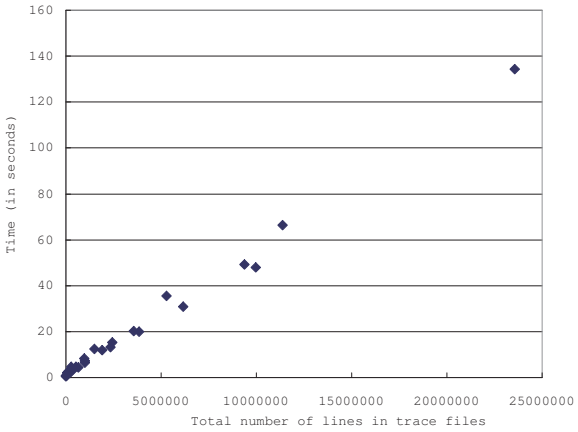

Figure 4. Trace length and processing time for logicalization

The processing time is dominated by the construction of the 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 5. 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.

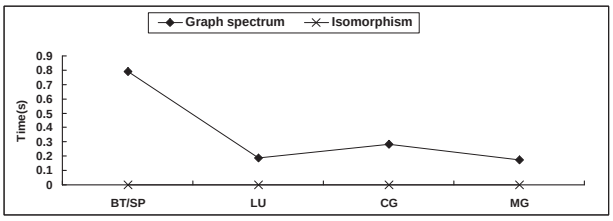

Figure 5. Graph spectrum and isomorphism processing time for 121/128 process NAS benchmarks

Scalability analysis with synthetic data

The results noted above for the NAS benchmarks are very encouraging. However, this is a small test suite, the number of processes was limited to 128, and virtually all the processes were numbered along the axes for grid/torus topologies. In this section, we analyze potential performance issues as we scale to over 1000 nodes and encounter cases with irregular numbering of processes.

We have already noted that matrix construction time is linear, and hence predictable, and the simple tests are extremely fast. Potential performance issues may be encountered in 1) Graph spectrum test

that involves computation of eigenvalues with $O(N^3)$ complexity and 2) Graph isomorphism test which is based on a non polynomial heuristic. We investigate the performance of these tests further with the following synthetic data:

1. *Ordered and Unordered grids:* Matching an application matrix representing a 2D grid against a corresponding template topology for varying sizes. In the ordered case the grid nodes are numbered in the normal row major order. In the unordered case, 2/3rds of the graph nodes were arbitrarily renumbered after starting with a row major order, leading to a partially randomized ordering.

2. *Ordered and Unordered stencils:* Similar to the above grid case except matching is for a 6 point stencil pattern on a 2D grid.

The results for graph spectrum computation are plotted in Figure 6. The computation times are within 70 seconds for up to 1000 nodes or processes, hence it appears that this test is sufficiently efficient for larger practical scenarios.

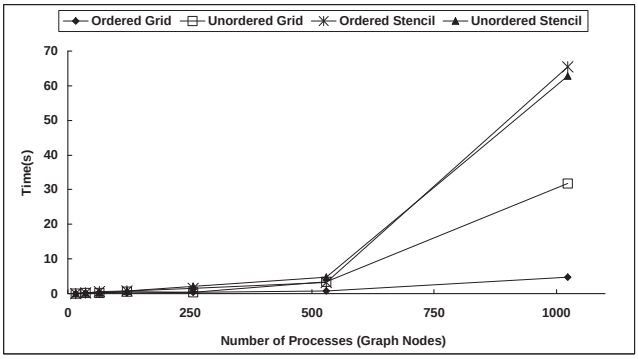

Figure 6. Graph spectrum performance on synthetic communication patterns

The results for graph isomorphism test are plotted in Figure 7. The processing times are very low for all topologies up to 1000 nodes, except for for unordered stencil. The processing time for the unordered stencil rises rapidly, and was around 4000 seconds for 529 nodes. Larger size cases could not be completed. We speculate that the reasons are related to heuristics employed for graph isomorphism. This represents a potential limitation. However, this 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 a nearly random numbering that was used in this stress test. Overall, we conclude that the methodology is effective for 1000 or more nodes for any numbering of processes that are likely to be encountered in practice.

6 Discussion

We present some observations on the enhancement of the methodology and extensions of the approach.

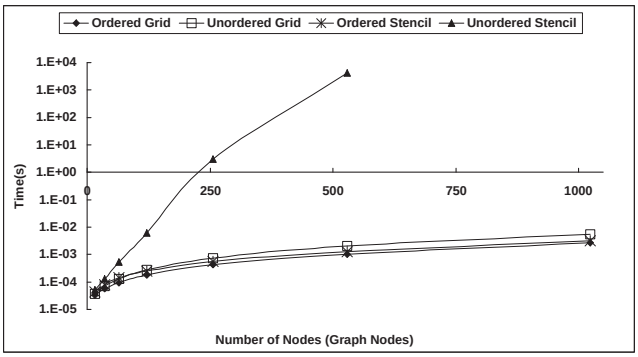

Figure 7. Graph isomorphism performance on synthetic communication patterns

6.1 Methodology

Performance of graph spectrum and isomorphism: One unexpected experimental result is that the execution time of polynomial eigenvalue computation for the graph spectrum test, although low, was much higher than the execution time for the heuristic graph isomorphism test for all benchmarks. Since graph spectrum test is meant to reduce the cases for graph isomorphism test, this result brings into question the need for the graph spectrum test. However, as the synthetic graph results show, graph isomorphism processing time is highly variable and very sensitive to numbering of nodes in isomorphic graphs. It was very expensive for some randomly numbered synthetic data. Hence we still believe it is appropriate to have the eigenvalue based test in the suite. One possibility is a framework based on concurrent application of graph spectrum and graph isomorphism tests.

On the fly implementation: Our implementation is based on stored trace files. However, processing on each trace file is minimal - it simply involves recording the number of communication calls and data transfered. Related work [8] emphasizes on-the-fly processing of traces citing very large trace files for some applications. However, the size of the traces we record is significantly smaller than those reported in this work for the same benchmarks. Trace size is related to the end application of profiling and the difference in this case may be the recording of call chains. Hashing can also be used to reduce trace volumes inexpensively. While the bulk of the trace processing in our method can be done on-the-fly, our experience and results do not indicate that it is critical.

6.2 Limitations and extensions

We discuss some of the restrictions and extensions of the implementation presented in this paper.

Static and known patterns: The approach assumes that the dominant execution pattern is known and static. However, it is important to note that the pattern library is extensible and entire families of patterns can be specified succintly, such as all sizes and dimensions of grids or all trees with a specific degree. When the procedure fails because a pattern is unknown, specifying the pattern manually still allows the construction of the logical trace.

Multiple phases: An application can have multiple phases where the communication pattern changes across phases. Clearly the approach presented in this paper will be far more effective if the phases were separated and logicalization was done one phase at a time. Phase separation based on detection of change in the communication pattern is possible but we have not implemented it.

Multiple patterns: Communication in some applications consists of multiple well defined patterns. One form is where the patterns are spatially separated, e.g., by the use of MPI communicators. We can identify the communicator structure from the trace. It is possible to discover topologies independently within separate communicators but this is not implemented. Another form is where multiple patterns exist temporally across all processes during execution. A known pattern is computation on a grid followed by a tree reduction. Our current approach is limited to a single pattern. The approach can be extended to identify mixed well defined patterns which have been studied in [7]. However, the computation feasibility for larger application sizes is yet to be determined. Such an approach would involve addressing the subgraph isomorphism problem. This problem is known to be NP-complete and our experiments show that it is computationally expensive in practice also.

7 Conclusions

Application communication traces are at the core of performance analysis and performance modeling of communicating parallel programs. This paper presents a framework to automatically construct a single logical trace that is representative of the overall parallel execution. 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 are presented that show that the procedure was successful and efficient for the NAS benchmark suite. Detailed analysis of performance data show that the execution time of trace logicalization is likely to be modest for most, if not all, realistic scenarios. The basic limitation of the approach is that the dominant application communication patterns must be static. Some potential weaknesses of the approach and extension to a wider class of applications and scenarios is discussed. 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.

Acknowledgement: This material is based upon work supported by the National Science Foundation under Grant No. ACI- 0234328 and Grant No. CNS-0410797

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] M. Noeth, F. Mueller, M. Schulz, and M. 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.

[9] R. M. Shuyi Shao, Alex K. Jones. A compiler-based communication analysis approach for multiprocessor systems. In *20th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Rhodes Island, Greece., April 2006.

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

[11] S. Sodhi and J. Subhlok. Automatic construction and evaluation of performance skeletons. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, April 2005.

[12] S. Sodhi, Q. Xu, and J. Subhlok. Performance prediction with skeletons. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 2008. Available Online through Springerlink.

[13] D. P. Spooner and D. J. Kerbyson. Performance feature identification by comparative trace analysis. *Future Generation Comp. Syst.*, 22(3):369–380, 2006.

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

[15] A. Toomula and J. Subhlok. Replication memory behavior for performance prediction. In *LCR 2004: The 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Houston, TX, October 2004.

[16] 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.

[17] 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.

[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. *Automatic Construction of Coordinated Performance Skeletons*. PhD thesis, University of Houston, August 2007.

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

Benchmark	Processes	Simple Tests	Graph Spectrum Test	Isomorphism Test
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 7. Identification of communication topologies of NAS benchmarks. Unique topologies are listed in boldface. Topologies listed not in boldface are isomorphic to the boldface topology above them.