Towards Adapting Parallel Programs to Different Platforms: Identifying Interaction Patterns

Bogdan Florin Cornea*, Jaroslaw Slawinski*, Julien Bourgeois†‡, Vaidy Sunderam*
*Mathematics & Computer Science, Emory University, USA
Email: {bcornea, jslawin, vss}@emory.edu
†UFC/FEMTO-ST Institute, UMR CNRS 6174, France
‡Carnegie Mellon University, USA
Email: julien.bourgeois@femto-st.fr

***Abstract*—Modern parallel computing platforms exhibit substantial variation in communication performance between on-socket, on-node and inter-rack locations. When application interaction patterns are irregular, communication-aware process placement on such platforms can be critical for overall runtime. Understanding program interaction patterns can be valuable in order to minimize the impact of capacity variations in communications parameters. Such knowledge can also be useful in selecting the best execution platform from the available options. In this paper we propose an approach based on source code analysis which identifies parallel application communication patterns such as star, ring, mesh, or torus. Our implementation based on ROSE framework was tested on various parallel programs that exhibit different communication patterns. We present the accuracy as well as the limitations of our static identification approach. We envisage augmenting our approach with trace information from pilot runs and best effort approaches to determine a process interaction graph.**

Keywords*—MPI, static analysis, communication patterns, HPC, novel computing architectures.

I. Introduction

Constant advances in computing platforms are a mixed blessing, especially for high performance parallel scientific and engineering applications. On the one hand, emerging platforms provide greater power and flexibility; on the other hand, they often necessitate substantial effort in the build, deployment, and runtime management phases. Sometimes they even require application modifications. In essence, as high performance computing parallel applications are tuned to match architectures existing at the time, they require periodic re-development according to latest trends for ensuring effective executability on evolving architectures.

As part of an effort to enhance executability of high performance parallel computing applications on varied target platforms [1], we address the application to platform matching, from two perspectives. First, given the choice of multiple platforms on which an application may be executed, there may be a subset that is better suited not only in terms of processor types, memory configuration, or storage access, but most importantly in terms of network interconnection type. At a very basic level, the interconnect topology of a parallel machine may exactly match the interaction pattern of the application. If other characteristics (such as CPU speed, memory capacity) are equal or comparable, it can be reasonably expected that this platform will be the best choice, as this perfectly matches the application communication requirements. Second, at a more detailed level, contemporary parallel computing platforms exhibit substantial heterogeneity in their communication capacity between different pairs of processing elements. The interconnect capability between two processor cores on the same node can be orders of magnitude better than the one between processor cores on the same cloud cluster platform that happen to be allocated in different racks or even across data centers. At the same time, application processes often have vastly differing communication needs between different pairs of processes. Therefore, placing application program components (processes) on parallel computing platforms in a manner that minimizes communication traffic across slower interfaces can be beneficial.

In this paper, we outline an alternative scheme, which uses static program analysis, to characterize communication patterns in parallel programs based on the MPI standard [2]. This approach was preferred over a dynamic (or *post-facto*) one mainly because (i) it focuses on the impact of communication heterogeneity, (ii) does not require executing the analyzed application, hence the approach is platform-independent, and (iii) it constitutes an intermediate phase towards a larger goal i.e. mapping topologies on varied execution platforms.

Our approach determines whether an MPI application uses a regular or an irregular communication topology. It was implemented in C, as part of the ADAPT framework [1] (Adaptive Application and Platform Translation). Obtaining the actual process placement does not constitute the focus of this paper, although we are making efforts in this direction. Determining the communication signature is valuable in the future for satisfying the requirements of the analyzed application with most suited platform capabilities. In this context, this work contributes to adapting parallel applications to the execution on target platforms, in a way that would lead to obtaining higher execution performance.

This paper is structured as follows: previous work is discussed in Section II; in Section III we give details about our motivation that led to this work, present the methodology and how the latter was implemented in ADAPT; for several small programs and for two benchmarks from the NAS Parallel Benchmarks [3] (NPB)—Integer Sort (IS) and Data Traffic (DT)—we discuss the communication patterns identified with our implementation (see Section IV); we conclude and present our perspectives in Section V.

II. Related work

On contemporary platforms, large variations in interconnects and subsequently in their data transfer performance are common: core-core, socket-socket, node-node, rack-rack, datacenter-datacenter, and cloud-cloud communications can vary by orders of magnitude [4].

In order to achieve higher performance for distributed applications executed on heterogeneous architectures, most approaches aim at optimizing the way parallelism is implemented at program level [5]–[10]. All these projects rely on execution traces to identify the communication pattern of applications. Bathia *et al.* [5] aim at identifying parallelization strategies at application level that can be optimized. Xu *et al.* [6] identify the communication pattern and use this knowledge to perform event trace-compression with the purpose of obtaining application performance skeletons. Preissl *et al.* [7], [8] present a method based on the ROSE framework [11], which was extended in [9] with the purpose of optimizing collective operations through static and dynamic analysis of applications. Although the above methods have proven efficient in identifying communication topologies, they all differ two-fold from the work presented here: first, our approach relies on identifying the communication patterns without prior execution of the code, and second, our method is network-centric, meaning that the knowledge we obtain on the communication topology is intended for a better mapping of processes to processing elements of the physical topology.

Alawneh *et al.* [10] only tackle the abstraction of execution traces, for which they propose two techniques that identify patterns from communication traces. This method gives good results but it requires program execution (for obtaining the necessary traces) and does not tackle the application execution optimization.

A different set of approaches that reduces the impact of communication heterogeneity focuses on obtaining a "communication signature" for the parallel application that may then be used to optimize the use of the physical network. These frameworks [12], [13] use static code analysis and have proven very efficient in identifying parallel communication patterns. Shao *et al.* [12] identifies the application logical topology as well as the different communication phases. Bronevetsky [13] proposes a more thorough static analysis using ROSE compiler, relying on the concept of parallel control flow graphs (pCFG) to identify communication patterns. Although Shao *et al.* have a similar approach to the work presented in this paper, their analysis relies on a different compiler (SUIF [14]) and can only match `sends` and `receives` at runtime. On the other hand, Bronevetsky has a completely different static approach, the only common point with our work being the use of ROSE compiler as a basis for building the static analyzer. His method gives very good results for (blocking) `MPI_Send` and `MPI_Recv`, but does not support any other MPI event. Relying on application execution to match point to point communication limits the applicability of the tool [12] in the sense that it requires for the target architecture to be physically available. By considering only `MPI_Send` and `MPI_Recv`, the tool [13] does not cover the wide range of message passing functions that may be used in parallel applications. Such approach is limited in the sense that it narrows down the range of applications that can be analyzed.

III. Static identification of program communication topology

The development and the execution of real life parallel applications has a significant impact on performance. From an application development viewpoint, we are interested in the implementation of parallel communication, i.e., in the logical topology also referred to as communication signature or communication pattern. With respect to program execution, we are concerned only with the target architecture network connection, or the physical topology.

A. Motivation

In a parallel program, the volume and the number of messages exchanged by participating processes is heavily dependent on the physical topology of the computing system. In this context, researchers proposed many different methods for improving application execution performance, new mapping methods [15]–[18], modification of MPI [19], congestion reduction [20] or clustering of processes [21].

While *post-facto* (dynamic) methods can have some use e.g. for subsequent runs of the same application with the same set of execution parameters, or can provide better information on data dependencies, they are sometimes impractical or too expensive. Because of this, our method uses static identification of the communication patterns of parallel applications that is a prerequisite for our future work on mapping topologies on varied execution platforms. Our motivation comes from successful previous work on identifying communication patterns (see Section II), as well as from the above-mentioned research and their limitations in application portability imposed by evolving architectures.

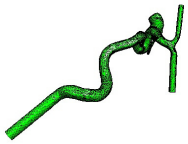
From our point of view, logical topology identification can be used in two ways, both aiming at increasing application performance:

First, given any scientific parallel C/MPI application found within the scope of our tool (see Section III-D), we can identify the best execution platform, from a variety of available ones, having a physical topology close to, or matching, the application communication pattern. This approach does not take into account the volume nor the number of exchanged messages.

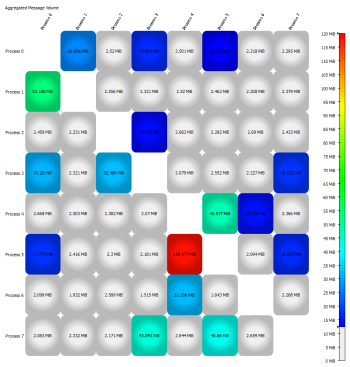
Second, given the capabilities of an execution platform, we are interested in studying various process to processor core mappings, as to obtain a better usage of the network. Process placement that utilizes the fastest links for high-traffic interfaces will likely result in the best overall performance. One approach for a topology-aware mapping is to superpose two weighted graphs: (i) the logical topology graph of the application and (ii) the physical topology graph of the platform. In this manner, recommendations can be made regarding the placement of parallel processes such that pairs exchanging high volume of messages use the network interconnect with the largest bandwidth, whereas pairs exchanging messages frequently will use network interconnects with low latency. For example, let us consider a sample from a real application which computes the fluid dynamics of a blood vessel (see Figure 1a). For the purpose of better visualizing and describing this example, the signatures were obtained *post-facto* through

analysis of trace files generated through instrumentation and the Vampir [22] software tool. The program was executed on 8 processes and the communication pattern in Figures 1d and 1c were obtained. Edge thickness indicates the volume (1d) or the number of messages (1c) exchanged between processes. For determining the graph in Figure 1d, the volume of communication is obtained from the communication matrix shown in Figure 1b to which we apply a threshold ε_v[1] of 12 MiB, representing in this case 10 percent of the 120 MiB maximum volume of exchanged data. Similarly, a threshold ε_{nb}[1] of 22,000 exchanged messages, representing for this example half of the maximum number of messages, is applied for obtaining the pattern in Figure 1c for process mapping.

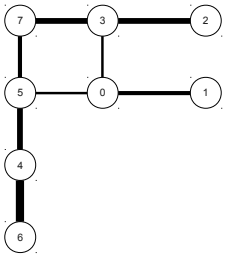
By filtering out low traffic—in terms of volume or number of messages—the application communication can be characterized by (i) a regular pattern, such as star, ring or mesh, (ii) by an irregular one, (iii) or by communication phases characterized by different patterns. Assuming that the candidate



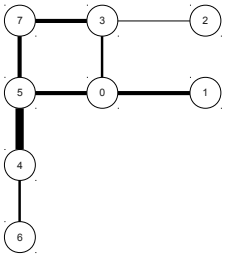
(a) The blood vessel.



(b) Volume of exchanged messages.



(c) Communication pattern. Edge thickness shows the number of messages.



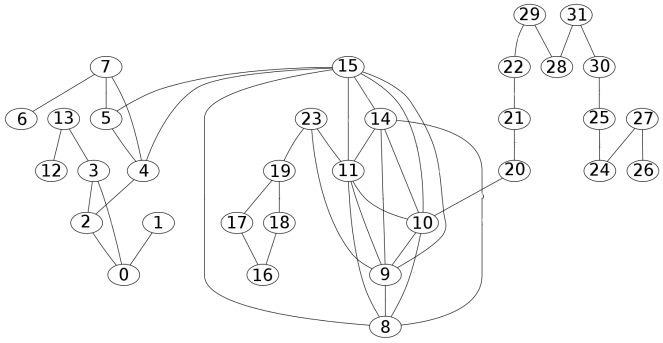
(d) Communication pattern. Edge thickness shows communication volume.

Fig. 1: A real life application simulating on 8 processes the fluid dynamics of a blood vessel.

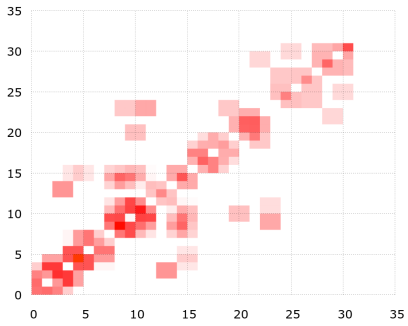
platforms are listed in Table I, and knowing the communication topology (Figure 1d), we observe that all processes can be placed on a single computing node on platforms 1 and 2 (Table I), or split into groups of maximum 4 processes (see Figure 3a) on platform 3. When executing the application on 32 nodes, the new communication topology (see Figures 2a and 2b) would benefit if process placement on platforms 1, 2, and 3 would be done as seen in Figures 3b, 3c, and 3d, respectively.

TABLE I: Example of available execution platforms. Intra-node (on-node) network capability is much higher than inter-node one.

	Name	Topology	Cores per node	Inter-node[2] network
1	Titan	3D torus	16	Cray Gemini
2	SuperMUC	tree	8	Infiniband
3	Puma	star	4	Gigabit Ethernet



(a) The communication graph on 32 processes.



(b) The communication pattern on 32 processes. X,Y axis are the processes. White through red is the overall message volume.

Fig. 2: A real life application simulating on 32 processes the fluid dynamics of a blood vessel.

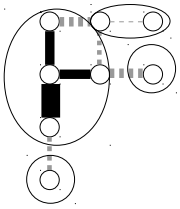
Identifying application communication signature is, in our opinion, useful either when choosing the target platform, or when mapping processes on a given platform. While both aspects aim at increasing application execution performance, in this paper we present the pattern identification aspect of our ongoing project.

B. Methodology

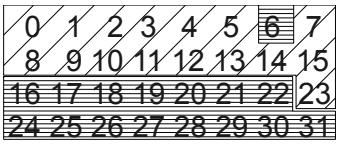
The following logical topologies can be identified by our framework: star, ring, two-dimensional (2D) mesh and two-dimensional (2D) torus.

[1] ε_v and ε_{nb} are the thresholds for volume and number of messages, respectively, above which edges are considered during the identification of the logical topology. The values of ε_v and ε_{nb} must not eliminate any process from the logical topology graph.

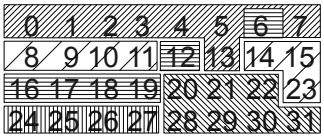
[2] By inter-node we refer to the communication between processing units located on different computing nodes, or node boards. It opposes the intra-node, or on-node, communication which takes place between processing units located on the same computing node.



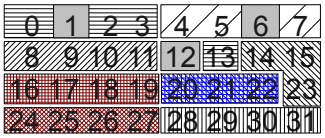
(a) On Puma, 8 processes are placed on a star topology.



(b) On Titan, 32 processes are placed on a 3D torus topology.



(c) On SuperMUC, 32 processes are placed on a fat tree topology.



(d) On Puma, 32 processes are placed on a star topology.

Fig. 3: Using the knowledge on communication topology acquired from a real life application to place processes on the processing elements (PE).

For all communication patterns defined in the following sub-sections, let P be a parallel program and let G be an undirected communication graph, with $G = (V,E)$. $V(G)$ is a set of *vertices* and $E(G)$ is a set of *edges*. Each vertex represents a parallel *process* of P and each edge corresponds to an existing bi-directional *communication* between two vertices (processes) of P. We emphasize that $k \in \mathbb{N}$ and the processes used in the following formulae (apart from *star*) range from 0 to $N-1$ ($k \in [0;N-1]$). $V_d(G)$ is the set of all vertices from G where each element has d communication neighbors.

***Star* pattern definition**

Processes in program P communicate in a star logical topology if and only if $\forall k \in [1;N-1]$,

$$E_{star}(G) = \{\{v_0 v_k\}\} \, and \, V_1(G) = \{v_k\} \tag{1}$$

with $G(V,E) = G(V,E_{star})$ and $V = V_1 \cup \{v_0\}$.

***Ring* pattern definition**

In this unidimensional logical topology, we state that program P follows a ring logical topology if $\forall k \in [0;N-1]$,

$$E_{ring}(G) = \{\{v_k v_j\} | j = (k \pm 1) \, mod \, N\} \, and \, V_2(G) = \{v_k\} \tag{2}$$

with $G(V,E) = G(V_2,E_{ring})$.

***2D mesh* pattern definition**

Processes in program P communicate in a two-dimensional mesh logical topology if $\forall i \in [0;c-1]$, $j \in [0;l-1]$,

$$E_{mesh2D}(G) = \left\{ \{v_{i,j} v_{i+x,j+y}\} \left| \begin{array}{l} -1 \leq x,y \leq 1; \\ 0 \leq i+x \leq c-1; \\ 0 \leq j+y \leq l-1; \\ |x+y| = 1 \end{array} \right. \right\} \tag{3}$$

with $i,j,x,y,c,l \in \mathbb{N}$, $G(V,E) = G(V,E_{mesh2D})$.

***2D torus* pattern definition**

Program P is following a two-dimensional torus pattern if $\forall i \in [0;c-1]$, $j \in [0;l-1]$,

$$E_{torus2D}(G) = \left\{ \{v_{i,j} v_{(i+x) \, mod \, c,(j+y) \, mod \, l}\} \left| \begin{array}{l} -1 \leq x,y \leq 1; \\ |x+y| = 1 \end{array} \right. \right\} \tag{4}$$

with $i,j,x,y,c,l \in \mathbb{N}$. The number of elements in set V becomes $|V| = |V_4(G)| = c \cdot l$ processes.

***Mixed* communication pattern**

In some cases, throughout the execution, applications use several topologies. This leads to the assumption that at least two patterns from those presented earlier in this section must be correctly identified by the end of the static analysis. Communication phases can be identified such that in each phase a single logical topology would characterize the communication. This type of pattern reduces false-positive results of our approach.

***Irregular* communication pattern**

If none of the previously presented topologies is identified, we assume that the program communication graph has an irregular pattern. We represent this as a graph $G(V_{ir},E_{ir})$ having the following properties: $V_{ir}(G)$ is the set of processes, with each process having any number of neighbors and $E_{ir}(G)$ is the set of edges for each element in graph G. Then, $\forall k,j \in [0;N-1]$, we define the edges of an irregular graph as follows:

$$E_{ir}(G) = \left\{ \{v_k v_j\} \left| \begin{array}{l} volume(v_k v_j) \geq \varepsilon_v; \\ total_{msg}(v_k v_j) \geq \varepsilon_{nb} \end{array} \right. \right\} \tag{5}$$

with $k,j \in \mathbb{N}$. $Volume(v_k v_j)$, and $total_{msg}(v_k v_j)$ are, respectively, the volume and the number of messages exchanged between v_k and v_j. As mentioned previously, ε_v and ε_{nb} are the volume and number of messages, respectively, above which edges are considered during the identification of the logical topology. The values of ε_v and ε_{nb} must not eliminate any process v_k from the logical topology graph.

In order to identify program interaction patterns through static analysis, all communication patterns presented in this section have been implemented as an extension to the ADAPT framework, and are described in the next section.

C. Implementation

The architecture of our static analyzer is presented in Figure 4. The *input* consists all source files of a C/MPI program. The analyzer—our module for communication pattern identification—is based on the *ROSE framework* [11] which is a compiler infrastructure supplying very complex methods for source-to-source code analysis and transformation. The output consists of information with respect to the communication pattern identified in the application.

In a first phase, the input is pre-processed by the Edison Design Group C compiler [23] available from ROSE. This generates several intermediate representations (IR), but we only use the Abstract Syntax Tree (AST) and the System Dependence Graph (SDG). On the one hand, the AST is an accurate representation of the input code, at a point where, at the end of an analysis or transformation, this IR can be *unparsed* to generate a new source code containing all transformations made on the AST. On the other hand, the SDG

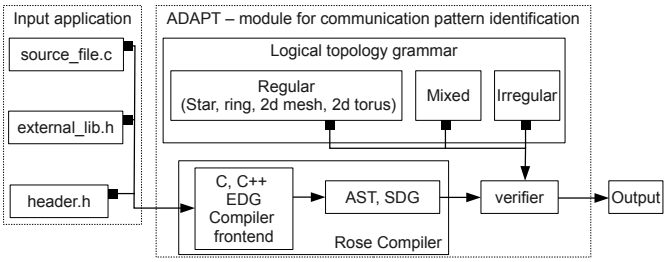


Fig. 4: ADAPT module for identifying program communication patterns.

is a complex IR useful especially in analyzing the `data` and `control` dependencies of the input code.

In a second phase, we implement the *verifier* such that it traverses the AST in search of MPI communication events that need to validate or invalidate the logical topology rules presented in Section III-B. If MPI events use variable names or expressions for the `source` or `destination`, an isolated traversal is done on the SDG until the `source` or the `destination` are found to depend upon constants or key variables such as `my_rank`, `comm_size`, `numprocs`. The following paragraphs present they way our *verifier* searches in the AST and the SDG for any of the communication patterns defined in Section III-B.

The *star* pattern requires identifying in the AST the MPI functions for sending and receiving. From each MPI event identified in the AST, the upper nodes of the IR are analyzed. In the input code, these are the instruction blocks that contain the MPI communication. By analyzing the conditions in statements, such as `for` or `if`, it can be decided if the program communicates according to formula (1) or not. Two elements that play a key role in analyzing the conditions are the number of processes and the number of iterations for the dominant communication loop.

The *ring* pattern uses a slightly different approach than for star pattern. Assuming that the programs use `MPI_Send` and `MPI_Recv` to communicate in a ring, for all MPI events found, which are of type `SgFunctionCallExp` in the AST, the focus is set on the fourth argument, i.e., the `source` and `destination`. In the case of `MPI_Send` and `MPI_Recv`, these elements could be of the following type: (i) a constant value, (ii) a macro, (iii) a mathematical expression, or (iv) a variable. When generating the AST, all macros are preprocessed and replaced with their corresponding value. When dealing with mathematical expressions, the AST node of the expression is parsed and split into operations and operands (left-hand and right-hand sides). From this point, the IR node representing each variable from the AST is identified in the SDG and its propagation can be traced back. This operation on the SDG helps express the `source` and `destination` variables with respect to common key elements such as the number of processes, the current rank, or the communication size. This asserts the applicability of formula (2) in the communication pattern.

The *two-dimensional mesh* pattern is identified in two ways: (i) by analyzing the use of `MPI_Cart_create` or (ii) by identifying the manually defined communication topology (see formula 3).

If `MPI_Cart_create` is found in the AST, our module extracts from the argument list 3 elements: one integer value and two arrays of integers. The integer gives the number of dimensions of the mesh, one array is for the number of processes in each dimension, and the other array of integers is for the periodicity of each dimension.

However, when the 2D mesh logical topology is not defined using `MPI_Cart_create`, the *verifier* applies formula (3) on the MPI events found in the AST. For this purpose, we assume that process ranking, or the numbering of processes, is done from top-left to bottom-right position, with 1 through $c \cdot l$ values in consecutive order, with c the number of columns and l the number of rows. For any analyzed applications, if $V_i(G_k)$ has a value different from the ones here-below, we state that logical topology used is not a 2D mesh.

$$| V_i(G) | = \begin{cases} 4\,processes, & i = 2 \\ (c+l-4)\cdot 2\,processes, & i = 3 \\ (c-2)\cdot(l-2)\,processes, & i = 4 \end{cases} \quad (6)$$

with i the number of neighbor processes from $V_i(G)$, c the columns and l the rows of the 2D mesh. Process ranks are expressed by using the SDG and by following a reverse propagation of each relevant variable. They are then expressed relative to constants or to MPI initialization variables (number of processes or current process rank).

The *two-dimensional torus* pattern is identified similarly to the two-dimensional mesh, with one difference, i.e., the use of `MPI_Cart_create` is expected to use periodicity. Otherwise, the manually defined communication topology is checked using the formula (4), verifying if each process is connected to four specific processes in order to comply with the 2D torus definition.

D. Scope and limitations of static analysis

The use of ROSE comes with some limitations to our implementation. First, the input application must comply with the C standard. Then, a multiple file project requires merging the AST obtained for individual files into one representation of the entire project. Third, static analysis requires that communication be initiated in the analyzed source code and not by underlying libraries. Our approach does not address the algorithmic correctness of the programs, nor does it search to optimize the code, as these constitute the focus of other research teams.

As some developers might choose not to use common implementation of algorithms, our tool might require supplementary effort for supporting the analysis of their programs. The number of processes, the source and destination of communication processes must be defined statically or with respect to MPI program variables, i.e., total number of processes and current rank. If this is not the case, or if data-dependency can not be solved using AST and SDG, our static analysis is not applicable.

Currently, only communication patterns mentioned in Section III-B are supported, but the architecture of ADAPT allows new communication topology modules to be added without effort, extending the identification capabilities of our approach. The fully connected mesh pattern was not presented in Section III-B, but it is identified when collective communications, such as `MPI_Alltoall`, `MPI_Allgather`, are used, and there is no `MPI_Cart_create` found in the code.

IV. EXPERIMENTS

Several applications were chosen to verify our method. For better comparing the logical topology discovery results, we consider only C/MPI applications.

A. MPI events used during the experiments

We briefly present the main characteristics of the MPI events that were identified in the applications used in this section.

As our approach is a static one, the information required to identify the communication pattern of applications is extracted from (i) the MPI event name—`MPI_`**`Send`**, `MPI_`**`Recv`**, `MPI_`**`Gather`** and so on—and from (ii) the arguments of the MPI call, such as the source (for `MPI_Recv`) or the destination (for `MPI_Send`).

In the message passing standard, parallel processes are organized in `groups`, and a group of processes that may communicate to each other is associated with a `communicator`. Our approach assumes that all processes of an analyzed application exchange messages as part of the same communicator.

For the point to point blocking communications **`MPI_Send`** and **`MPI_Recv`** the relevant arguments are the sender and receiver ranks, the exchanged data size and type. The collective communication **`MPI_Bcast`** broadcasts a message from the root process—usually rank 0—to all processes (of the communicator). Broadcasting is characteristic to "star" type of communication. The **`MPI_Reduce`** event, as opposed to broadcast, collects at process "root" the values from all processes in the communicator, by applying an arithmetic operator. This occurs in "star" communications as well. The **`MPI_Barrier`** function is used to synchronize processes in a communicator. When reached, it blocks the execution until all other processes have reached the same routine. It does not give any information regarding the communication topology. **`MPI_Sendrecv_replace`** performs a `Send` and a `Recv` operation using a single buffer. By itself it does not provide sufficient information about the communication topology. A call to **`MPI_Cart_create`** function creates a new communicator containing information about the communication topology. In `C`, this function has 5 arguments among which we mention the number of dimensions for the new Cartesian grid, an array indicating the number of processes in each dimension, and an array stating if the grid is periodic or not in each of the dimensions. **`MPI_Scatter`** is a collective communication that scatters a message over all processes from a communicator. The opposite of this collective communication is **`MPI_Gather`**, and they both indicate a "star" communication. The identification of collective events such as **`MPI_Alltoall`**, **`MPI_Alltoallv`**, **`MPI_Allreduce`** generally indicate that the processes communicate in a fully-connected mesh.

B. Asserting the star pattern

For this experiment, we use three C/MPI applications (see Table II, programs 1-3). All three programs implement differently the calculus of π (`PI`).

For the trivial case of *test code #1*, our tool indicates that it uses a *star* topology by verifying formula (1). The information extracted by our tool from the AST which led to this conclusion are presented in Table III. The result is expressed according to the number of processes (`p_numtasks`) and the number of iterations for the dominant communication loop (10 in this case), both values identified in the AST.

For the *test code #2*, the `source` and the `destination` parameters from the AST that correspond to `MPI_Bcast` and `MPI_Reduce` respectively, are verified by using formula (1) and an identification is done.

Similarly, a correct identification is done for the *test code #3*.

TABLE III: *Test code #1*: result of the static analysis for calculus of π.

Phase	MPI event	Source process	Destination process
1	Send	p1, p2, .. p10	p0
2	Recv	p1, p2 .. p_numtasks	p0

C. Ring pattern

The approach implemented in ADAPT was tested on programs 4 and 5 (see Table II). We begin describing this approach on the `test code #4` for having the highest complexity of these two programs.

Our method identifies in `MPI_Send` and `MPI_Recv` their `destination` and `source`, respectively. These relevant arguments are variables `right` and `left` which must be expressed with respect to a common variable or constant. The expression elements—`SgVarRefExp` in the AST—must be found such that the left-hand side operand is either `right` or `left`, and the right-hand side operand is related to constants or to MPI initialization variables. The traversal of the AST indicates the name of the argument to look for, while the SDG points precisely to the data-dependency for both `right` and `left`. Our method identified the correct pattern due to the identification throughout the code of the following expressions containing `right` and `left`:

```
1: right = 0;
2: right = procnum + 1;
3: left = procnum - 1;
4: left = numprocs - 1;
```

where `numprocs` is the variable extracted from `MPI_Comm_size` argument list.

For `test code #5`, a similar method is used to identify the occurrence of the relevant variable `world_rank` as part of an expression's right-hand side operand. It is therefore found as dependent on the number of processes, which is the variable named `world_size`. This simply requires more in-depth analysis of the respective right-hand side operand in the IR.

D. Two-dimensional mesh pattern

For the *test code #6* listed in Table II, our framework identified a 2D mesh based on the following elements: (i) `MPI_Cart_create`, with arguments `ndim`, `dims`, `cyclic`, and (ii) `MPI_Barrier` as the only MPI event used. Our tool followed the reverse propagation of variables in the SDG and obtained:

TABLE II: Characteristics of C/MPI programs [24] used in the experimental section.

Code	Pattern	Communication type	MPI event(s)	Notes
1	star	Point-to-point	Send, Recv	Calculus of π
2	star	Collective	Bcast, Reduce	-
3	star	Collective	Reduce	-
4	ring	Point-to-point	Send, Recv	using variables
5	ring	Point-to-point	Send, Recv	using expressions
6	2D mesh	-	Barrier	MPI_Cart_create
7	mixed: 2D torus, star	Point-to-point, collective	Sendrecv_replace, Gather, Scatter, Bcast	MPI_Cart_create
8	mixed: ring, full mesh, star	Point-to-point, collective	Irecv, Send, Allreduce, Alltoall, Alltoallv, Bcast, Reduce	NAS IS
9	star	Point-to-point	Send, Recv	NAS DT

```
1: Topology defined using MPI_Cart_create;
2: dimensions (dim): 2
3: nodes/dimension: 3x2 (dims[0]=nrow=2;
   dims[1]=mcol=2)
4: periodicity/dimension: 0 (cyclic[0]=0;
   cyclic[1]=0)
```

The occurrence of any other MPI event would have meant that the pattern is mixed or irregular.

E. Mixed pattern: two-dimensional torus + star

For the *input code #7*, ADAPT identifies several communication patterns based on `MPI_Cart_create`, `MPI_Bcast`, `MPI_Scatter`, `MPI_Sendrecv_reduce`, `MPI_Gather`. This leads to an indication of 3 phases in the program, each characterized by one pattern: (i) star, (ii) irregular point-to-point, (iii) star. The arguments of `MPI_Cart_create` indicate that, in fact, the irregular pattern is a 2D torus:

```
1: dimensions: 2 (set by macro)
2: nodes/dimension: (int)sqrt((double)
   grid->Size)
3: periodicity/dimension: 1 (by rows and
   columns).
```

`Line 3` is the result of reversed variable propagation performed on the input by ADAPT using the SDG, starting from instruction line:
`Dimensions[0] = Dimensions[1] = grid->p_proc`.
Any additional MPI event would have added another communication phase characterized by a regular or an irregular pattern.

F. Verifying our approach on NAS IS (mixed pattern: ring + full mesh + star)

On this input code, our framework identifies 3 topologies: (i) ring, (ii) fully connected mesh, and (iii) star. The decision is based on the following MPI events identified in the code: `Irecv`, `Send`, `Allreduce`, `Alltoall`, `AlltoallV`, `Bcast`, `Reduce`. We observe that a result was obtained which takes into account if several regular topologies are found, and does not determine that the overall result of the analysis is an *irregular topology*. This code shows that our method lacks a finer grained *verifier* that estimates the time percentage spent communicating in each topology, which we intend to address in our future work.

G. Verifying our approach on NAS DT (star pattern)

For this benchmark available in NPB as of version 3.2, the identification result indicates the presence of point-to-point communication. By analyzing the conditionals containing the MPI event, and by verifying the data and control dependencies among variables or function calls, the identification result validates formula (1), i.e., the NAS Data Traffic benchmark communicates in a *star* topology.

We observe that by identifying the communication pattern in a static manner, our result is scalable in the way that it analyses the MPI event types found in the code as well as the conditional statements that they depend on. This approach consists of one advantage in comparison to trace-based identification, i.e., it reduces the possibility of identifying additional communication phases when the problem size increases.

V. Conclusions and future work

As part of the ADAPT project for achieving adaptation of C/MPI applications for execution on various computational resources, in this paper we propose a novel mechanism for automatically analyzing parallel applications and determining their communication signature. Our approach is based on the ROSE framework and it statically identifies regular communication topologies, i.e., star, ring, 2D mesh, 2D torus, as well as mixed and irregular communication patterns. Several parallel programs were used to verify the proposed method.

For the automatic process of mapping application communication signature to platform capabilities, our team is undergoing research on two directions: (i) providing micro-benchmarks for determining target platforms capabilities, and (ii) implementing in ADAPT a decision-making mechanism determining if static analysis using ROSE is applicable or if it is recommended to use historical runs and trace-based simulation instead. An approach using trace-based simulation is envisaged for parallel applications where communication is handled entirely by underlying libraries. We are working on an approach that estimates the percentage of occurrence of each communication phase throughout the execution of the program. The presented work is part of a framework for predicting the performance of distributed applications based on process-to-processor mapping. The current contribution is not yet available as a standalone tool, although we are taking this

aspect into consideration. Our future work aims at proposing a best process placement on the physical architecture, hence estimating a best application to platform performance.

ACKNOWLEDGMENT

This work is partially funded by the US National Science Foundation grant OCI-1124418.

REFERENCES

[1] J. Bourgeois, V. Sunderam, J. Slawinski, and B. Cornea, "Extending executability of applications on varied target platforms," in *IEEE HPCC'11: 13-th IEEE International Conference on High Performance Computing and Communications*. IEEE Computer Society, 2011.

[2] MPI, "The message passing interface standard," http://www-unix.mcs.anl.gov/mpi.

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber *et al.*, "The NAS Parallel Benchmarks," *International Journal of HPC Apps*, vol. 5, no. 3, p. 63, 1991.

[4] D. Reed and D. Grunwald, "The performance of multicomputer interconnection networks," *Computer*, vol. 20, no. 6, pp. 63 –73, june 1987.

[5] N. Bhatia, F. Song, F. Wolf, J. Dongarra, B. Mohr, and S. Moore, "Automatic experimental analysis of communication patterns in virtual topologies," in *ICPP'05: International Conference on Parallel Processing*, 2005, pp. 465–472.

[6] Q. Xu, J. Subhlok, R. Zheng, and S. Voss, "Logicalization of communication traces from parallel execution," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*, ser. IISWC'09. IEEE Computer Society, 2009, pp. 34–43.

[7] R. Preissl, T. Kockerbauer, M. Schulz, D. Kranzlmuller, B. Supinski, and D. Quinlan, "Detecting patterns in mpi communication traces," in *ICPP '08: 37th International Conference on Parallel Processing*, 2008, pp. 230–237.

[8] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. Supinski, and D. J. Quinlan, "Using mpi communication patterns to guide source code transformations," in *Proceedings of the 8th International Conference on Computational Science, Part III*, ser. ICCS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 253–260.

[9] R. Preissl, B. de Supinski, M. Schulz, D. Quinlan, D. Kranzlmuller, and T. Panas, "Exploitation of dynamic communication patterns through static analysis," in *ICPP'10: 39th International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 51–60.

[10] L. Alawneh and A. Hamou-Lhadj, "Pattern recognition techniques applied to the abstraction of traces of inter-process communication," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, ser. CSMR '11. IEEE Computer Society, 2011, pp. 211–220.

[11] M. Schordan and D. Quinlan, "A source-to-source architecture for user-defined optimizations," in *Modular Programming Languages*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2003, vol. 2789, pp. 214–223.

[12] S. Shao, A. K. Jones, and R. Melhem, "A compiler-based communication analysis approach for multiprocessor systems," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 85–85.

[13] G. Bronevetsky, "Communication-sensitive static dataflow for parallel message passing applications," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[14] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "SUIF: an infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Not.*, vol. 29, no. 12, pp. 31–37, 1994.

[15] C.-L. Chou and R. Marculescu, "Contention-aware application mapping for network-on-chip communication architectures," in *IEEE ICCD'08: IEEE International Conference on Computer Design*, 2008, pp. 164 –169.

[16] A. Bhatele, G. R. Gupta, L. V. Kalé, and I.-H. Chung, "Automated mapping of regular communication graphs on mesh interconnects," in *HiPC'10: International Conference on High Performance Computing*, 2010, pp. 1–10.

[17] A. Bhatele and L. V. Kalé, "Heuristic-based techniques for mapping irregular communication graphs to mesh topologies," in *IEEE HPCC'11: IEEE 13th International Conference on High Performance Computing and Communications*, 2011, pp. 765–771.

[18] A. Bhatele, T. Gamblin, S. H. Langer, P.-T. Bremer, E. W. Draeger, B. Hamann, K. E. Isaacs, A. G. Landge, J. A. Levine, V. Pascucci, M. Schulz, and C. H. Still, "Mapping applications with collectives over sub-communicators on torus networks," in *SC'12: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, pp. 97:1–97:11.

[19] M. J. Rashti, J. Green, P. Balaji, A. Afsahi, and W. Gropp, "Multi-core and network aware MPI topology functions," ser. EuroMPI'11: Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface. Springer-Verlag, 2011, pp. 50–60.

[20] T. Hoefler and M. Snir, "Generic Topology Mapping Strategies for Large-scale Parallel Architectures," in *ICS'11: Proceedings of the 2011 ACM International Conference on Supercomputing*. ACM, 2011, pp. 75–85.

[21] P. Fan, Z. Chen, J. Wang, Z. Zheng, and M. R. Lyu, "Topology-aware deployment of scientific applications in cloud computing," *IEEE CLOUD'12: IEEE 5th International Conference on Cloud Computing*, vol. 0, pp. 319–326, 2012.

[22] Vampir, "website," http://www.vampir.eu.

[23] EDG, "Edison design group front-end compiler website," http://www.edg.com.

[24] MPI, "Test codes," http://mathcs.emory.edu/~bogdan/publications/commPatterns.html.