I. INTRODUCTION

Different applications in the high performance computing (HPC) field exhibit different communication patterns, which can be characterized by three key attributes: volume, spatial and temporal [1, 2]. Proper understanding of communication patterns of parallel applications is important to optimize the communication performance of these applications [3, 4]. For example, with the knowledge of spatial and volume communication attributes, MPIPP [3] optimizes the performance of Message Passing Interface (MPI) programs on non-uniform communication platforms by tuning the scheme of process placement. Besides, such knowledge can also help design better communication subsystems. For instance, for circuit-switched networks used in parallel computing, communication patterns are used to pre-establish connections and eliminate the runtime overhead of path establishment. Furthermore, a recent work shows spatial and volume communication attributes can be employed by replay-based MPI debuggers to reduce replay overhead significantly [5].

Previous work on communication patterns of parallel applications mainly relies on traditional trace collection methods [2, 6, 7]. A series of trace collection and analysis tools have been developed, such as ITC/ITA, KOJAK, Paraver, TAU and VAMPIR [8-13]. These tools need to instrument original programs at the invocation points of communication routines. The instrumented programs are executed on full-scale parallel systems and communication traces are collected during the execution. The collected communication trace files record type, size, source and destination etc. for each message. The communication patterns of parallel applications can be easily generated from the communication traces. However, traditional communication trace collection methods have two main limitations: huge resource requirement and long trace collection time. For example, ASCI SAGE routinely runs on 2000-4000 processors [14] and FT program in the NPB consumes more than 600 GB memory for Class E input [15]. Therefore, it is impossible to use traditional trace collection methods to collect communication patterns of large-scale parallel applications without full-scale systems. Moreover, it takes several months to complete even on a system with thousands of CPUs. It is prohibitive long for trace collection and prevents many interesting explorations of using communication traces, such as input sensitivity analysis of communication patterns. Additionally, MPIP[16] is a lightweight profiling library for MPI applications and only collects statistical information of MPI functions. However, all these traditional trace collection methods require the execution of the entire instrumented programs, which restricts their wide usage for analyzing large-scale applications. Our method adopts the similar technique to capture the communication patterns at runtime as the traditional trace collection methods.

We have two observations on existing communication trace collection and analysis approaches. (I) Many important applications of communication pattern analysis, such as the process placement optimization [3] and subgroup replay [17], do not require temporal attributes. (II) Most computation and message contents in message-passing parallel applications are not relevant to their spatial and volume communication attributes. Motivated by the above observations, we describe a novel technique in this chapter, called FACT, which can perform fast communication trace collection for large-scale parallel applications on small-scale systems. Our idea is to reduce the original program to obtain a program slice through static analysis, and to execute the program slice to acquire communication traces. The program slice preserves all the variables and statements in the original program relevant to the spatial and volume attributes, but deletes any unrelated parts. In order to recognize the relevant variables and statements, we propose a live-propagation slicing algorithm (LPSA) to simplify original programs. By solving an inter-procedural data flow equation, it can identify all the variables and statements affecting the communication patterns. We have implemented FACT and evaluated it with 7 NPB programs as well as Sweep3D. The results show that FACT can preserve the spatial and volume communication attributes of original programs and reduce resource consumptions by two orders of magnitude in most cases.

Large-scale parallel computers usually consist of thousands of processor cores and cost millions of dollars which take years to design and implement. For designers of these computers, it is critical to answer the following question at the design phase: What is the performance of application X on a parallel machine Y with 10000 nodes connected by network Z? Accurate answer to the above question enables designers to

evaluate various design alternatives and make sure that the design can meet the performance goal. In addition, it also helps application developers to design and optimize applications even before the target machine is available. However, accurate performance prediction of parallel applications is difficult because the execution time of large parallel applications is determined by sequential computation time in each process, the communication time and their convolution. Due to the complex interactions between computation and communications, the prediction accuracy can be hurt significantly if either computation or communication time is estimated with notable errors.

In this chapter, we also focus on how to acquire sequential computation time accurately for large-scale parallel applications. This is because existing approaches address the communication time estimation and the convolution issues fairly well, such as BigNetSim and DIMEMAS [9, 18]. The bottleneck of current prediction framework is to estimate sequential computation time in each process accurately and efficiently for large-scale parallel applications on non-existing target parallel machines. We illustrate a novel approach based on deterministic replay techniques to solve the problem. There are two main contributions: (I) employing deterministic replay techniques to measure sequential computation time of strong-scaling applications without full-scale target machines and (II) employing representative replay to reduce measurement time. We have implemented a performance prediction framework, called PHANTOM, which integrates the above computation-time acquisition approach with a trace-driven network simulator. We validate our approach on several platforms. For ASCI Sweep3D, the error of our approach is less than 5% on 1024 processor cores. We compare the prediction accuracy of PHANTOM with a recent regression-based prediction approach. The results show that PHANTOM has better prediction accuracy across different platforms than the regression-based approach.

II. COMMUNICATION TRACE COLLECTION

We present an overview of our approach followed by our live-propagation slicing algorithm. The implementation and evaluation are also described in this section.

A. Design Overview

FACT consists of two primary components, a compilation framework and a runtime environment as shown in Figure 1. The compilation framework is divided into two phases, intra-procedural analysis followed by inter-procedural analysis. The program is sliced based on the results of the inter-procedural analysis. Finally, the communication traces are collected in the runtime environment.

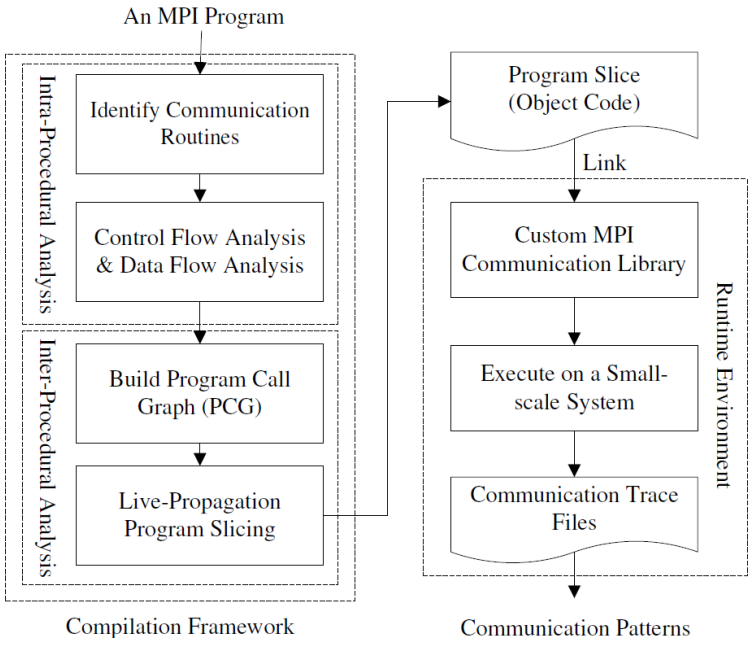

Fig. 1: Overview of FACT

During the intra-procedural analysis phase, FACT parses the source code of an MPI program and identifies the invoked communication routines. The relevant arguments of these routines that determine communication patterns are collected. Information about control dependence, data dependence and communication dependence for each procedure is gathered, which will be explained in detail in the following subsections. During the inter-procedural analysis phase, the program call graph is built based on the information of call sites collected during the intra-procedural phase. LPSA is used to identify all the variables and statements that affects the communication patterns. The output of the compilation framework is the program slice as well as directives for usage at runtime.

A program slice is a skeleton of the original program that cannot be executed on the system directly. Runtime environment of FACT provides a custom MPI communication library to collect the communication traces from the program slice based on the directives inserted at compile time. The program slice is linked to the custom communication library and executed on a small-scale system. The communication traces of applications are collected during the execution according to the specified problem size, input parameters and number of processes.

B. Live-Propagation Slicing Algorithm

From a formal point of view, the definition of program slice is based on the concept of slicing criterion [19]. A slicing criterion is a pair (*p*, *V*), where *p* is a program point, and *V* is a subset of the program variables. A program slice on the slicing criterion (*p*, *V*) is a subset of program statements that preserve the behavior of the original program at the program point *p* with respect to the program variables in *V*. Therefore, determining the slicing criterion and designing an efficient slicing algorithm according to the actual problem requirements are two key challenges in the compilation framework.

1) Slicing Criterion: Since our goal is to collect communication traces for analyzing spatial and volume communication attributes, we record the following communication properties in LPSA for a given parallel program:

- For point-to-point communication, we record message type, message size, message source and destination, message tag and communicator id.
- For collective communication, we record message type, sending message size, receiving message size, root id (if exist) and communicator id.

Message size, source and destination are used to compute spatial and volume communication attributes, while message type, message tag and communicator id are useful for other communication analysis.

```
MPI_Send(buf, count, type, dest, tag, comm)
buf : initial address of send buffer
[Comm] count: number of elements in send buffer
[Comm] type : datatype of each buffer element
[Comm] dest : rank of destination
[Comm] tag : uniquely identify a message
[Comm] comm : communication context
```

Fig. 2: Comm variables in the routine for *MPI_Send*. The variables marked with Comm directly determine the communication patterns of the parallel program.

In an MPI program, these properties can be acquired directly from the corresponding parameters of the MPI communication routines. For example, in the routine *MPI_Send* in Figure 2, the parameters *count* and *type* determine the message size. The parameters *dest* and *comm* determine the message destination. The message tag and communicator id can be acquired from the parameters *tag* and *comm*. The parameter buf does not affect the communication patterns directly. However, sometimes it may affect the communication patterns indirectly through data flow propagation and we will analyze it in the following subsections. Comm Variable is defined in this chapter to represent those parameters that determine the communication patterns directly.

Definition II.1 (Comm Variable). *Comm Variable is a parameter of a communication routine in a parallel*

program, the value of which directly determines the communication patterns of the parallel program.

As MPI is a standard communication interface, we can explicitly mark Comm Variables for each MPI routine. In Figure 2, Comm Variables in the routine for *MPI_Send* are marked. All the parameters, except *buf*, are Comm variables. When a communication routine is identified in the source code, the corresponding Comm Variables are collected. For each procedure *P*, we use a Comm Set, $C(P)$, to record all the Comm Variables. $C(P) = \{ (\ell, v) \mid \ell$ is the unique lable of v, v is a Comm Variable $\}$. The Comm Set $C(P)$ is the slicing criterion for simplifying the original program in LPSA, which will be optimized during the phase of data dependence analysis.

2) Dependence of MPI Programs: For convenience, we assume that a control flow graph (CFG) is built for each procedure and the program call graph (PCG) is constructed for the whole program. To describe our slicing algorithm easily, we use statement instead of basic block as a node in the CFG. We assume that each statement in the program is uniquely identified by its label l and is associated with two sets: DEF[ℓ], a set of variables whose values are defined at ℓ, and USE[ℓ], a set of variables whose values are used at ℓ.

In an MPI program, there are three main types of dependence for statements and variables that would change the behavior for a given program point, data dependence (dd), control dependence (cd) and communication dependence (md).

Data Dependence Data dependence between statements means that the program's computation might be changed if the relative order of statements were reversed [20]. To analyze the data dependence, we must first calculate the reaching definitions for each procedure. We define the *GEN* and *KILL* sets for each node in the CFG. Then we adopt the iterative algorithm presented in [21] to calculate the reaching definitions. The data flow graph (DFG) can be constructed based on the results of the reaching definitions analysis. The node in the DFG is either a statement or a predicate statement. The edge represents the data dependence of the variables. The data dependence information computed by the reaching definitions is stored in the data structures of DU and UD chains [22].

Definition II.2 (DU and UD Chain). *Def-use (DU) Chain links each definition of a variable to all of its possible uses. Use-def (UD) Chain links each use of a variable to a set of its definitions that can reach that use without any other intervening definition.*

We can further optimize the Comm Set based on the results of data flow analysis. If there are no other intervening definitions for the consecutive Comm Variables, we keep only the last Comm Variable.

Control Dependence If a statement *X* determines whether statement *Y* is executed, statement *Y* is control dependent on statement *X*. The DFG does not include information of control dependence. Control dependence can be computed with the post-dominance frontier algorithm [23]. In this chapter, we convert the control dependence into data dependence by treating the predicate statement as a definition statement and then incorporating the control dependence into the UD chains.

Communication Dependence Communication dependence is an inherent characteristic for MPI programs due to message passing behavior. MPI programs take advantage of explicit communication model to exchange data between different processes. For example, sending and receiving routines for the point-to-point communications are usually used in pairs in the programs.

Definition II.3 (Communication Dependence). *Statement X in process i is communication dependent on statement Y in process j, if*

1) Process *j* sends a message to process *i* through explicit communication routines.
2) Statement *X* is a receiving operation and statement *Y* is a sending operation ($X \neq Y$).

In MPI programs, both point-to-point communications and collective communications can introduce communication dependence. Due to limitations of space, we do not list more examples in this chapter.

Communication dependence can be computed through identifying all potential matching communication operations in MPI programs. Although in general, it is a difficult problem for static analysis to determine the matching operations, we find it is sufficient to deal with this problem using simple heuristics in practice.

We conservatively connect all potential sending operations with a receiving operation, and adopt some heuristics, such as mismatched tags or data types of message buffer, to prune edges that cannot represent real matches. We will further discuss the communication dependence issues in the following subsections.

In MPI programs, the message is exchanged through the message buffer variable, *buf*. The communication dependence can be represented with the message buffer variable. *msg_buf* (ℓ) is used to denote the message buffer variable in the communication statement ℓ. Additional considerations for non-blocking communications will be presented in the implementation of runtime environment.

Definition II.4 (MD Chain). *Message-Dependence Chain (MD Chain) links each variable of message receiving buffer to all of its sending operations*.

Definition II.5. *The slice set of an MPI program (M) with respect to the slicing criterion* $C(M)$, *denoted by* $C(M)$, *consists of all statements* ℓ *on which the values of variables in* $C(M)$ *directly or indirectly dependent. More formally:*

$$S(C(M)) = \{\ell \mid v \xrightarrow{d_1} \cdots \xrightarrow{d_n} \ell \rightarrow v, \in M(n), > 0, r \leq 1 \leq n, d_i \in \{d\}\}$$

We use the symbol $\rightarrow$ to denote the dependence between variables and statements. For computing the program slice with respect to the slicing criterion $C(M)$, we define LIVE Variable to record dependence relationship between the variables of programs. A Comm Variable itself is also a LIVE Variable based on the definition of LIVE Variable.

Definition II.6 (LIVE Variable). *A variable x is LIVE, if the change of its value at statement* ℓ *can affect the value of any Comm Variable v directly or indirectly through dependence of MPI programs, denoted by* $v \rightarrow^{*} (\ell, x)$. *There is a LIVE Set for each procedure P, LIVE* [P]. *LIVE*[P] = $\{(\ell, x) \mid v \rightarrow^{*} (\ell, x), v \in C(P)\}$.

3) Intra-Procedural Analysis: During the intra-procedural analysis phase, data dependence, control dependence and communication dependence are collected and put into corresponding data structures. Each procedure *P* is associated with two sets, *WL*[*P*] and LIVE[*P*]. *WL*[*P*] is a worklist that holds the variables waiting to be processed and LIVE[*P*] holds the LIVE Variables for procedure *P*. As program slicing in this chapter is a backward data flow problem, we use a worklist algorithm to traverse the UD chains and iteratively find all the LIVE Variables. We put the statements that define LIVE Variables into slice set $S[P]$ and mark MPI statements that define LIVE Variables or have communication dependence with marked MPI statements. The main body of the analysis algorithm is given in Algorithm 1. *receive_buf* denotes the message buffer variables in the receiving operations. The worklist *WL*[*P*] for each procedure is initialized with its Comm Set and LIVE[*P*] is initialized with null set.

In Algorithm 1, the statements not in slice sets except MPI routines are deleted, while all the MPI routines are retained. For unmarked MPI routines by Algorithm 1, it means that no LIVE Variable is defined or no communication dependence exists in these routines. The retained unmarked MPI routines are served for runtime environment of FACT to collect communication traces. In contrast, for marked MPI routines by Algorithm 1, LIVE Variables are defined or communication dependence exists in these routines. For MPI routines used for message passing, it means that the contents of messages are relevant to the communication patterns. In Figure 3, the LIVE Variable, *num*, is defined in *MPI_Irecv* of Line 5 that is communication dependent on *MPI_Send* of Line 2. Therefore, both MPI routines are marked by the algorithm and will be executed at runtime. Algorithm 1 is sufficient for the MPI program with one function. In the real parallel application, the program is always modularized with several procedures. In the following subsection, we will present additional considerations for inter-procedural analysis.

4) Inter-Procedural Analysis: Slicing across procedure boundaries is complicated due to the necessity of passing the LIVE Variables into and out of procedures. Because program slicing in this chapter is a backward data flow problem and the slicing criterion can arise either in the calling procedure (caller) or in the called procedure (callee), the LIVE Variable can propagate bidirectionally between the caller and the callee through parameter passing. To obtain a precise program slice, we adopt a two-phase traverse over the PCG, Top-Down phase followed by Bottom-Up phase. Additionally, the UD chains built during the

intra-procedural phase are refined to consider the side effects of procedure calls. In this chapter, we assume that all the parameters are passed by reference and our algorithm can be extended to the case where they are passed by value.

```
1. if(myid == 0){
2. [M] MPI_Send(&num, 1, MPI_INT, 1, 55,...)
3.       MPI_Recv(buf, num, MPI_INT, 1, 66,...)
4. }else{
5. [M] MPI_Irecv(&num, 1, MPI_INT, 0, 55,..., req)
6.       MPI_Wait(req,...)
7.       size = num
8.       MPI_Send(buf, size, MPI_INT, 0, 66,...)
9. }
```

Fig. 3: Marked MPI point-to-point communication routines by Algorithm 1 (*M* means marked).

```
 1. foo(){
 2.    a = 5
 3.    call bar(a)
 4.    size = a
 5.    call MPI_Send(...,size,...)
 6. }
 7. bar(b){
 8.    m = 4
 9.    b = m
10. }
```

Fig. 4: An example of LIVE Variable propagation from the caller to the callee.

MOD/REF Analysis To build precise UD chains we use the results of inter-procedural MOD/REF analysis. For example, in Figure 4, before incorporating the information from the MOD/REF analysis, $UD(4,a)=\{2,3\}$. We compute the following sets in the MOD/REF analysis for each procedure [24]: GMOD(*P*) and GREF(*P*). GMOD(*P*) is a set of variables that are modified by an invocation of procedure *P*, while GREF(*P*) is a set of variables that are referenced by an invocation of procedure *P* [25]. The information from the MOD/REF analysis tells us whether a variable is modified or referenced due to the procedure calls. With these results, we can refine the UD chains built during the intra-procedural analysis. For example, $UD(4,a)=\{3\}$.

Extension of MD Chains The MD chains collected during the intra-procedural phase do not include inter-procedural communication dependence. During the inter-procedural analysis phase, MD chains are extended to consider cross-procedural dependence. At the same time, Algorithm 1 is extended to Algorithm 2 that will be invoked by Algorithm 3. Only the different parts from Algorithm 1 are listed here. $P':\ell_j$ denotes the statement ℓ_j in procedure P'.

```
 1. foo(){
 2.    n = 5
 3.    a = n
 4.    call bar(a)
 5. }
 6. bar(b){
 7.    size = b
 8.    call MPI_Send(...,size,...)
 9.    ...
10. }
```

Fig. 5: An example of LIVE Variable propagation from the callee to the caller

Top-Down Analysis The Top-Down phase propagates the LIVE Variables from the caller to the callee over the PCG by binding the actual parameters of the caller to the formal parameters of the callee. As the LIVE Variable can be modified by the called procedure via parameter passing, we need to find the corresponding definition of this variable in the called procedure. For example, in Figure 4 we can compute from the intra-procedural analysis, that (3, *a*) is a LIVE Variable. This calling context is then passed into the procedure *bar*. The corresponding formal parameter in *bar* is the parameter *b*. There may be several definitions of *b* in procedure *bar*, however we only care about the last definitions (it is a set due to the effects of control flow) of variable *b* due to the property of the backward data flow analysis. This definition appears in statement 9 in *bar*. In addition, we put this statement into slice set and put its USE variables into the worklist of procedure *bar*. Other LIVE Variables in procedure *bar* can be computed iteratively by

Algorithm 2. In procedure *foo* the actual parameter (3, *a*) is no longer put into its worklist. We define the LIVE_Down function to formalize this data flow analysis.

Definition II.7 (LIVE Down). *Procedure P invokes procedure Q, v is a LIVE Variable and also an actual parameter at callsite ℓ in procedure P, v′ is the corresponding formal parameter in procedure Q, $LIVE\_Down(P,\ell,v,Q)$ returns statement set (L is the label set) of the last definitions of v′ in procedure Q: $LIVE\_Down(P,\ell,v,Q)=L$.*

Algorithm 1 Compute LIVE Set and Mark MPI statements for intra-procedure

```
1: procedure INTRA-LIVE(P)
2:   input: worklist WL[P] and LIVE set LIVE[P]
3:   output: program slice set of procedure: S(P)
4:   Change ← False
5:   while WL[P] ≠ ∅ do
6:     Remove an item (ℓ, v) from WL[P]
7:     if (ℓ, v) ∉ LIVE[P] then
8:     Change ← True
9      LIVE[P] ← {(ℓ_i, v)} ∪ LIVE[P]          ▷ Process communication dependence!
10:    if (ℓ_i, v) ∈ receive_buf then
11:      for ℓ_i ∈ MD(ℓ_i, v) do
12:        Mark MPI statement ℓ_i
13:        S(P) = S(P) ∪ {ℓ_j}
14:        WL[P] ← {(ℓ_j, msg_buf(ℓ_j))} ∪ WL[P]
15:      end for
16:    else                                    ▷ Process control and data dependence!
17:      for ℓ_k ∈ UD(ℓ_i, v) do
18:        if ℓ_k ∈ MPI_Routines then
19:          Mark MPI statement ℓ_k
20:        end if
21:        S(P) = S(P) ∪ {ℓ_k}
22:        for x ∈ USE[ℓ_k] do
23:          WL[P] ← {(ℓ_k, x)} ∪ WL[P]
24:        end for
25:      end for
26:    end if
27:  end if
28:  end while
29:  return S(P)
30: end procedure
```

Algorithm 2 Extension of INTRA-LIVE(P)

```
1: procedure INTRA-LIVE-EXT(P)
  ...
12: for each P′ : ℓ_j ∈ MD(ℓ_i, v) do
13:   Mark MPI statement P′ : ℓ_j
14:   S(P′) = S(P′) ∪ {ℓ_j}
15:   WL[P′] ← {(ℓ_j, msg_buf(ℓ_j))} ∪ WL[P′]
  …
```

Bottom-Up Analysis The Bottom-Up phase is responsible for propagating the LIVE Variables from the callee to the caller. For a LIVE Variable in the called procedure, if its definition is a formal parameter, we need to propagate the LIVE information by binding the formal parameters to the actual parameters. For example, in Figure 5, the formal parameter of *b* in procedure bar is a LIVE Variable computed by the intra-procedure analysis. We need to propagate this information into the calling procedure *foo*. The

corresponding actual parameter is the parameter *a* in *foo*. We put this variable into the worklist of procedure *foo* and Algorithm 2 is used for computing other LIVE Variables. The LIVE_Up function is defined as follows:

Algorithm 3 Pseudo code for Live-Propagation Slicing Algorithm (LPSA)

```
1: input: An MPI program M
2: output: Program slice S(M) and marked information
3: For each procedure P: Build UD and MD Chains
4: For each procedure P: Build Comm Set C(P)
5: MOD/REF analysis over the PCG
6: For each procedure P: Refine UD and MD chains
7: For each procedure P: WL[P] ← C(P)
8: For each procedure P: LIVE[P] ← ∅
9: Change ← True
10: while (Change = True) do
11:     Change ← False                                   ▷ Top-Down Phase
12:     for procedure P in Pre-Order over PCG do
13:         call INTRA-LIVE-EXT(P)
14:         for Q ∈ successor(P) do
15:             for parameter v at callsite ℓ do
16:                 if (ℓ, v) ∈ LIVE[P] then
17:                     L = LIVE_Down(P, ℓ, v, Q)
18:                     for ℓ′ ∈ L do
19:                         if ℓ′ ∈ MPI_Routines then
20:                             Mark MPI statement ℓ′
21:                         end if
22:                         S(Q) = S(Q) ∪ {ℓ′}
23:                         for x ∈ USE[ℓ′] do
24:                             WL[Q] ← {(ℓ′, x)} ∪ WL[Q]
25:                         end for
26:                     end for
27:                 end if
28:             end for
29:         end for
30:     end for
31:                                                      ▷ Bottom-Up Phase
32:     for procedure Q in Post-Order over PCG do
33:         call INTRA-LIVE-EXT(Q)
34:         for P ∈ predecessor(Q) do
35:             for formal parameter ℓ0 in Q do
36:                 if ( (ℓ0, v) ∈ LIVE[Q] ) then
37:                     (ℓ′, x) = LIVE_Up(Q, ℓ0, v, P)
38:                     WL[P] ← {(ℓ′, x)} ∪ WL[P]
39:                     S(P) = S(P) ∪ {ℓ′}
40:                 end if
41:             end for
42:         end for
43:     end for
44: end while
45: For each procedure P: return S(P)
```

Definition II.8 (LIVE Up). *Procedure Q is invoked by procedure P, v is a LIVE Variable and also a formal parameter (the label of procedure entry point is* ℓ_0 *in procedure Q,* v' *is the corresponding actual parameter in procedure P at callsite* ℓ', $LIVE\_Up(Q, \ell_0, v, P)$ *returns the label of the callsite and the actual parameter pair:* $LIVE\_Up(Q, \ell_0, v, P) = (\ell', v')$.

The final algorithm for program slicing based on live-propagation is given in Algorithm 3 that invokes Algorithm 2. The output of LPSA is the program slice set as well as directives for MPI routines. Our experimental results show that LPSA can converge within 3-4 iterations for the outer loop. Let $C(M)$ be the slicing criterion for a given MPI program M; and $S(M)$ be the slice set computed by LPSA. Then the correctness of the algorithm can be stated by Theorem II.9. A sketch of the proof of this theorem is given in [25].

Theorem II.9. $S(C(M)) = S(M)$

C. Implementation and Evaluation

We have implemented LPSA for FACT in the production compiler, Open64 [27]. Open64 is the open source version of the SGI Pro64 compiler under the GNU General Public License (GPL). As shown in Figure 6, the major functional modules of Open64 are the front end (FE), pre-optimizer (PreOPT), inter-procedural analysis, loop nest optimizer (LNO), global scalar optimizer (WOPT) and code generator (CG). To exchange data between different modules, Open64 utilizes a common intermediate representation (IR), called WHIRL. WHIRL consists of five levels of abstraction, from very high level to lower levels. Each optimization module works on a specific level of WHIRL.

FACT is implemented in the PreOPT and inter-procedural analysis modules as shown in Figure 6. In the PreOPT phase, the CFG is created for each procedure. Control dependence analysis is carried out on the CFG in reverse dominator tree order while the data dependence is collected into the DU and UD chains. The inter-procedural analysis module can be further divided into three main phases: Inter-Procedural Local Analysis (IPL), Inter-Procedural Analyzer (IPA), and Inter-Procedural Optimizer (IPO). During the IPL phase, we parse the WHIRL tree and identify the communication routines. Communication dependence is collected into MD chains and Comm Variables are stored in the form of summary data. During the IPA phase, the PCG is constructed. MOD/REF analysis is performed on this and the DU and UD chains built in the IPL phase are refined. MD chains are extended to consider cross-procedural communication dependence. By solving an inter-procedural data flow equation during the IPA phase, we compute the LIVE, sets and slice sets for each procedure and mark necessary MPI statements. During the IPO phase, we delete all the statements that are not in slice sets except MPI routines and remove the variables that are not in the LIVE, sets from the symbol table. The marked information for MPI communication routines are retained in the program slice. Currently, we only support Fortran programs in FACT and supporting other languages remains as our future work.

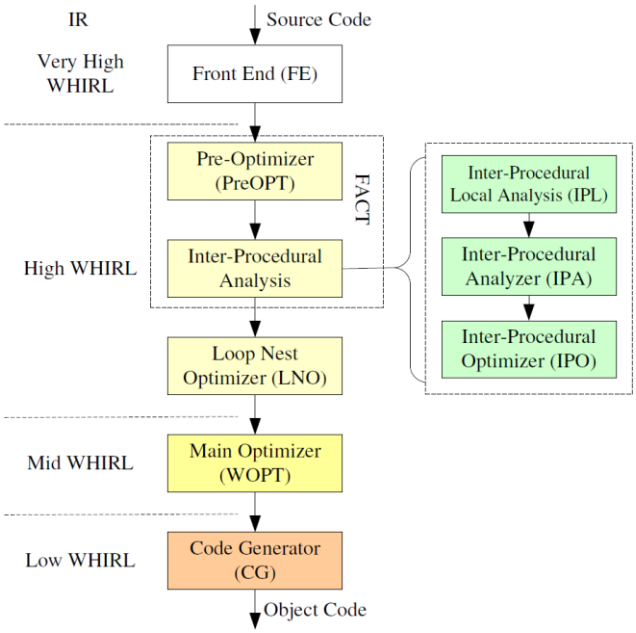

Fig. 6: FACT in the Open64 infrastructure.

Runtime environment is in charge of collecting communication traces from the program slice. It provides a custom MPI wrapper communication library which differentiates MPI routines based on their functions. For MPI routines used to create and shut down MPI runtime environment, such as *MPI_Init*, *MPI_Finalize*, they are not modified and executed directly in the library. For MPI routines used to manage communication contexts, such as *MPI_COMM_Split*, *MPI_COMM_Dup*, the library requires executing these routines and collecting information about the relation for process translation between different communicators. For MPI routines used for message passing, such as *MPI_Send*, *MPI_Irecv*, *MPI_Bcast*, the library first judges the state of the MPI routine based on the results of LPSA analysis. If the communication routine is marked, we need to execute it and meanwhile collect communication property information. Otherwise only related information is recorded. In addition, for unmarked non-blocking communication routines, the parameters *request* of these routines are set so that the library guarantees that corresponding communication operations are not executed, such as *MPI_Wait* or *MPI_Waitall*.

We use the MPI profiling layer (PMPI) to capture the communication events, record the communication traces to a memory buffer, and eventually write them on local disks. Figure 7 gives an example of collecting the communication traces for *MPI_Send* routine. In Figure 7, *myid* is a global variable computed with *PMPI_Comm_rank*. Our runtime environment also provides a series of communication trace analyzers which can generate the communication profiles of applications, such as the distribution of message sizes, communication topology graph.

To present the advantages of FACT over the traditional trace collection methods, we collect communication traces of NPB programs (Class D) and Sweep3D (150×150×150) with a large date set on a small-scale system, the 4-node *test platform* which has only 32 GB memory in all. The memory requirements of these programs except EP and LU for 512 processes exceed the memory capacity of the *test platform*. For example, the NPB FT with Class D input for 512 processes will consume about 126 GB memory size. Therefore, the traditional trace collection methods cannot collect the communication traces on such a small-scale system due to the memory limitation.

```
MPI_Send(buf, count, datatype, dest, tag,comm){
    If the routine is marked by LPSA
      PMPI_Send(buf, count, datatype, dest, tag, comm)
    Endif
    typesize = PMPI_Type_size(datatype)
    Record the following information:
      message type : MPI_Send
      message source : myid
      message destination : dest
      message size : typesize * count
      message tag : tag
      communicator ID : comm
}
```

Fig. 7: Pseudo code for collecting the communication traces for *MPI_Send* routine at runtime.

Our experimental results shown in Figure 8 demonstrate that FACT is able to collect the communication traces for these programs on the *test platform*. Moreover, it consumes very little memory resources. The memory requirements of the original programs are collected on the *validation platform*. In most cases, the memory consumption for collecting the communication traces with FACT is reduced by two orders of magnitude compared to the original programs. For example, Sweep3D only consumes 0.13 GB memory for 64 processes, 1.25 GB memory for 512 processes with FACT while the original program consumes 26.61 GB and 213.83 GB memory respectively.

Figure 9 lists the execution time of FACT when collecting the communication traces on the *test platform*. As the traditional trace collection methods cannot collect the communication traces on the *test platform*, the execution time of the original programs is collected on the *validation platform*. In addition, as the problem size is fixed for each process in the Sweep3D, its execution time increases with the number of processes.

Since FACT deletes irrelevant computations of the original program at compile time and only executes necessary communication operations at runtime, the execution time of the original program can be reduced significantly. For example, FACT just takes 0.28 seconds for collecting the communication traces of BT for 64 processes, while the original program running on the 512-core *validation platform* takes 1175.65 seconds yet. As few of communication operations are used in the EP program, its execution time is negligible after slicing. Overall, the execution time with FACT is acceptable for most developers to study the communication patterns on such a small-scale system. In addition, FACT can benefit when more nodes are available.

III. BASE PERFORMANCE PREDICTION FRAMEWORK

We use a trace-driven simulation approach for the performance prediction. In our framework, we split the parallel applications into computation and communication parts, predict computation and communication performance separately and finally use a simulator to convolute them to get the execution time of whole parallel applications. The framework includes the following key steps.

Collecting computation and communication traces: We generate communication traces of parallel applications by intercepting all communication operations for each process, and mark the computation between communication operations as sequential computation units. The purpose of this step is to separate communications and computation in parallel applications to enable us to predict them separately. Figure 12 shows a simple MPI program and its computation and communication traces are given in Figure 11(a) when the number of processes is 2 (The elapsed time for the k^{th} computation unit of process x is denoted by CPU_Burst(x,k)).

It should be noted that we only need the communication information (e.g. message type, message size, source and destination etc.) and the interleave of communication / computation in this step. All temporal properties are not used in later steps of performance prediction. A common approach to generating these traces is to execute parallel applications with instrumented MPI libraries. To further reduce the overhead in this step, we employ the FACT technique which can generate traces of large-scale applications on small-scale systems fast.

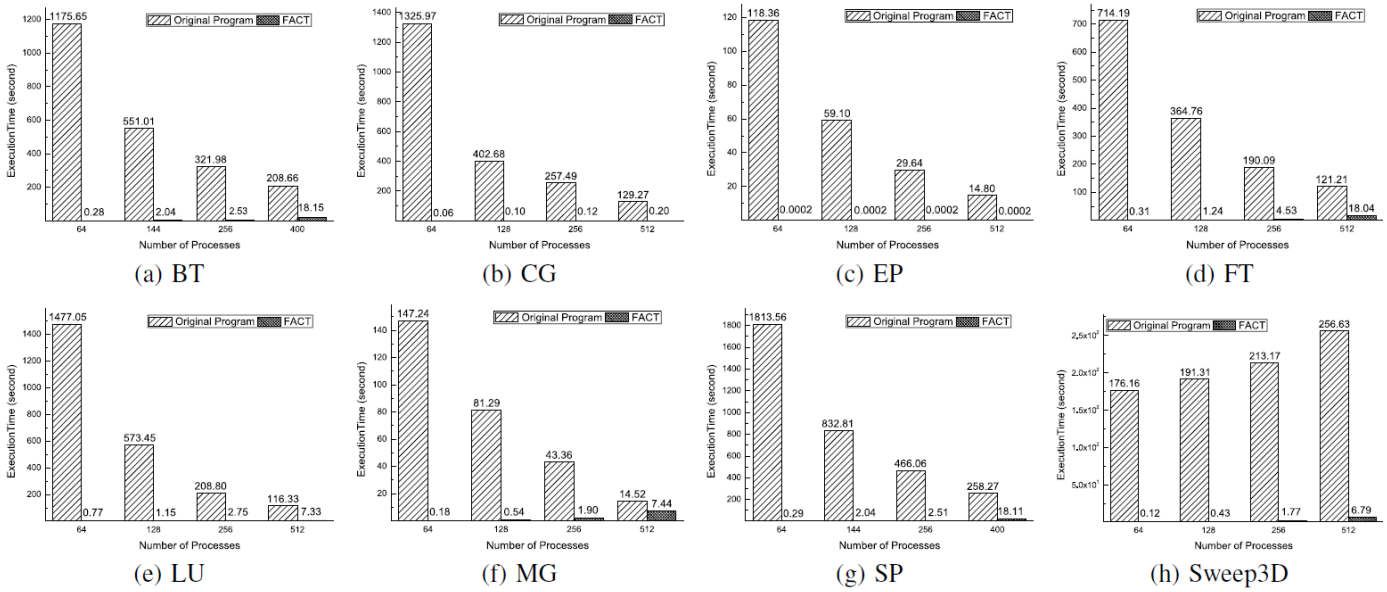

Fig. 8: The memory consumption (in **GigaByte**) of FACT for collecting the communication traces of NPB programs (Class D) and Sweep3D (150×150×150) on the *test platform*. Traditional trace collection methods cannot achieve this on the *test platform* due to the memory limitation. The memory consumption of the original programs are collected on the *validation platform*.

Obtaining sequential computation time for each process: The sequential computation time for each MPI process is measured through executing each process separately on a node of the target platform with deterministic replay techniques. For now, we just assume that we obtain the accurate computation time for

each MPI process which can be filled into the traces generated in step 1. Figure 11(b) shows obtained sequential computation time for process 0 of the program in Figure 12.

Using a trace-driven simulator to convolute communication and computation performance: Finally, a trace driven simulator, called SIM-MPI [28], is used to convolute communication and computation performance. As shown in Figure 11(c), the simulator reads trace files generated in step 1, the sequential computation time obtained in step 2, and network parameters of target platforms, to predict the communication performance of each communication operation and convolute it with sequential computation time to predict the execution time of the whole parallel application on the target platform. Our SIM-MPI is similar to DIMEMAS [9], but with a more accurate communication model, called LogGPO, which is an extension of LogGP model [29]. It can model the overlap between computation and communication more accurately than existing communications models. Our simulator is a trace-driven simulator which should be fed with communication event trace before predicting the performance of parallel programs. Event trace is a time sequenced stream of event records. To get the communication event trace in parallel program based MPI policy, MPI communication library should be instrumented. We write our own MPI wrapper library with the MPI profile interface. The operational flow of our simulator is shown in Figure 9. We enumerate all the processes in a loop. When a process *i* is current process, we simulation a MPI routine of it, by calling a corresponding function defined in protocol level such as *MPI_Send*, *MPI_Waitall* and *MPI_Finalize*.

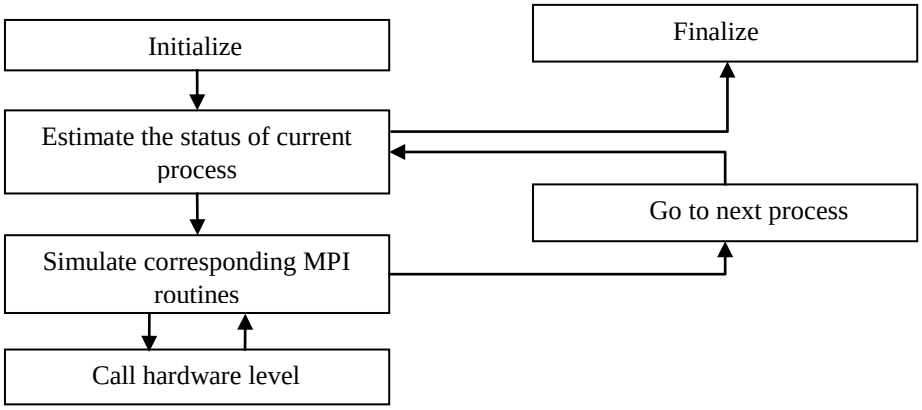

Fig. 9: The flow chart of our SIM_MPI

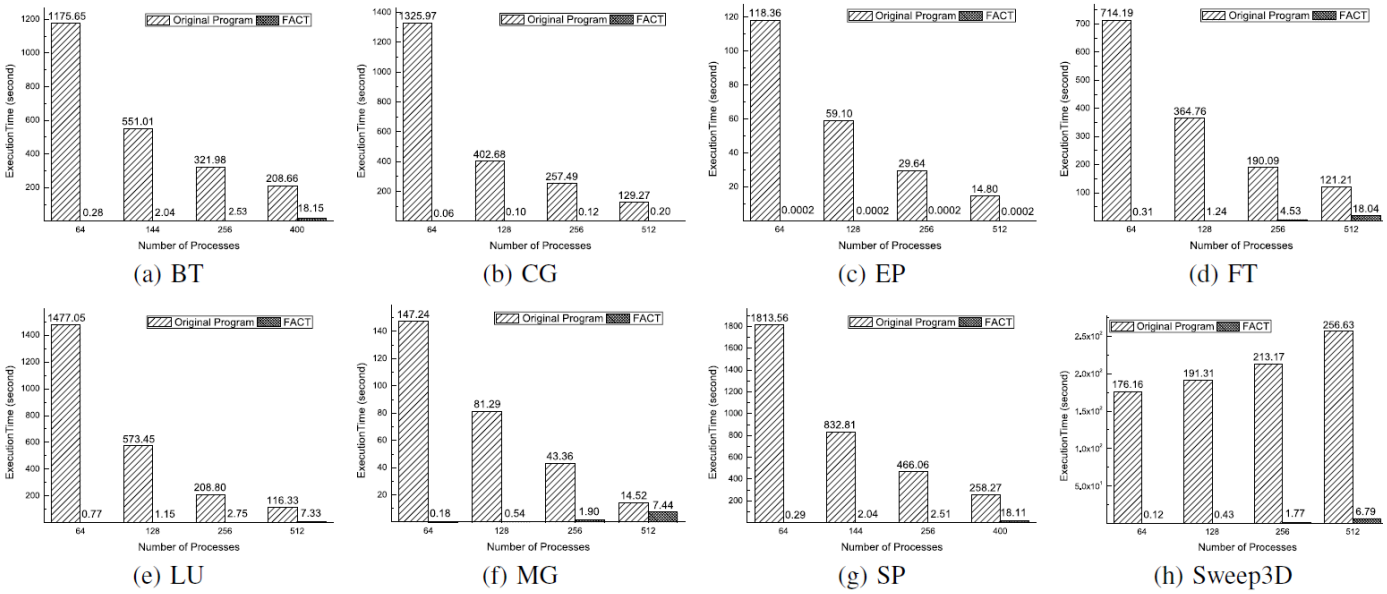

Fig. 10: The execution time (**in Second**) of FACT when collecting the communication traces of NPB programs and Sweep3D on the *test platform* (32 cores). The execution time of original programs are collected on the *validation platform* (512 cores).

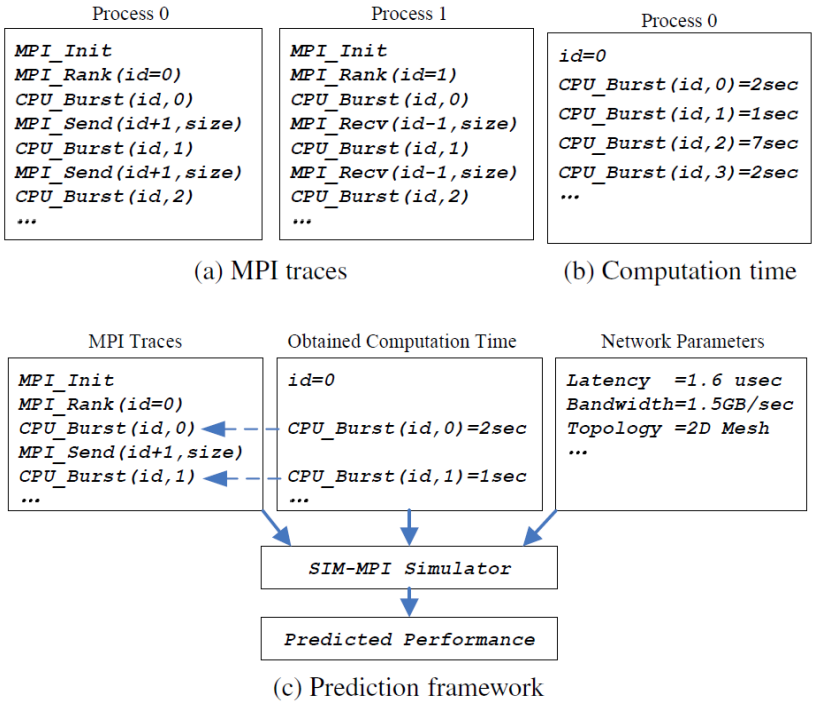

Fig. 11: Base performance prediction framework

***Definition and Obtaining Sequential Computation Time*:** Our approach is based on data-replay techniques to acquire sequential computation time. Our approach of replay-based requires two platforms. One is the host platform, which is used to collect the message logs of applications like traditional data-replay techniques. The other is one single node of the target platform on which we want to predict performance. For homogeneous HPC systems, just one node of the target platform is sufficient for our approach. More nodes of the target platform can be used to replay different processes in parallel. If the target platform is heterogeneous, at least one node of each architecture type is needed. The main steps of our approach to acquiring sequential computation time of applications include:

```
real A(MAX,MAX), B(MAX,MAX), C(MAX,MAX),
buf(MAX,MAX)
call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid...)
DO iter=1, N
   if (myid .gt. 0) then
      call MPI_RECV(buf(1, 1),num,MPI_REAL,myid-1,...)
   endif
   DO i=1, MAX
      DO j=1, MAX
         A(i,j)=B(i,j)*C(i,j)+buf(i,j)
      END DO
   END DO
   if (myid .lt. numprocs-1) then
      call MPI_SEND(A(1, 1),num,MPI_REAL,myid+1,...)
   endif
END DO
call MPI_FINALIZE(rc)
```

Fig. 12: An example of Fortran MPI program.

(1) *Building message-log database*: Record all necessary information as in the data-replay tools when executing the application on the host platform and store these information to a message-log database. This step is only done once and the message-log database can be reused in the future prediction.

(2) *Replaying each process separately*: Replay each process of the application on the single node separately and collect the elapsed time for each sequential computation unit.

To accurately measure the effects of resource contention on application performance, we propose the strategy of concurrent replay in PHANTOM. During the replay phase, we replay multiple processes simultaneously according to the number of processes running on one node of the target platform. Thus, the effects of resource contention can be captured during the concurrent execution.

IV. PHANTOM FRAMEWORK

We implement a performance prediction framework for parallel applications based on *representative replay*, called PHANTOM. In fact, *representative replay* can be used in other prediction framework for improving prediction accuracy, such as PERC and macro-level simulation [30, 31]. PHANTOM is an automatic tool chain which requires little manpower for understanding the algorithm and implementation of the parallel application. Figure 13 gives an overview of PHANTOM. PHANTOM consists of three main modules, *CompAna*, *CommAna* and *NetSim*.

TABLE I: Parallel platforms used in the evaluation.

System	**Explorer**	**Dawning**	**DeepComp-F**	**DeepComp-B**
CPU type	Intel E5345	AMD 2350	Intel X7350	Intel E5450
CPU speed	2.33 GHz	2.0 GHz	2.93 GHz	3.0 GHz
#cores/node	8	8	16	8
#nodes	16	32	16	128
mem/node	8 GB	16 GB	128 GB	32 GB
Network	Infiniband	InfiniBand	InifiniBand	InfiniBand
Shared FS	NFS	NFS	StorNext	StorNext
OS	Linux	Linux	Linux	Linux

Table I gives a description of parallel platforms used in our evaluation. *Explorer platform* serves as the host platform used to collect message logs of applications. The other three platforms are used to validate the prediction accuracy of our approach.

We evaluate our approach with 6 NPB programs [32], BT, CG, EP, LU, MG, SP and ASCI Sweep3D (S3) [33]. The version of NPB is 3.3 and input data set is Class C. For Sweep3D, both execution modes are used, strong-scaling and weak-scaling modes. For strong-scaling mode (S3-S), the total problem size of 512×512×200 is used. For weak-scaling mode (S3-W), the problem size is 100×100×100 which is fixed for each process.

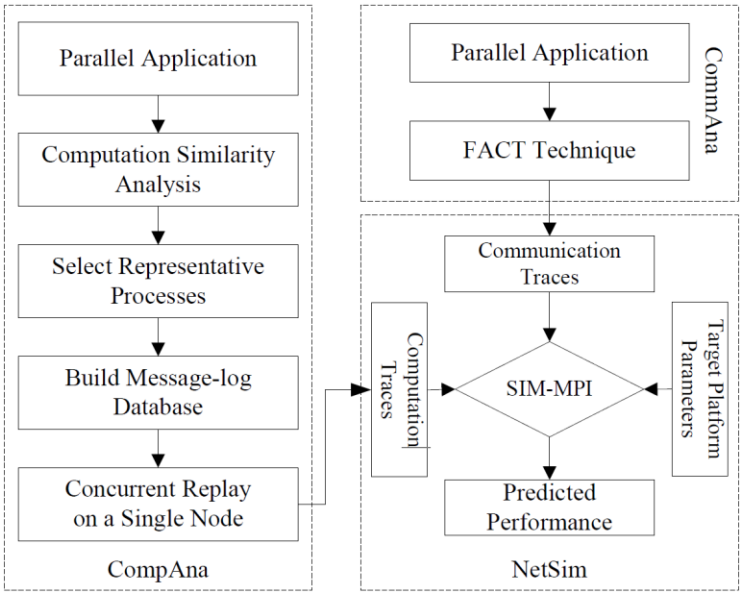

Fig. 13: Overview of PHANTOM

We compare the predicted time using PHANTOM with a recent prediction approach proposed by Barnes *et al.* (regression-based model) [34]. This model predicts the execution time T of a given parallel

application on p processors by using several instrumented runs of this program on q processors, where $q \in \{2,..., p_0\}, p_0 < p$. Through varying the values of the input variables $(x_1, x_2,..., x_n)$ on the instrumented runs, this model aims to calculate coefficients $(\beta_0,..., \beta_n)$ by linear regression fit for $\log_2(T)$.

$$\log_2(T) = \beta_0 + \sum_{i=1}^{n} \beta_i \log_2(x_i) + g(q) + error$$

In this model $g(q)$ can be either a linear function or a quadratic function for the number of processors, q. Once these coefficients are determined, above equation can be used to predict the application performance on p processors. In this chapter, we use three different processor configurations for training set: $p_0 = 16$, $p_0 = 32$ and $p_0 = 64$. For each program, we predict the performance with two forms of $g(q)$ function given by the authors, and the best results are reported.

In PHANTOM, all the sequential computation time of representative processes is acquired using a single node of the target platform. The network parameters needed by SIM-MPI are measured with micro-benchmarks on the network of the target platform. In this chapter, error is defined as $(measured\ time - predicted\ time) / (measured\ time \times 100)$ and all the experiments are conducted for 5 times.

We predict the performance for Sweep3D on three target platforms. The real execution time is measured on each target platform to validate our predicted results. All the message logs are collected on Explorer platform. As shown in Figure 14, PHANTOM can get high prediction accuracy on these platforms. Prediction errors on Dawning, DeepComp-F and DeepComp-B platforms are on average 2.67%, 1.30% and 2.34% respectively, with only -6.54% maximum error on Dawning platform for 128 processes. PHANTOM has a better prediction accuracy as well as greater stability across different platforms compared to the regression-based approach. For example, on the DeepComp-B platform, the prediction error for PHANTOM is 4.53% with 1024 processes, while 23.67% for regression-based approach (p_0 =32, 64, 128 used for training). Note that while Dawning platform has lower CPU frequency and peak performance than DeepComp-F platform, it has better application performance before 256 processes. DeepComp-B platform presents the best performance for Sweep3D among three platforms.

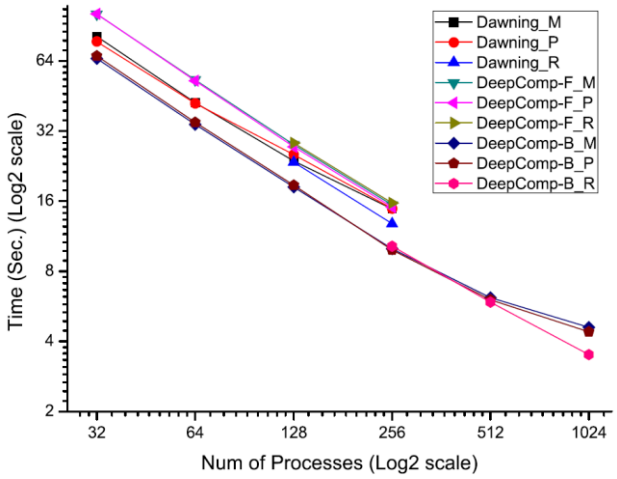

Fig. 14: Performance prediction for Sweep3D on *Dawning*, *DeepComp-F* and *DeepComp-B* platforms (*M* means the real execution time, *P* means predicted time with PHANTOM, *R* means predicted time with Regression-based approach).

Figure 15 demonstrates the prediction results with both PHANTOM and the regression-based approach for seven programs on the Dawning platform. As shown in Figure 15, the agreement between the predicted execution time with PHANTOM and the measured time is remarkably high. The prediction error with PHANTOM is less than 8% on average for all the programs. Note that EP is an embarrassing parallel program, which does not need communications. Its prediction accuracy actually reflects the accuracy of sequential computation time acquired with our approach. For EP, the prediction error is only 0.34% on average. Table II also lists the prediction errors with PHANTOM and the regression-based approach (Due

to the limitations of the regression-based approach, only the performance of applications with p processes ($p > p_0$) can be predicted.). The prediction accuracy with PHANTOM is much higher than that with the regression-based approach for most programs.

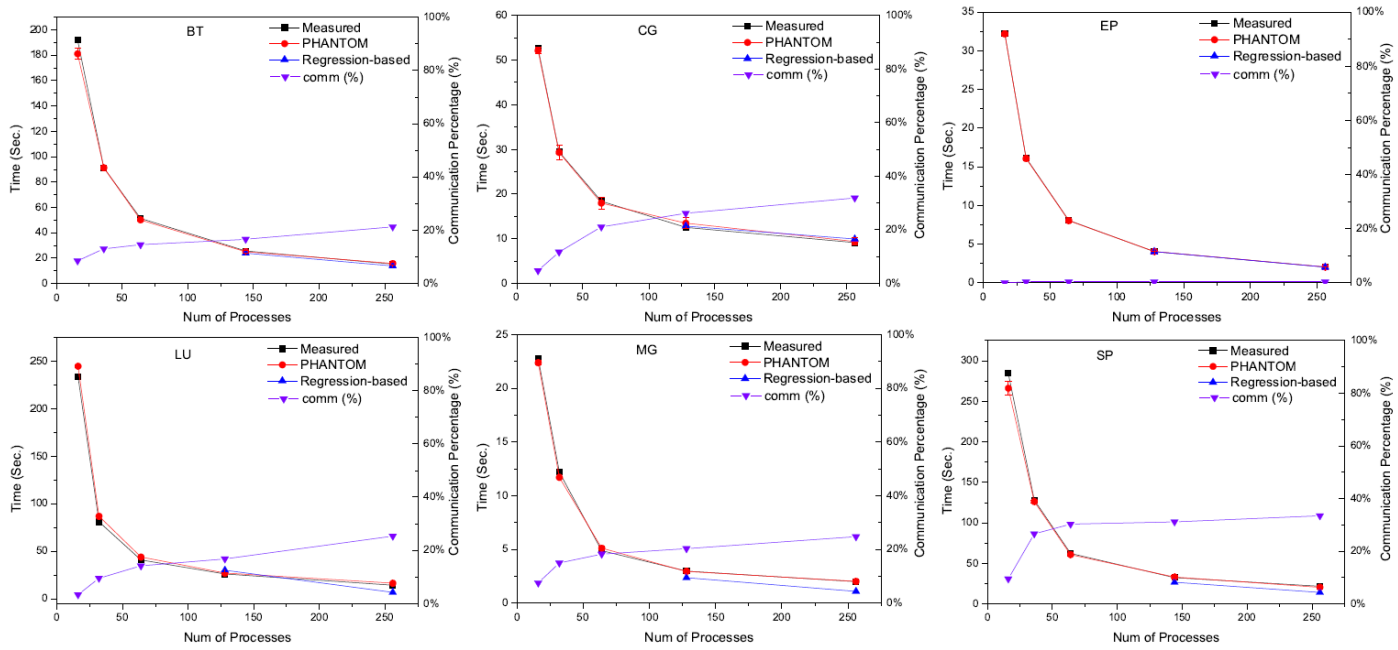

Fig. 15: Predicted time with PHANTOM compared with that with Regression-based approach on *Dawning Platform*. *Measured* means the real execution time of applications. *comm* means communication time percentage of total execution time.

TABLE II: Prediction errors (%) with PHANTOM (P.T.) vs. Regression-Based approach (R.B.) on *Dawning platform*.

Proc. #		BT	CG	EP	LU	MG	SP	S3-S	S3-W
128	P.T.	2.22	-7.60	-0.34	-3.95	0.97	-2.29	-6.54	-0.15
	R.B.	6.32	-2.22	0.01	-15.02	20.58	17.72	1.30	-4.75
256	P.T.	-3.27	-2.65	-0.27	-14.28	-0.97	5.97	-0.52	-0.27
	R.B.	9.50	-8.95	0.76	53.20	45.32	34.38	13.41	-0.28

V. CONCLUSIONS

In this chapter, we describe a novel approach, called FACT, to acquiring communication traces of large parallel message passing applications on small-scale systems. It can preserve the spatial and volume communication attributes while greatly reducing the time and memory overhead of trace collection process. We have implemented FACT and evaluated it with several parallel programs. Experimental results show that FACT is very effective in reducing the resource requirements and collection time. In most cases, we get 1-2 orders of magnitude of improvement. With FACT, we are enabled to explore more applications of using communication patterns. For predicting performance of parallel applications, we extend existing trace-driven simulation framework by using deterministic replay techniques to measure computation time process by process on prototype of target systems. We further propose a representative replay scheme which employs similarity of computation pattern in parallel applications to reduce time of prediction significantly. The approach proposed in this chapter is a combination of operating system and performance analysis techniques. We expect this chapter will motivate more interactions between these two fields in the future.

REFERENCES

[1] S. Chodnekar, V. Srinivasan, A. S. Vaidya, A. Sivasubramaniam, and C. R. Das. Towards a communication characterization methodology for parallel applications. In *HPCA*, 1997.

[2] J. Kim and D. J. Lilja. Characterization of communication patterns in message-passing parallel scientific application programs. In *CANPC*, pages 202-216, 1998.

[3] H. Chen, W. G. Chen, J. Huang, B. Robert, and H. Kuhn. MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In *ICS*, 2006.
[4] R. Preissl, M. Schulz, D. Kranzlmuller, B. R. de Supinski, and D. J. Quinlan. Using MPI communication patterns to guide source code transformations. In *ICCS*, pages 253-260, 2008.
[5] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. M. Voelker. MPIWiz: subgroup reproducible replay of mpi applications. In *PPoPP*, pages 251-260, 2009.
[6] R. Preissl, T. Kockerbauer, M. Schulz, D. Kranzlmuler, B. R. de Supinski, and D. J. Quinlan. Detecting patterns in MPI communication traces. In *ICPP*, pages 230-237, 2008.
[7] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *IPDPS*, pages 853-865, 2002.
[8] Intel Ltd. Intel trace analyzer & collector. http://www.intel.com/cd/software/products/asmo-na/eng/244171.htm.
[9] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. DiP: A parallel program development environment. In *Euro-Par*, pages 665-674, 1996.
[10] B. Mohr and F. Wolf. KOJAK-A tool set for automatic performance analysis of parallel programs. In *Euro-Par*, 2003.
[11] W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. Supercomputer, 12(1), Jan. 1996.
[12] S. Shende and A. D. Malony. TAU: The tau parallel performance system. International Journal of High Performance Computing Applications, 20(2), 2006.
[13] Jidong Zhai, Jianfei Hu, Xiongchao Tang, Xiaosong Ma, Wenguang Chen. CYPRESS: Combining Static and Dynamic Analysis for Top-Down Communication Trace Compression. In *SC*. Nov. 2014.
[14] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J.Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *SC*, pages 37-48, 2001.
[15] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, 1995.
[16] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *PPoPP*, pages 123–132, 2001.
[17] J. Zhang, J. Zhai, W. Chen, and W. Zheng. Process mapping for mpi collective communications. In Euro-Par, 2009.
[18] N. Choudhury, Y. Mehta, and T. L.W. *et al*.. Scaling an optimistic parallel simulation of large-scale interconnection networks. In WSC, pages 591-600, 2005.
[19] M. Weiser. Program slicing. IEEE Transactions on Software Engineering, 10(4):352-357, 1984.
[20] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst., 12(1):26-60, 1990.
[21] A. W. Appel. Modern Compiler Implementation in C: Basic Techniques. Cambridge University Press, New York, USA, 1997.
[22] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
[23] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst., 9(3):319-349, 1987.
[24] J. Banning. An efficient way to find side effects of procedure calls and aliases of variables. In *POPL*, pages 29-41, 1979.
[25] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
[26] Tsinghua University. Proof of live-propagation slicing algorithm. http://www.hpctest.org.cn/paper/Thu-HPC-TR20090717.pdf, 2009.
[27] SGI. Open64 compiler and tools. http://www.open64.net.
[28] Tsinghua University. SIM-MPI simulator. http://www.hpctest.org.cn/resources/sim-mpi.tgz.
[29] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the logp model for parallel computation. Journal of Parallel and Distributed Computing, 44(1): 71-79, 1997.
[30] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for application performance modeling and prediction. In *SC*'02, pages 1-17, 2002.
[31] R. Susukita, H. Ando, and M. A. et al. Performance prediction of large-scale parallell system and application using macro-level simulation. In *SC*, pages 1-9, 2008.

[32] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, 1995.
[33] LLNL. ASCI purple benchmark. https://asc.llnl.gov/computing resources/purple/archive/benchmarks.
[34] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *ICS*, pages 368-377, 2008.