Efficiently Acquiring Communication Traces for Large-Scale Parallel Applications

Jidong Zhai, Tianwei Sheng, Jiangzhou He, Wenguang Chen, and Weimin Zheng

Abstract—Communication patterns of parallel applications are important to optimize application performance and design better communication subsystems. Communication patterns can be extracted from communication traces. However, existing approaches to generate communication traces need to execute the entire parallel applications on full-scale systems that are time consuming and expensive. We propose a novel technique, called FACT, which can perform Fast Communication Traces 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 the communication traces. Our idea is based on an observation that most computation and message contents in parallel applications are not relevant to their spatial and volume communication attributes, and therefore can be removed for the purpose of communication trace collection. We have implemented FACT and evaluated it with NPB programs and Sweep3D. The results show that FACT can reduce resource consumptions by two orders of magnitude in most cases.

Index Terms—Communication pattern, communication trace, message-passing program, parallel application.

1 INTRODUCTION

Communication patterns are a key factor affecting the performance of message-passing parallel applications. Different applications 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]. For example, with the knowledge of spatial and volume communication attributes, MPIPP [3] optimizes the performance of Message-Passing Interface (MPI) programs on nonuniform 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 preestablish connections and eliminate the runtime overhead of path establishment [4].

In this paper, we focus on MPI-based parallel applications due to their popularity, but our approach can be applied to other message-passing parallel programs.

Previous work on communication patterns of parallel applications mainly relies on traditional trace collection methods [2], [5], [6]. A series of trace collection and analysis tools have been developed, such as ITC/ITA, KOJAK, TAU, and VAMPIR [7], [8], [9], [10]. 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 [5]. However, traditional communication trace collection methods have two main limitations:

- **Huge resource requirement.** Typically, parallel applications are designed to solve complex scientific computational problems and tend to consume huge computing power and memory. For example, ASCI SAGE routinely runs on 2,000 to 4,000 processors [11] and FT program in NAS Parallel Benchmark (NPB) consumes more than 600 GB of memory for Class E input [12]. Therefore, it is impossible to use traditional trace collection methods to collect communication patterns of large-scale parallel applications without full-scale systems.

- **Long trace collection time.** Although traditional trace collection methods do not introduce significant overhead to collect communication traces, they do require to execute the entire parallel applications from the beginning to the end. This results in very long trace collection time. Again, we use ASCI SAGE as an example, which 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.

We have two observations on existing communication trace collection and analysis approaches: 1) Many important applications of communication pattern analysis, such as the process placement optimization [3] and subgroup replay [13], do not require temporal attributes. 2) 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 expect to address the following problem in this paper: If we can tolerate missing the temporal attributes in communication traces, can we find a way to collect communication traces which still include all spatial and volume attributes in a more efficient way? For purposes of illustration, we use communication patterns in the rest parts of the paper to represent spatial and volume attributes of communications.

We propose a novel technique, 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 interprocedural data flow equation, it can identify all the variables and statements affecting the communication patterns.

We have implemented FACT and evaluated it with NPB programs and 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. For example, FACT of original programs and reduce resource consumptions by preserving the spatial and volume communication attributes of original programs.

The compilation framework is divided into two phases, intraprocedural analysis followed by interprocedural analysis. The program is sliced based on the results of the intraprocedural analysis. Finally, the communication traces are collected in the runtime environment.

During the intraprocedural 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 (cd), data dependence (dd), and communication dependence (md) for each procedure is gathered, which will be explained in detail in Section 3.2. During the interprocedural analysis phase, the program call graph (PCG) is built based on the information of call sites collected during the intraprocedural phase. LPSA is used to identify all the variables and statements that affect 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. The 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 with 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.

Fig. 2 uses an example to illustrate the differences between the sliced program and the original program. The example program is a parallel matrix-matrix multiplication program \( C = A \times B \) based on the domain decomposition algorithm. The main differences after slicing in FACT are as follows:

1. Line 4. The declaration of arrays A, B, and C is replaced with dummy arrays at Line 5.
2. Lines 14-20. The source codes for initializing matrices A and B are deleted.
3. Lines 41-49. The main computation codes for the matrix multiplication on each worker task are deleted.
4. Lines 23 and 31. The value of the variable offset is not relevant to the communication pattern and these two lines are deleted.
5. Additional directives for usage at runtime are added for MPI routines at Lines 7, 8, 24, 26, 32, 37, 39, and 50 (M means marked and U means unmarked).2

The sliced program is linked with the custom communication library and the communication traces are collected at runtime. At runtime, the library will judge the state of each MPI communication routine based on the directives. In this example, six unmarked communication routines at

2. The marked MPI routines will be executed at runtime and the unmarked will not be executed at runtime. Specific definitions of them will be given in Section 3.
Lines 24, 26, 32, 37, 39, and 50 will not be executed at runtime, since the contents of these messages do not affect the communication patterns.

3 Live-Propagation Slicing Algorithm

Program slicing was first proposed by Weiser [14]. From a formal point of view, the definition of program slice is based on the concept of slicing criterion. 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.

3.1 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 exists), and communicator id.

Message size, source, destination, and type are used to compute spatial and volume communication attributes, while message tag and communicator id are useful for other communication analysis. In an MPI program, these properties can be acquired directly from the corresponding parameters of the MPI communication routines. Comm Variable is defined in this paper to represent those parameters that determine the communication patterns of MPI programs directly.

**Definition 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 Fig. 3, Comm Variables in the routine for MPI_Send are marked. For each procedure \(P\), we use a Comm Set, \(\mathcal{C}(P)\), to record all the Comm Variables.

\[
\mathcal{C}(P) = \{ (\ell, v) \mid \ell \text{ is the unique label of } v, v \text{ is a Comm Variable} \}
\]

For example, the Comm Set for the parallel matrix multiplication program in Fig. 2 is (note that we use the line number of the variable as its unique label):

\[
\mathcal{C}(P) = \{ (7, myid), (8, nprocs), (24, N), (24, dest), (25, tag), (26, size), (27, dest), (27, tag), (32, size), (33, source), (33, tag), (37, N), (37, master), (37, tag), (39, size), (39, master), (39, tag), (50, size), (50, master), (51, tag) \}
\]

The Comm Set \(\mathcal{C}(P)\) is the slicing criterion for simplifying the original program in LPSA.
3.2 Dependence of MPI Programs

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

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

**Data dependence.** Data dependence between statements means that the program’s computation might be changed if the relative order of statements were reversed. 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 [15] to calculate the reaching definitions. The data flow graph (DFG) can be constructed based on the results of the reaching definitions analysis. The data dependence information computed through the reaching definitions is stored in the data structures of def-use (DU) and use-def (UD) chains [16].

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

**Example 1.** The DU chain for (10, size) and UD chain for (32, size) in Fig. 2 are DU(10, size) = \{26, 32, 39, 50\}, UD(32, size) = \{10\}.

We can further optimize the Comm Set based on the results of data flow analysis. If there are no other intervening definitions for consecutive Comm Variables, we keep only the last Comm Variable. Therefore, the Comm Set for the program in Fig. 2 can be optimized as: \( \mathbb{C}(P) = \{(7, n/myid), (8, n/procs), (27, dest), (33, source), (37, N), (50, size), (50, master), (51, tag)\} \).

**Control dependence.** If a statement \( X \) determines whether a statement \( Y \) is executed, the statement \( Y \) is control dependent on the statement \( X \). For example, the statement at Line 32 in Fig. 2 is control dependent on the if statement at Line 13 and the do while statement at Line 30. The DFG does not include information of control dependence. Control dependence can be computed with the postdominance frontier algorithm [17]. In this paper, 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.

**Example 2.** After converting the control dependence of Lines 13 and 30 into data dependence, the UD chain for size at Line 32 in Fig. 2 is: UD(32, size) = \{10, 13, 30\}.

**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 3 (Communication Dependence).** Statement \( x \) in process \( i \) is communication dependent on statement \( y \) in process \( j \), if a message can be transferred from process \( j \) to process \( i \) through explicit MPI communication routines. \( y \) is a sending operation and \( x \) is a receiving operation.

For example, in Fig. 2, MPI_Recv routine at Line 37 is communication dependent on the MPI_Send routine at Line 24. In MPI programs, the message sources and tags can be set to wildcard. As a result, one receiving routine can be matched with multiple sending routines. To address this problem, we can add a transitive closure requirement for sending-receiving and receiving-sending pairs to take the union of all possible sending routines into account.

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.

In MPI programs, the message is exchanged through the message buffer variable, \( buf \). The communication dependence can be represented with the message buffer variable. \( msg/buf/\ell \) is used to denote the message buffer variable in the communication statement \( \ell \).

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

**Example 3.** The MD chain for variable \( A \) at Line 37 in Fig. 2 is: \( MD(37, A) = \{24\} \). The message buffer variable in MPI communication routine of Line 24 is: \( msg/buf/24 = \{A\} \).

**Definition 5.** The slice set of an MPI program (\( M \)) with respect to the slicing criterion \( \mathbb{C}(M) \), denoted by \( S(\mathbb{C}(M)) \), consists of all statements \( \ell \) on which the values of variables in \( \mathbb{C}(M) \) directly or indirectly depend. More formally:

\[
S(\mathbb{C}(M)) = \{\ell | w \xrightarrow{d_i} \cdots \xrightarrow{d_1} \ell, w \in \mathbb{C}(M), n > 0, for 1 \leq i \leq n, d_i \in \{cd, dd, md\}\}.
\]

We use the symbol \( \rightarrow \) to denote the dependence between variables and statements. For computing the program slice with respect to the slicing criterion \( \mathbb{C}(M) \), 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 6 (LIVE Variable).** A variable \( (\ell, x) \) is LIVE, if the change of its value can affect the value of any Comm Variable \( w \) directly or indirectly through MPI programs, denoted by \( w \rightarrow^* (\ell, x) \). There is a LIVE Set for each procedure \( P \), \( LIVE[P] \). \( LIVE[P] = \{(\ell, x) | w \rightarrow^* (\ell, x), w \in \mathbb{C}(P)\} \).
3.3 Intraprocedural Analysis

During the intraprocedural 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 paper 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 \( \$ (P) \) and mark MPI statements that define LIVE Variables or have communication dependence with the 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.

**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: \( \$ (P) \)
4: Change ← False
5: while \( WL[P] \neq \phi \) do
6: Remove an item \((\ell_i, v)\) from \( WL[P] \)
7: if \((\ell_i, v) \notin LIVE[P] \) then
8: Change ← True
9: \( LIVE[P] \leftarrow \{(\ell_i, v)\} \cup LIVE[P] \)
10: /* Process communication dependence. */
11: if \((\ell_i, v) \in receive_{buf} \) then
12: for each \( \ell_j \in MD(\ell_i, v) \) do
13: Mark MPI statement \( \ell_j \)
14: \( \$ (P) = \$ (P) \cup \{\ell_j\} \)
15: \( WL[P] \leftarrow \{\ell_j, msg_{buf}(\ell_j)\} \cup WL[P] \)
16: else /* Process control and data dependence. */
17: for each \( \ell_k \in UD(\ell_i, v) \) do
18: if \( \ell_k \in MPI_{Routines} \) then
19: Mark MPI statement \( \ell_k \)
20: \( \$ (P) = \$ (P) \cup \{\ell_k\} \)
21: for each \( x \in USE[\ell_k] \) do
22: \( WL[P] \leftarrow \{\ell_k, x\} \cup WL[P] \)
23: end if
24: return \( \$ (P) \)
25: **end procedure**

**Example 4.** After the computation by Algorithm 1, the final LIVE Set for the program in Fig. 2 is:

\[ LIVE[P] = \{(7, myid), (8, nprocs), (27, dest), (33, source), (37, N), (50, size), (50, master), (51, tag), (22, nprocs), (30, nprocs), (13, myid), (13, master), (10, cols), (10, N), (9, N), (9, nprocs)\} \]

The slice set of program is: \( \$ (P) = \{3, 7, 8, 9, 10, 11, 12, 13, 22, 30\} \). The MPI routines at Lines 7-8 are marked by the algorithm.

3.4 Interprocedural Analysis

Slicing across procedure boundaries is complicated due to the necessity of passing the LIVE Variables into and out of procedures. Because the slicing criterion can arise either in the calling procedure (callee) 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 traversal over the PGC, Top-Down phase followed by Bottom-Up phase. Additionally, the UD chains built during the intraprocedural phase are refined to consider the side effects of procedure calls. In this paper, we assume that all the parameters are passed by reference and our algorithm can be extended to the case that they are passed by value.

**MOD/REF analysis.** To build precise UD chains, we use the results of interprocedural MOD/REF analysis. For example, in Fig. 4a, 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 [18]: GMOD(P) and GREF(P). 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 intraprocedural analysis. For example, \( UD(4, a) = \{3\} \).

**Extension of MD chains.** The MD chains collected during the intraprocedural phase do not include interprocedural communication dependence. During the interprocedural 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 \( \ell_j \) in procedure \( P' \).

**Algorithm 2. Extension of INTRA-LIVE \((P)\)**

1: **procedure** INTRA-LIVE-EXT\((P)\)

... 
12: for each \( P' : \ell_j \in MD(\ell_i, v) \) do
13: Mark MPI statement \( P' : \ell_j \)
14: \( \$ (P') = \$ (P') \cup \{\ell_j\} \)
15: \( WL[P'] \leftarrow \{\ell_j, msg_{buf}(\ell_j)\} \cup WL[P] \)
...
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. We define the $\text{LIVE}_\text{Down}$ function to formalize such data flow analysis.

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

**Example 5.** In Fig. 4a, after propagating the LIVE Variable $(3, a)$ from the procedure $\text{foo}$ into the procedure $\text{bar}$, we can compute $\text{LIVE}_\text{Down}(\text{foo}, 3, a, \text{bar}) = \{9\}$.

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. The $\text{LIVE}_\text{Up}$ function is defined as follows:

**Definition 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 $\ell_0$) in procedure $Q$, $v'$ is the corresponding actual parameter in procedure $P$ at callsite $\ell$, $\text{LIVE}_\text{Up}(Q, \ell_0, v, P)$ returns the label of the callsite and the actual parameter pair: $\text{LIVE}_\text{Up}(Q, \ell_0, v, P) = (\ell', v')$.

**Example 6.** In Fig. 4b, after propagating the LIVE Variable $(7, b)$ from the procedure $\text{bar}$ into the procedure $\text{foo}$, we can compute $\text{LIVE}_\text{Up}(\text{bar}, 6, b, \text{foo}) = (4, a)$.

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. Let $\mathcal{C}(M)$ be the slicing criterion for a given MPI program $M$. Let $\mathcal{S}(M)$ be the slice set computed by LPSA. Then, the correctness of the algorithm can be stated by Theorem 1. A sketch of the proof of this theorem is given in the supplemental file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.49.

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

1. **input:** An MPI program $M$
2. **output:** Program slice $\mathcal{S}(M)$ and marked information
3. For each procedure $P$: Build UD and MD Chains
4. For each procedure $P$: Build Comm Set $\mathcal{C}(P)$
5. MOD/REF analysis over the PCG
6. For each procedure $P$: Refine UD and MD chains
7. For each procedure $P$: $\mathcal{W}[P] \leftarrow \mathcal{C}(P)$
8. For each procedure $P$: $\text{LIVE}[P] \leftarrow \emptyset$
9. Change $\leftarrow$ True
10. **while** (Change = True) **do**
11. Change $\leftarrow$ False
12. /* Top-Down Phase */
13. **for each** procedure $P$ in Pre-Order over PCG **do**
14. call $\text{INTRA-LIVE-EXT}(P)$
15. **for each** $Q$ in $\text{successor}(P)$ **do**
16. **for each** parameter $v$ at callsite $\ell$ **do**
17. if $((\ell, v) \in \text{LIVE}[P])$ **then**
18. $L = \text{LIVE}_\text{Down}(P, \ell, v, Q)$
19. **for each** $\ell'$ in $L$ **do**
20. if $\ell' \in \text{MPI_Routines}$ **then**
21. Mark MPI statement $\ell'$
22. $\mathcal{S}(Q) = \mathcal{S}(Q) \cup \{v\}'$
23. **for each** $x \in \text{USE}(\ell')$ **do**
24. $\mathcal{W}[L] \leftarrow \{(\ell', x')\} \cup \mathcal{W}[L]$
25. /* Bottom-Up Phase */
26. **for each** procedure $Q$ in Post-Order over PCG **do**
27. call $\text{INTRA-LIVE-EXT}(Q)$
28. **for each** $P \in \text{predecessor}(Q)$ **do**
29. **for each** formal parameter $v$ at $\ell_0$ in $Q$ **do**
30. if $((\ell_0, v) \in \text{LIVE}(Q))$ **then**
31. $(\ell', x) = \text{LIVE}_\text{Up}(P, \ell_0, v, P)$
32. $\mathcal{W}[P] \leftarrow \{(\ell', x')\} \cup \mathcal{W}[P]$
33. $\mathcal{S}(P) = \mathcal{S}(P) \cup \{v\}'$
34. For each procedure $P$: return $\mathcal{S}(P)$

**Theorem 1.** $\mathcal{S}(\mathcal{C}(M)) = \mathcal{S}(M)$.

### 4 Evaluation

#### 4.1 Methodology

We evaluate FACT with seven NPB programs [12], BT, CG, EP, FT, LU, MG, SP, and ASCI Sweep3D (S3) [19]. We use version 3.3 of NPB and the Class D data set. In our experiments, the problem size in Sweep3D is fixed for each process $(150 \times 150 \times 150)$.

We perform our experiments on two platforms: a test platform and a validation platform. The test platform is a small-scale system with four nodes used to collect communication traces with FACT. Each node is a two-way Quad-Core with Intel Xeon E5345 2.33 GHz CPUs and 8 GB of memory, and connected with a Gigabit Ethernet. Our custom communication library is implemented based on mpich2-1.0.7 [20]. To execute more processes on test platform, we overload each node to reach the full set of processes. While the validation platform is a large-scale system with 32 nodes used to validate the communication traces collected with FACT and record memory requirements and execution time of the original programs. Each node is a four-way Quad-Core with AMD8347 1.9 GHz CPUs and 32 GB of memory, and connected with a 20 Gbps Infiniband network. MPI library is mvpach-1.1.0 [21].

#### 4.2 Validation

We give the proof for correctness of our algorithm in the supplemental file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.49. In addition, we also validate the implementation of FACT through comparing the communication traces collected by FACT with traces collected by traditional trace collection methods on the validation platform. We perform comparison with seven NPB programs and Sweep3D for different numbers of processes (64, 128, 256, and 512). The experimental results show that
they are identical except that the traces collected by FACT do not include time stamps.

Communication patterns of parallel applications can be extracted from the communication traces. Fig. 5 shows extracted communication patterns for BT, CG, LU, and MG with FACT. From the figures, we observe that the communications in these programs have high correlation with locality. Most of the communications exist between adjacent processes around the diagonal line in the communication matrix.

4.3 Performance
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 data set on a small-scale system, the 4-node test platform which has only 32 GB of memory in all. The memory requirements of these programs except EP and LU for 512 processes exceed the memory capacity of the test platform. Therefore, the traditional trace collection methods cannot collect the communication traces on such a small-scale system due to the memory limitation.

Experimental results shown in Fig. 6 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 of memory for 64 processes, 1.25 GB of memory for 512 processes with FACT while the original program consumes 26.61 and 213.83 GB of memory, respectively.

Fig. 7 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. Since FACT deletes irrelevant computations of the original programs at compile time and only executes necessary communication operations at runtime, the execution time of the original programs 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. As few communication operations are used in the EP program, its execution time is negligible after slicing.

Fig. 5. Extracted communication patterns for BT, CG, LU, and MG (CLASS = D, NPROCS = 64) with FACT. The gray level of a cell at the $i$th row and $j$th column represents the communication volume (in Byte) between two processes $i$ and $j$. (a) BT, (b) CG, (c) LU, and (d) MG.

Fig. 6. The memory consumption (in GigaByte) of FACT for collecting the communication traces of NPB programs and Sweep3D on the test platform. The memory consumption of the original programs is collected on the validation platform. (a) BT, (b) CG, (c) EP, (d) FT, (e) LU, (f) MG, (g) SP, and (h) Sweep3D.
5 LIMITATIONS AND FUTURE WORK

In this paper, we propose a novel idea to efficiently acquire communication traces for parallel applications. We have implemented a prototype system and evaluated it with NPB benchmarks and Sweep3D. However, there are some issues needed to be addressed in the future. First, we only support Fortran programs in FACT now due to limitations of alias analysis algorithm in Open64 compiler. We plan to implement LPSA in other compilers to support more languages. Second, our system can stress I/O systems even more than traditional tracing methods. We will integrate the Noeth’s work [22] into our system to compress generated trace data.

6 CONCLUSIONS

In this paper, we propose a novel approach, called FACT, to acquire communication traces of large parallel message-passing applications on small-scale systems. Our approach 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 one to two orders of magnitude of improvement. To the best of our knowledge, FACT is the first work that can collect communication traces of large-scale parallel applications on small-scale systems.

ACKNOWLEDGMENTS

The authors would like to thank the editor and the anonymous reviewers for their suggestions on improving this paper. This work was supported by the National Natural Science Foundation of China under Grant 60703055, National Laboratory for Information Science and Technology (THList) Cross-discipline Foundation, National High-Tech Research and Development Plan (863 plan) under Grants 2006AA01A105 and 2009AA01A138. A preliminary version of this work appeared in SC ’09. For more information about this work, please refer to the supplemental file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.49.

REFERENCES

Jidong Zhai received the bachelor’s degree in computer science from the University of Electronic Science and Technology of China (UESTC) in 2003, and the PhD degree in computer science and technology from Tsinghua University in 2010. He is now working as a postdoc in the Department of Computer Science and Technology, Tsinghua University. His research interests include performance evaluation for high performance computers, performance analysis, and modeling of parallel applications and compiler techniques. He has published several papers in important conferences such as SC, PPoPP, and EuroPar.

Tianwei Sheng received the bachelor’s degree from Harbin Institute of Technology in 2004 and is now working toward the PhD degree in Tsinghua University. His major research interests include program analysis, system reliability, and compiler optimizations. He is also one of the maintainers for Open64 Compiler.

Jiangzhou He received the bachelor’s degree from Tsinghua University in 2003, and is now working toward the PhD degree from the Institute of High Performance Computing of the same university. His major research interests include parallel programming model and compiler techniques. Besides, he did some work on OpenMP compilation and runtime library and structural data layout optimization. He has published some papers about structural data layout optimization.

Wenguang Chen received the BS and PhD degrees in computer science from Tsinghua University in 1995 and 2000, respectively. He was the CTO of Opportunity International, Inc. from 2000 to 2002. Since January 2003, he has been with Tsinghua University. He is now a professor and an associate head in the Department of Computer Science and Technology, Tsinghua University. His research interests are parallel and distributed computing, programming model, and mobile cloud computing.

Weimin Zheng received the master’s degree from Tsinghua University in 1982. He is now a professor in the Department of Computer Science and Technology at Tsinghua University. His research interests include parallel and distributed computing, compiler technique, grid computing, and network storage.