FACT: Fast Communication Trace Collection for Parallel Applications through Program Slicing

Jidong Zhai, Tianwei Sheng, Jiangzhou He, Wenguang Chen, Weimin Zheng
Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University, Beijing, China
{dijd03,ctw04,hejz07}@mails.tsinghua.edu.cn, {cwg,zwm-dcs}@tsinghua.edu.cn

ABSTRACT

A proper understanding of communication patterns of parallel applications is important to optimize application performance and design better communication subsystems. Communication patterns can be obtained by analyzing 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.

In this paper, 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 the communication traces. The program slice preserves all the variables and statements in the original program relevant to spatial and volume communication attributes. Our idea is based on an observation that most computation and message contents in message-passing parallel applications are independent of these attributes, and therefore can be removed from the programs 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 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 collects the communication traces of the Sweep3D for 512 processes on a 4-node (32 cores) platform in just 6.79 seconds, consuming 1.25 GB memory, while the original program takes 256.63 seconds and consumes 213.83 GB memory on a 32-node (512 cores) platform. Finally, we present an application of FACT.

Keywords: Communication Pattern, Communication Trace, Message Passing Program, Parallel Application

1. INTRODUCTION

Communication performance is 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] [10, 21]. Figure 1 presents the spatial and volume communication attributes of CG in NAS Parallel Benchmark (NPB) [4] with 64 processes.

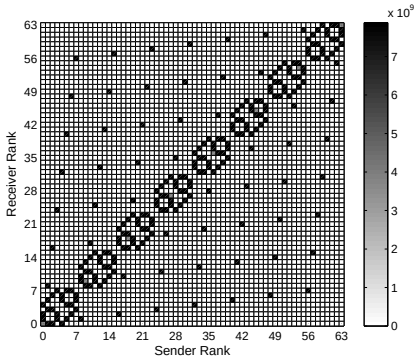


Figure 1: The communication spatial and volume attributes of NPB CG program (CLASS=D, NPROCS=64). The gray level of a cell at the x^{th} row and y^{th} column represents the communication volume (in Byte) between two processes x and y.

Proper understanding of communication patterns of parallel applications is important to optimize the communication performance of these applications [8, 29]. For example, with the knowledge of spatial and volume communication attributes, MPIPP [8] 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 [11]. Furthermore, a recent work shows spatial and volume communication attributes can be employed by replay-based MPI debuggers to reduce replay overhead significantly [41].

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 [21, 28, 38]. A series of trace collection and analysis tools have been developed, such as ITC/ITA, KOJAK, Paraver, TAU and VAMPIR [19,

[1]The communication volume is specified by the number of messages and the message size. The spatial attribute is characterized by the distribution of message source and destination. The temporal behavior is captured by the message generation rate.

SC09 November 14-20, 2009, Portland, Oregon, USA.

22, 24, 26, 33]. 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 [28]. 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 2000-4000 processors [20] and FT program in the NPB consumes more than 600 GB memory for Class E input [4]. 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 of using communication traces, such as input sensitivity analysis of communication patterns.

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 [8, 43] and subgroup replay [41], 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 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 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. For example, FACT collects the communication traces of the Sweep3D for 512 processes on a 4-node (32 cores) platform in just 6.79 seconds and using 1.25 GB memory, while the original program takes 256.63 seconds and consumes 213.83 GB memory on a 32-node (512 cores) platform.

This paper is organized as follows. Section 2 gives a discussion of related work. In Section 3, we present an overview of our approach followed by our live-propagation slicing algorithm in Section 4. Section 5 describes the implementation of FACT. Our experimental results are reported in Section 6. Section 7 presents an application of FACT. We discuss our work in Section 8. Finally, we conclude in Section 9.

2. RELATED WORK

Communication patterns of parallel applications have been studied extensively by many research groups [10, 13, 21, 38, 42]. Typically, these studies have mainly relied on instrumentation-based trace collection methods. A series of trace collection and analysis tools have been developed by both academia and industry, such as ITC/ITA [19], KOJAK [24], Paraver [22], VAMPIR [26] and TAU [33]. These tools instrument original programs and execute them to acquire communication traces. Additionally, mpiP [37] 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. However, our method only requires executing the program slice rather than the entire program. Therefore, our method can analyze large-scale applications on small-scale systems.

A few studies have tried to compute a symbolic expression of the communication patterns for a given parallel program through data flow analysis [17, 32]. Shao *et al.* proposed a technique named communication sequence to present communication patterns of applications [32]. Ho and Lin described an algorithm for static analysis of communication structures in the programs written in a channel-based message passing language [17]. Since these approaches only employ static analysis techniques trying to represent communication patterns of applications, they suffer from intrinsic limitations of static analysis. For example, they cannot deal with program branches, loops and the effects from input parameters. In fact, our approach is a hybrid method of static analysis and traditional trace collection methods. In our approach, program slicing is used to simplify the original program at compile time, and then a custom communication library is used to collect communication traces from the program slice at runtime. Therefore, our approach can address above limitations of static analysis.

Our approach exploits the technique of program slicing in the compiler. Program slicing was first proposed by Mark Weiser [40]. Traditionally, it has been used to assist in tedious and error prone tasks such as program debugging, software testing and software maintenance in sequential programs [6, 15, 16, 39]. It has also been used to hide I/O latency for parallel applications [9].

3. DESIGN OVERVIEW

FACT consists of two primary components, a compilation framework and a runtime environment as shown in Figure 2. 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.

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

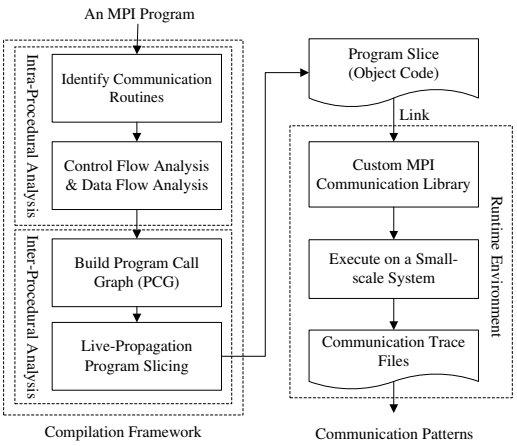


Figure 2: Overview of FACT

each procedure is gathered, which will be explained in detail in Section 4.2. 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.

Figure 3 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 problem is decomposed by assigning each worker task a number of consecutive columns of matrix B, and replicating matrix A to all tasks. Each worker task computes one or more columns of the result matrix C. Process 0 is the master task, which is in charge of distributing the matrices and collecting the results, but does not take part in the calculation. 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* has no effect on the communication patterns 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

```
1  program MM
2  include 'mpif.h'
3  parameter (N = 80)
4  real A(N,N), B(N,N), C(N,N)
5  ⇒ real A(1,1), B(1,1), C(1,1)
6  call MPI_Init(ierr)
7  [M] call MPI_COMM_Rank(MPI_COMM_WORLD,myid,ierr)
8  [M] call MPI_COMM_Size(MPI_COMM_WORLD,nprocs,ierr)
9  cols   = N/(nprocs-1)
10 size   = cols*N
11 tag    = 1
12 master = 0
13 if (myid .eq. master) then
14 C Initialize A and B
15   do i=1, N
16     do j=1, N
17       A(i,j) = (i-1)+(j-1)
18       B(i,j) = (i-1)*(j-1)
19     end do
20   end do
21 C Send matrix data to the worker tasks
22   do dest=1, nprocs-1
23     offset = 1 + (dest-1)*cols
24 [U] call MPI_Send(A, N*N, MPI_REAL, dest,
25     &       tag, MPI_COMM_WORLD, ierr)
26 [U] call MPI_Send(B(1,offset),size,MPI_REAL,
27     &       dest, tag, MPI_COMM_WORLD, ierr)
28   end do
29 C Receive results from worker tasks
30   do source=1, nprocs-1
31     offset = 1 + (source-1)*cols
32 [U] call MPI_Recv(C(1,offset),size,MPI_REAL,
33     &  source,tag,MPI_COMM_WORLD,status,ierr)
34   end do
35 else
36 C Worker receive data from master task
37 [U]call MPI_Recv(A, N*N, MPI_REAL, master, tag,
38    &             MPI_COMM_WORLD, status, ierr)
39 [U]call MPI_Recv(B,size, MPI_REAL, master, tag,
40    &             MPI_COMM_WORLD, status, ierr)
41 C Do matrix multiply
42   do k=1, cols
43     do i=1, N
44       C(i,k) = 0.0
45       do j=1, N
46         C(i,k) = C(i,k) + A(i,j) * B(j,k)
47       end do
48     end do
49   end do
50 [U]call MPI_Send(C, size, MPI_REAL, master,
51    &             tag, MPI_COMM_WORLD, ierr)
52 endif
53 call MPI_Finalize(ierr)
54 end
```

Figure 3: A parallel Fortran Matrix-Matrix Multiplication program. The source codes in the boxes are deleted after slicing and Line 4 is replaced by Line 5. And additional directives for usage at runtime are added for the MPI routines (M means marked and U means unmarked). All the variables and statements affecting the communication patterns are preserved.

[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 4.

runtime, the library will judge the state of each MPI communication routine based on the directives. In this example, six unmarked

communication routines at Lines 24, 26, 32, 37, 39 and 50 are not executed at runtime, since the contents of these messages do not affect the communication patterns.

The original program consumes about $3N^2$ memory, does $2N^3/(P-1)$ floating point computations and performs 3 times communication operations for each worker process, performs $3(P-1)$ times communication operations for the master process (where N is the size of the matrix for each dimension and P is the number of processes). These memory consumption and computation time are reduced in the sliced program. Meanwhile, the communication time is reduced at runtime.

4. LIVE-PROPAGATION SLICING ALGORITHM

From a formal point of view, the definition of program slice is based on the concept of slicing criterion [40]. A slicing criterion is a pair $\langle p, V\rangle$, where p is a program point, and V is a subset of the program variables. A program slice on the slicing criterion $\langle p, V\rangle$ 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.

4.1 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.

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 4, 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 paper to represent those parameters that determine the communication patterns directly.

DEFINITION 4.1 (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 4, 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

```
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
```

Figure 4: Comm variables in the routine for `MPI_Send`. The variables marked with Comm directly determine the communication patterns of the parallel program.

Comm Variables are collected. For each procedure P, we use a Comm Set, $\mathbb{C}(P)$, to record all the Comm Variables. $\mathbb{C}(P) = \{(\ell, v) \mid \ell$ is the unique lable of v, v is a Comm Variable$\}$. For example, the Comm Set for the parallel matrix multiplication program in Figure 3 is (note that we use the line number of the variable as its unique label): $\mathbb{C}(P) = \{(7, myid), (8, nprocs), (24, N), (24, dest), (25, tag), (26, size), (27, dest), (27, tag), (32, size), (33, source), (33, tag), (37, N), (37, master), (37, tag), (39, size), (39, master), (39, tag), (50, size), (50, master), (51, tag)\}$. The Comm Set $\mathbb{C}(P)$ is the slicing criterion for simplifying the original program in LPSA, which will be optimized during the phase of data dependence analysis.

4.2 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 [18]. 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 [2] 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 [1].

DEFINITION 4.2 (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.*

EXAMPLE 4.3. *The DU chain for (10, size) and UD chain for (32, size) in Figure 3 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

the consecutive Comm Variables, we keep only the last Comm Variable. Therefore, the Comm Set for the program in Figure 3 can be optimized as: $\mathbb{C}(P) = \{(7, myid),\ (8, nprocs),\ (27, dest),\ (33, source),\ (37, N),\ (50, size),\ (50, master),\ (51, tag)\}$.

Control Dependence If a statement X determines whether statement Y is executed, statement Y is control dependent on statement X. For example, the statement at Line 32 in Figure 3 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 post-dominance frontier algorithm [14]. 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 4.4. *After converting the control dependence of Lines 13 and 30 into data dependence, the UD chain for $size$ at Line 32 in Figure 3 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 4.5 (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 ($x \neq y$).*

For example, in Figure 3 `MPI_Recv` routine at Line 37 is communication dependent on the `MPI_Send` routine at Line 24. 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 paper.

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 Section 4.5.

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 ℓ. Additional considerations for non-blocking communications will be presented in the implementation of runtime environment.

DEFINITION 4.6 (MD CHAIN). *Message-Dependence Chain (MD Chain) links each variable of message receiving buffer to all of its sending operations.*

EXAMPLE 4.7. *The MD chain for variable A at Line 37 in Figure 3 is: $MD(37, A) = \{24\}$. The message buffer variable in MPI communication routine of Line 24 is: $msg\_buf(24) = \{A\}$.*

DEFINITION 4.8. *The slice set of an MPI program (M) with respect to the slicing criterion $\mathbb{C}(M)$, denoted by $\mathcal{S}(\mathbb{C}(M))$, consists of all statements ℓ on which the values of variables in $\mathbb{C}(M)$ directly or indirectly dependent. More formally:*

$\mathcal{S}(\mathbb{C}(M)) = \{\, \ell \,|\, \mathbb{v} \xrightarrow{d_1} ... \xrightarrow{d_n} \ell,\ \mathbb{v} \in \mathbb{C}(M),\ n > 0, for\ 1 \leq i \leq n,\ d_i \in \{cd, dd, md\}\}$

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

4.3 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 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 $\mathbb{S}(P)$ 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.

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] ≠ φ do
6:          Remove an item (ℓ_i, v) from WL[P]
7:          if (ℓ_i, v) ∉ LIVE[P] then
8:              Change ← True
9:              LIVE[P] ← {(ℓ_i, v)} ∪ LIVE[P]
10:             /* Process communication dependence! */
11:             if (ℓ_i, v) ∈ receive_buf then
12:                 for each ℓ_j ∈ MD(ℓ_i, v) do
13:                     Mark MPI statement ℓ_j
14:                     S(P) = S(P) ∪ {ℓ_j}
15:                     WL[P] ← {(ℓ_j, msg_buf(ℓ_j))} ∪ WL[P]
16:             else /* Process control and data dependence! */
17:                 for each ℓ_k ∈ UD(ℓ_i, v) do
18:                     if ℓ_k ∈ MPI_Routines then
19:                         Mark MPI statement ℓ_k
20:                     S(P) = S(P) ∪ {ℓ_k}
21:                     for each x ∈ USE[ℓ_k] do
22:                         WL[P] ← {(ℓ_k, x)} ∪ WL[P]
23:             end if
24:     return S(P)
25: end procedure
```

EXAMPLE 4.10. *After computing by Algorithm 1, the final* LIVE *Set for the program in Figure 3 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:* $\mathbb{S}(P) = \{3, 7, 8, 9, 10, 11, 12, 13, 22, 30\}$. *The MPI routines at Lines 7-8 are marked by the algorithm.*

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. For example, the six unmarked communication routines in Figure 3 do not need to transfer the messages over the network actually. We only need to collect the values of their Comm Variables at runtime. Therefore, the communication time of the original program can be significantly reduced. 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 5, 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.

```
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 }
```

Figure 5: Marked MPI point-to-point communication routines by Algorithm 1 (M means marked).

Algorithm 1 is sufficient for the MPI program with one function, such as the program in Figure 3. 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.

4.4 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 paper 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 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 inter-procedural MOD/REF analysis. For example, in Figure 6, 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 [5]: 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'.

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']
   ...
```

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 6 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.

```
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 }
```

Figure 6: An example of LIVE Variable propagation from the caller to the callee

DEFINITION 4.11 (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_$Down(P, \ell, v, Q) = L$.

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 7, 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:

```
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 }
```

Figure 7: An example of LIVE Variable propagation from the callee to the caller

DEFINITION 4.12 (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 $\mathbb{C}(M)$ be the slicing criterion for a given MPI program M; Let $\mathbb{S}(M)$ be the slice set computed by LPSA. Then the correctness of the algorithm can be stated by Theorem 4.13. A sketch of the proof of this theorem is given in [36].

THEOREM 4.13. $\mathcal{S}(\mathbb{C}(M)) = \mathbb{S}(M)$

4.5 Discussions

A common question to our algorithm is that how it works with applications whose communication behavior is dependent on message data, or even input data. As demonstrated by Theorem 4.13, LPSA algorithm can always guarantee that the generated program slice will preserve these message data or input data and related computation statements.

We perform experiments with 7 NPB programs and Sweep3D. There are a few marked MPI communication routines. For example, in NPB-BT program of Figure 8, MPI_Bcast is used to broadcast the iteration count and datatype to slave processes. In order to not affect the communication patterns, these routines will be executed at runtime.

An interesting observation is that all the marked MPI communication statements are collective communications and none of point-to-point communications is marked in these programs, i.e. the message contents of all the point-to-point communications are irrelevant to communication patterns. This can be explained by the fact that in mature MPI applications, collective communications are generally more preferred than point-to-point communications. Thus, although our approach for matching communication routines is quite conservative, we find it works well for all benchmark applications we have tested.

There is some work on more accurate communication matching algorithms. MPI-ICFG [35] is considered an effective approach to identify potential matching operations. Recently Bronevetsky [7]

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
12:     /* Top-Down Phase */
13:     for each procedure P in Pre-Order over PCG do
14:         call INTRA-LIVE-EXT(P)
15:         for each Q ∈ successor(P) do
16:             for each parameter v at callsite ℓ do
17:                 if ((ℓ, v) ∈ LIVE[P]) then
18:                     L = LIVE_Down(P, ℓ, v, Q)
19:                     for each ℓ' ∈ L do
20:                         if ℓ' ∈ MPI_Routines then
21:                             Mark MPI statement ℓ'
22:                         S(Q) = S(Q) ∪ {ℓ'}
23:                         for each x ∈ USE[ℓ'] do
24:                             WL[Q] ← {(ℓ', x)} ∪ WL[Q]
25:     /* Bottom-Up Phase */
26:     for each procedure Q in Post-Order over PCG do
27:         call INTRA-LIVE-EXT(Q)
28:         for each P ∈ predecessor(Q) do
29:             for each formal parameter v at ℓ0 in Q do
30:                 if ((ℓ0, v) ∈ LIVE[Q]) then
31:                     (ℓ', x) = LIVE_Up(Q, ℓ0, v, P)
32:                     WL[P] ← {(ℓ', x)} ∪ WL[P]
33:                     S(P) = S(P) ∪ {ℓ'}
34: For each procedure P: return S(P)
```

```
134:[M]  call MPI_Bcast(niter, 1, MPI_INTEGER,
        >             root, comm_setup, error)
136:[M]  call MPI_Bcast(dt, 1, dp_type,
        >             root, comm_setup, error)
          ...
191:      do  step = 1, niter
          ...
217:      endo
          ...
```

Figure 8: Marked MPI collective communications by LPSA, program snippet is from NPB-3.3/bt.f (M means marked).

has also proposed an uniform data flow framework to address this problem. We are studying more MPI applications and may include them when there is a demand. In addition, we are defining some annotation constructs that programmers can use to tag the matching communication operations.

5. IMPLEMENTATION

5.1 Compilation Framework

We have implemented `LPSA` for FACT in the production compiler, Open64 [31]. Open64 is the open source version of the SGI Pro64 compiler under the GNU General Public License (GPL). As shown in Figure 9, 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.

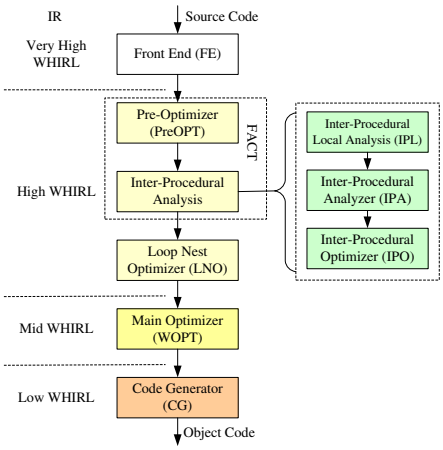


Figure 9: FACT in the Open64 infrastructure.

FACT is implemented in the PreOPT and inter-procedural analysis modules as shown in Figure 9. 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.

5.2 Runtime Environment

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 be executed, such as `MPI_Wait` or `MPI_Waitall`.

```
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
}
```

Figure 10: Pseudo code for collecting the communication traces for `MPI_Send` routine at runtime.

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 10 gives an example of collecting the communication traces for `MPI_Send` routine. In Figure 10, *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.

6. EVALUATION

6.1 Methodology

We evaluate the FACT with 7 NPB programs [4], BT, CG, EP, FT, LU, MG, SP and ASCI Sweep3D (S3) [23]. NPB is a set of scientific benchmarks derived from computational fluid dynamics applications. We use version 3.3 of NPB and the Class D data set. Sweep3D is an application in the ASCI Purple suite which is used to solve a three-dimensional particle transport problem. In our experiments, the problem size in the Sweep3D is fixed for each process ($150 \times 150 \times 150$).

We perform our experiments on two platforms: *test platform* and *validation platform*. The *test platform* is a small-scale system used to collect communication traces with FACT, while the *validation*

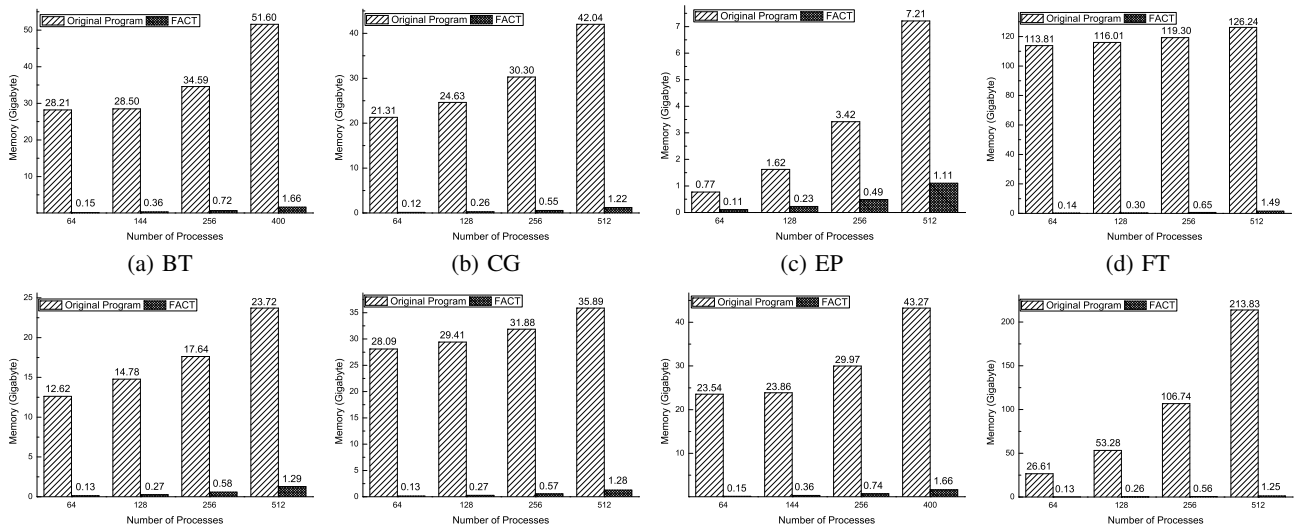


Figure 11: The memory consumption (in GigaByte) of FACT for collecting the communication traces of NPB programs (Class D) and Sweep3D ($150 \times 150 \times 150$) on the *test platform*. Traditional trace collection methods cannot achieve this on the *test platform* due to the memory limitation. The memory consumption of the original programs are collected on the *validation platform*.

platform is a large-scale system used to validate the communication traces collected with FACT and record memory requirements and execution time of the original programs. Details of the two platforms are given below:

- **Test Platform (32 cores):** Small cluster of four nodes, where each node is a 2-way Quad-Core with Intel Xeon E5345 2.33 GHz CPUs and 8 GB memory, and connected with a Gigabit Ethernet. Our custom communication library is implemented based on mpich2-1.0.7 [3].
- **Validation Platform (512 cores):** A cluster consisting of 32 nodes. Each node is a 4-way Quad-Core with AMD8347 1.9 GHz CPUs and 32 GB memory, and connected with a 20 Gbps Infiniband network. MPI library is mvapich-1.1.0 [27].

6.2 Validation

The proof of our algorithm can be found in [36]. In addition, we also validate the implementation of FACT by comparing the communication traces collected by FACT with traces collected by traditional trace collection methods on the *validation platform*. We perform comparison for the 7 NPB programs and the Sweep3D for different numbers of processes (64, 128, 256, and 512). They are identical except that the traces collected by FACT do not include time stamp.

6.3 Performance

6.3.1 Memory Consumption

To present the advantages of FACT over the traditional trace collection methods, we collect communication traces of NPB programs (Class D) and Sweep3D ($150 \times 150 \times 150$) 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.

Our experimental results shown in Figure 11 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.

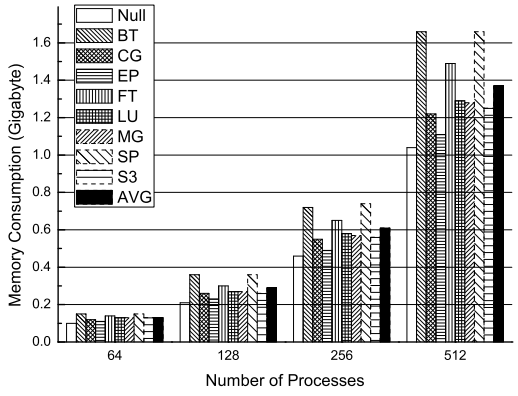
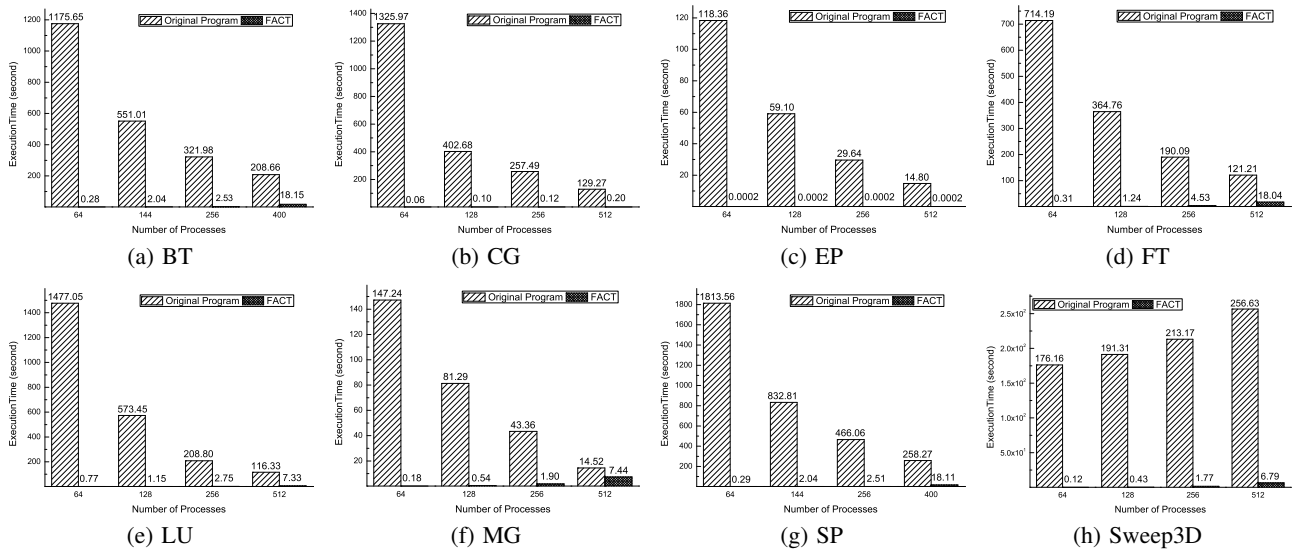


Figure 12: The memory consumption of FACT compared to *Null* micro-benchmark when collecting the communication traces for different programs. AVG is the arithmetic mean for all the programs.

Figure 12 shows the memory consumption of the FACT compared to the *Null* benchmark. *Null* is a micro-benchmark which only contains the invocations of `MPI_Init` and `MPI_Finalize`, and no other computation or communication operations. *Null* is



(a) BT (b) CG (c) EP (d) FT

(e) LU (f) MG (g) SP (h) Sweep3D

Figure 13: 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)*.

used to provide a lower bound on memory consumption for an MPI program with different numbers of processes. Note that the memory consumption of *Null* grows when the number of processes increases. It is because that the MPI communication library itself consumes a certain memory for process management. As shown in Figure 12, the memory consumption of FACT for all the programs is close to the *Null* benchmark with different numbers of processes. Among these programs, BT and SP consume relatively more memory resources than others. In contrast, EP and CG consume the least memory. For example, with 512 processes, the *Null* benchmark consumes 1.04 GB memory, while EP and CG consume 1.11 GB and 1.22 GB memory respectively. Additionally, the memory buffer in our runtime library of FACT will also consume a certain memory, no more than 320 KB for each process. The results prove that `LPSA` algorithm in FACT can effectively reduce memory requirements of the original programs.

6.3.2 Execution Time

Figure 13 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.

When collecting communication traces for 512 processes, the execution time with FACT has a slight increase for a few programs. This is caused by the following two reasons: (1) Besides the computation time in the program slice, the execution time with FACT also includes the time of recording the communication traces into the memory buffer and eventually writing them into the disk files. When the number of processes is small enough, the buffer of the file system can hold all the trace files in the memory. When the number of processes increases, the size of communication trace files will exceed the buffer limitation of the file system. The file system will flush the buffer to the hard disk. As a result, the I/O time increases dramatically for a large number of processes. (2) As the marked communication operations at compile time would be executed at runtime, such as the `MPI_Bcast` invocation of the BT program in Figure 8, the communication time will also increase on such a small-scale system for a large number of processes due to network contention.

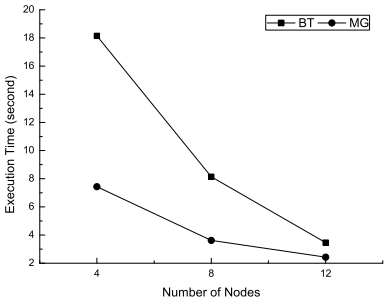


Figure 14: The execution time when collecting the communication traces of MG with 512 processes and BT with 400 processes by using more nodes on the *test platform*.

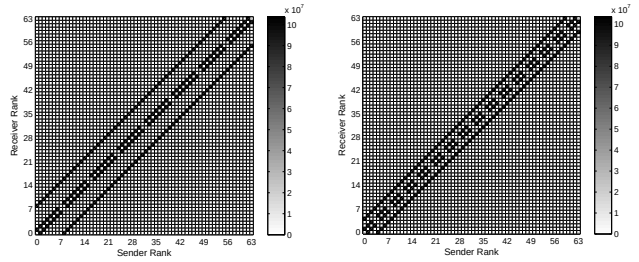
Overall, the execution time with FACT is acceptable for most developers to study the communication patterns on such a small-scale system. In addition, FACT can benefit when more nodes are available. Figure 14 illustrates the results when collecting the communication traces of MG with 512 processes and BT with 400 processes by using more nodes on the *test platform*. FACT represents good scalability with the number of nodes. For example, with 12 nodes FACT only takes 2.43 seconds and 3.46 seconds to collect the communication traces for MG and BT respectively. This is because both the I/O time and the communication time mentioned before

are reduced with the increase of number of nodes.

7. APPLICATION

There are arguments on the usage of communication traces, just like any other profile-based analysis, that communication traces are in fact dependent on input parameters. Thus, it is important to reveal the relationship between input parameters and communication patterns. Traditionally, developers manually express the application specific knowledge of the parallel applications, but it is very time consuming and error-prone. It is greatly desired that we can perform sensitivity analysis of communication patterns to key input parameters automatically.

Because of the prohibitive resource and time requirements of traditional methods, it is too expensive to get communication traces even for a single input, not to mention the sensitivity analysis which requires multiple executions. FACT enables us to explore the sensitivity analysis in a cost-effective way. We use Sweep3D as an example.



(a) (mk=10 mmi=3 i=8 j=8) (b) (mk=10 mmi=3 i=4 j=16)

Figure 15: The sensitivity of communication spatial locality of Sweep3D to input parameters: i and j. The gray level of a cell at the x^{th} row and y^{th} column represents the communication volume (in Byte) between two processes x and y.

Sweep3D has four key input parameters affecting the communication performance of the program: i, j, mk and mmi. The values of i and j determine the mapping shape of the processes. The number of processes must be equal to the product of i and j. The computation granularity of the pipeline is determined by a k-plane blocking factor (mk) and an angle blocking factor (mmi). In our experiments, it takes *less than 1 second* for collecting 7 sets of communication traces by FACT with different input parameters on the *test platform*. The process number is 64. Figure 15 exhibits the communication locality [21] of Sweep3D when varying the input parameters, i and j. Note that the two sets of input parameters present different communication localities for the application. For example, in Figure 15(a), *Process 8* communicates with *Processes 0, 9, 18* frequently, while in Figure 15(b), *Process 8* communicates with *Processes 4, 9, 12* more frequently.

Table 1: The sensitivity of message size (msg_size) in byte and message count (msg_count) to the input parameters: mk and mmi.

Input Paras	mk=10 mmi=1	mk=20 mmi=1	mk=10 mmi=3	mk=20 mmi=3	mk=50 mmi=6
msg_size	12000	24000	36000	72000	360000
msg_count	17280	9216	5760	3072	576

Table 1 shows that when increasing the value of mk, the message size will increase. It is the similar for the parameter of mmi. When "mk=50" and "mmi=6", the message size changes to 360000 Byte and the message count is 576 in the Sweep3D. This information is very important for understanding the communication behavior of the Sweep3D.

8. LIMITATIONS AND DISCUSSIONS

Absence of temporal attributes The FACT framework is based on the observation that there are many applications of communication trace analysis that can be done without temporal attributes. We reiterate some known applications here: MPI process mapping optimization [8, 43], communication subsysetm design [12], and subgroup replay-based MPI debuggers [41]. In addition, FACT can also be used to do performance prediction as described in [34]. We do not intend and it is impossible to list all potential applications of FACT, but we believe these known applications are sufficient to support its usefulness.

FACT can not be used to support performance optimization and performance debugging that require temporal information. For example, traces collected by FACT can not be used with performance tuning tools such as Intel Trace Analyzer [19] to get the overhead of a message transfer. Some automatic performance tuning algorithms, such as the critical path analysis [30], can not be supported by FACT either.

However, we would like to mention that although traces collected by FACT do not include temporal attributes, they do include more than spatial and volume attributes. One important attribute we preserve in FACT framework is the $order$ attribute. Traces collected by FACT have all the message operations and their sequences in addition to statistics on volume and spatial attributes. The $order$ attribute has many potential applications. For example, they can be used in identifying similarity between MPI processes. Due to the limitation of space, we omit more discussion on this issue.

Communication non-determinism Like any trace-based approaches, FACT also has its limitation in representing many executions with one trace. One possible way to address the problem is to obtain traces for more than one execution and observe the sensitivity of communication patterns to different executions. In this sense, FACT has significant advantage over traditional trace collection approaches because it costs much less in acquiring traces.

FACT may also affect communication patterns of non-deterministic applications in a more subtle way. Because FACT may remove computation from original programs, it changes the load balancing characteristics that may result in the change of communication patterns. We will perform more investigation on this direction and see if this is a real problem in practice.

9. 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 1-2 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. With FACT, we are enabled to explore more applications of using communication patterns. In the future, we will evaluate FACT with more parallel applications.

10. ACKNOWLEDGMENTS

We would like to thank our shepherd, Allen Malony, and the anonymous reviewers for their insightful comments. We thank Kang Chen, Wei Xue, Ruini Xue, Chuntao Hong, Shiming Xu, Jianian Yan, Jin Zhang, Jiaqi Zhang, Dehao Chen, Hongshan Jiang, Dandan Song for their valuable feedback and suggestions. This research is supported by National High-Tech Research and Development Plan (863 plan) 2006AA01A105 and Chinese National 973 Basic Research Program 2007CB310900.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] A. W. Appel. *Modern Compiler Implementation in C: Basic Techniques*. Cambridge University Press, New York, USA, 1997.

[3] Argonne National Laboratory. MPICH2. `http://www.mcs.anl.gov/research/projects/mpich2`.

[4] 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.

[5] J. Banning. An efficient way to find side effects of procedure calls and aliases of variables. In *POPL*, pages 29–41, 1979.

[6] D. Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11-12):583–594, 1998.

[7] G. Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *CGO*, 2009.

[8] 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.

[9] Y. Chen, S. Byna, X. Sun, R. Thakur, and W. Gropp. Hiding I/O latency with pre-execution prefetching for parallel applications. In *SC'08*, pages 1–10, 2008.

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

[11] Z. Ding, R. Hoare, A. Jones, D. Li, S. Shao, S. Tung, J. Zheng, and R. Melhem. Switch design to enable predictive multiplexed switching. In *IPDPS*, page 100.1, 2005.

[12] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers, 2003.

[13] A. Faraj and X. Yuan. Communication characteristics in the NAS parallel benchmarks. In *International Conference on Parallel and Distributed Computing Systems*, 2002.

[14] 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.

[15] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[16] M. Harman and S. Danicic. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability*, 5:143–162, 1995.

[17] S. Ho and N. Lin. Static analysis of communication structures in parallel programs. In *International Computer Symposium*, 2002.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[19] Intel Ltd. Intel trace analyzer & collector. `http://www.intel.com/cd/software/products/asmo-na/eng/244171.htm`.

[20] 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 *Supercomputing*, pages 37–48, 2001.

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

[22] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. DiP: A parallel program development environment. In *Euro-Par'96*, pages 665–674, 1996.

[23] LLNL. ASCI purple benchmark. `https://asc.llnl.gov/computing_resources/purple/archive/benchmarks`.

[24] B. Mohr and F. Wolf. KOJAK–A tool set for automatic performance analysis of parallel programs. In *Euro-Par*, 2003.

[25] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[26] 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.

[27] Ohio State University. MVAPICH: MPI over infiniband and iWARP. `http://mvapich.cse.ohio-state.edu`.

[28] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan. Detecting patterns in MPI communication traces. In *ICPP*, pages 230–237, 2008.

[29] R. Preissl, M. Schulz, D. Kranzlmüller, B. R. de Supinski, and D. J. Quinlan. Using MPI communication patterns to guide source code transformations. In *ICCS*, pages 253–260, 2008.

[30] M. Schulz. Extracting critical path graphs from MPI applications. In *CLUSTER*, pages 1–10, 2005.

[31] SGI. Open64 compiler and tools. `http://www.open64.net`.

[32] S. Shao, A. K. Jones, and R. G. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *IPDPS*, 2006.

[33] S. Shende and A. D. Malony. TAU: The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2), 2006.

[34] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for application performance modeling and prediction. In *SC*, pages 1–17, 2002.

[35] M. M. Strout, B. Kreaseck, and P. Hovland. Data-flow analysis for MPI programs. In *ICPP*, pages 175–184, 2006.

[36] Tsinghua University. Proof of live-propagation slicing algorithm. `http://www.hpctest.org.cn/paper/Thu-HPC-TR20090717.pdf`, 2009.

[37] J. S. Vetter and M. O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *PPoPP*, pages 123–132, 2001.

[38] J. S. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. In *IPDPS*, pages 853–865, 2002.

[39] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–52, 1982.

[40] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[41] 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.

[42] R. Zamani and A. Afsahi. Communication characteristics of message-passing scientific and engineering applications. In *International Conference on Parallel and Distributed Computing Systems*, 2005.

[43] J. Zhang, J. Zhai, W. Chen, and W. Zheng. Process mapping for mpi collective communications. In *Euro-Par*, 2009.