A Tool for Structural Testing of MPI Programs

Alexandre C. Hausen¹  Silvia R. Vergilio¹  Simone R. S. Souza²
Paulo S. L. Souza²  Adenilso S. Simão²

Federal University of Paraná¹
UFPR-DInf, CP 19081
CEP 81531-970, Curitiba–Brazil
{ceolin,silvia}@inf.ufpr.br

State University of S são Paulo²
USP-ICMC, CP 668
CEP 13560-970, São Carlos–Brazil
{srocio,pssouza,adenilso}@icmc.usp.br

Abstract

Among the message passing environments, MPI (Message Passing Interface) has been considered by several authors as the de facto standard to build parallel software. In spite of this great popularization, there is a lack of tools that support the test of MPI programs. The existent tools do not support the application of a test criterion; they only aim at the visualization and debugging. The use of a testing criterion is fundamental to ensure the quality of the generated test sets and to offer coverage measure to evaluate the testing activity. This work describes a tool, named V aliMPI, that implements testing criteria specific to message passing environments, and considers the control, data and communication flows of parallel programs. The tool supports generation and evaluation of test sets, as well as the test replay.

1. Introduction

The use of parallelism is considered crucial to reduce computing time in several domains such as: weather forecast, image processing, bioinformatics and others. In this work, we consider parallel programs built with message passing environments. These environments use libraries, implemented for sequential languages (for instance, C and Fortran), that allow the communication among parallel processes. Some examples are: Express, Parmacs, PVM and MPI (Message Passing Interface [9]). The MPI is considered by several authors as the de facto message passing standard and is one of the most used and known environments [1].

Parallel program testing is an important and hard task, mainly due to features of these programs [5]: non-determinism, concurrence, synchronization and communication. In this way, the use of a test criterion is fundamental. A test criterion [4] is a predicate to be satisfied by a set of test cases and can be used as guideline for generating test data, offering a coverage measure that indicates whether enough test cases have been generated. Structural criteria use the program source code and structural features of the program to derive test cases.

There are in the literature some efforts to extend testing criteria to test parallel programs [14, 18, 19]. However, only [16] addresses message passing environments, in spite of the increasing use and popularization of this kind of parallel software. In addition, most mentioned works do not address supporting tools and we observe that the application of a testing criterion is only possible if a tool is available. When we consider MPI, we find some tools [15, 17] that only aid the simulation and debugging, but do not support the application of a testing criterion and the evaluation of a test set.

This paper presents the V aliMPI tool, that aims to fulfill the above mentioned demand. V aliMPI supports the application of the testing criteria introduced in [16] and is the version of the V aliPar tool [12] to support the test of MPI programs. V aliMPI produces the coverage obtained by a test set provided by the tester with respect to the implemented criteria and allows replay of test cases, by implementing controlled execution, easing debugging and regression test. Some experimental results show the applicability and advantages of V aliMPI.

This paper is organized as follows. In Section 2, we review the main concepts upon which the tool was built. In Section 3, the V aliMPI architecture is described. In Section 4, results of experiments are presented. Section 5 contains related work, and Section 6 makes some concluding remarks.
2. Previous Work

In our previous work [16], we proposed a set of criteria specific to message passing environments and an architecture, named ValiPar [12], to implement them. These criteria were based on testing criteria for sequential programs, but they also consider typical faults in parallel programs: communication, synchronization and nondeterminism related faults. The criteria consider a model based on the control, data and communication flows of the program. This model considers that the number of parallel processes is statically known.

A parallel program \( P \) is represented by a PCFG (Parallel Control Flow Graph), which is composed by the CFGs (Control Flow Graphs) of each process. The CFG is composed by a set of nodes and edges. Each node corresponds to a statement of the program and an edge links a node to another. A node \( i \) in the process \( p \) is represented by \( n^p_i \). A node might be associated to a communication primitive: a send or receive. A synchronization edge \((n^a_i, m^b_j)\) links a send node of process \( a \) to a receive node of process \( b \). These edges represent the possibility of communication and synchronization between processes. Fig. 1 illustrates a PCFG for two processes, in which the synchronization edge \((6^0, 8^1)\) or \((6^1, 8^0)\) can occur, depending on the control flow.

![Figure 1. An Example of PCFG](image)

The defined criteria require that elements of the PCFG be exercised [16]: (AN) All-Nodes; (AE) All-Edges; (AR) All-Nodes-R and (AS) All-Nodes-S related on nodes with receives and sends respectively; and (AES) All-Edges-S related on possible synchronizations edges. Other proposed data-flow based criteria are: (ACU) All-C-Uses; (APU) All-P-Uses; (ASU) All-S-Uses; (ASCU) All-S-C-Uses and (ASPU) All-S-P-Uses. They consider definitions and uses of variables. A variable \( x \) is defined when a value is saved in the correspondent memory position (for instance, assignments and input commands) and when it is passed as an output parameter (reference) to a function. In the message passing context, we must also consider the communication functions, such as receive, because it sets a variable with the value received in the message. A use of \( x \) occurs when the value associated to \( x \) is referred to. A use can be: 1) a computational use (c-use), which occurs in a computational statement related to a node in the CFG; 2) a predicative use (p-use), which occurs in a condition (predicate) associated to a control flow statement, related to an edge in the CFG; and 3) a communication use (s-use), which occurs in a synchronization statement, related to a synchronization edge in PCFG.

3. ValiMPI Architecture

The ValiMPI tool supports the following testing criteria: AN, AR, AS, AE, AES, ACU, APU and ASU. It implements the architecture of ValiPar [12] for MPI parallel programs, written in C language. The modules of ValiMPI (Vali-Inst, Vali-Elem, Vali-Exec and Vali-Eval) are described in the following subsections.

3.1. Vali-Inst

The Vali-Inst module extracts control and data flow information and does the program instrumentation. These tasks are supported by IDeL (Instrumentation Description Language) [11]. IDeL is a meta-language that accomplishes syntactic and semantic analysis of a language that can be instanced for different programming languages. We extended the IDeL version for C language to deal with specific aspects of MPI. The PCFG is generated in text files, one for each function, with information about definitions and uses of variables in the nodes, as well as the presence of send and receive commands. Listing 1 shows this textual representation for the PCFG given in the example of Fig. 1. The instrumented program is obtained by inserting check-points statements in the program being tested. These statements do not change the program semantics. They only write necessary information in a trace file, by registering the node and the process identifier in the send and receive commands. The instrumented program will produce the paths executed in each process, as well as the synchronization sequence produced within a test case. For the program presented in Listing 2, Vali-Inst generates the instrumented program of Listing 3. The instrumented program also contains, in each of its functions, some macros (start with the word "VALIMPI") for declaring local data structures, initializing these data structures and creating the respective files.
Figure 2. ValiMPI architecture

Listing 1. Textual representation for the PCFG

```plaintext
digraph gfc {
    node[shape=circle] 1;
    /* definition of argc at 1 */
    node[shape=doublecircle] 11;
    /* deref definition of argc at 2 */
    node[shape=circle] 3;
    /* deref definition of myrank at 3 */
    node[shape=circle] 4;
    /* deref definition of msg at 5 */
    node[shape=circle] 5;
    /* usage of msg at 6 */
    node[shape=circle] 10;
    /* usage of msg at 8 */
    node[shape=circle] 20;
    /* usage of msg at 10 */
    node[shape=circle] 30;
    /* usage of msg at 9 */
    node[shape=circle] 40;
    /* usage of msg at 7 */
    node[shape=circle] 50;
    /* usage of msg at 11 */
    node[shape=circle] 60;
    /* usage of msg at 11 */
    node[shape=circle] 70;
    /* usage of msg at 10 */
    node[shape=circle] 80;
    /* usage of msg at 12 */
}
```

Listing 2. Original source

```c
#include <mpi.h>
#include "valmpi.h"

int main(int argc, char *argv[]) {
    char msg[10];
    int myrank;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0) {
        strcpy(msg, "Hello!");
        MPI_Send(msg, 7, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    } else {
        MPI_Recv(msg, 10, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received: %s \n", msg);
    }
    MPI_Finalize();
    return 0;
}
```

Listing 3. Instrumented source

```c
#include "valmpi.h"
#include "mpi_hello.h"

int main(int argc, char *argv[]) {
    char msg[10];
    int myrank;
    VALIMPI_TRACE_DECL;
    VALIMPI_REQ_LIST_DECL;
    ValiMPI_Init_trace(&valimpiTrace, "main");
    ValiMPI_Send_trace(&valimpiTrace, 1); {
        ValiMPI_Check_trace(&valimpiTrace, 2);
        MPI_Init(&argc, &argv);
    }
    ValiMPI_Check_trace(&valimpiTrace, 3);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); {
        ValiMPI_Check_trace(&valimpiTrace, 4);
        if (myrank == 0) {
            ValiMPI_Check_trace(&valimpiTrace, 5); {
                ValiMPI_Check_trace(&valimpiTrace, 5);
                strcpy(msg, "Hello!");
            }
            ValiMPI_Send_trace(&valimpiTrace, msg, 7, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
        }
        ValiMPI_Check_trace(&valimpiTrace, 7);
    }
    else {
        ValiMPI_Check_trace(&valimpiTrace, 8);
        int status;
        if (status == 9) {
            ValiMPI_Check_trace(&valimpiTrace, 9);
            printf("received: %s \n", msg);
        }
    }
}
```
3.2. Vali-Elem

The Vali-Elem module generates the elements required by the coverage testing criteria. These elements are generated from the PCFG using the information produced by Vali-Inst. Besides the PCFG, two other graphs are also used to obtain the required elements: the heirs reduced graph and the graph(i) [2, 8]. The heirs reduced graph is generated based on the fact that there are some edges in CFG that are always executed when another one is executed. So, it is possible to minimize the number of required edges. The graph(i) is built for each node that contains a variable definition and it is used to establish associations of definitions and uses of variables, which are the required elements of the data-flow based testing criteria.

A descriptor for each element required by a criterion is also produced. This descriptor is a regular expression that describes a path that covers (matches) the required element.

Table 1 presents some elements required by the criteria implemented by Vali-Elem for program of Listing 2, considering two processes.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Required elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>AN</td>
<td>1 &lt; 5, 5 &gt; 10</td>
</tr>
<tr>
<td>AR</td>
<td>5 &lt; 5</td>
</tr>
<tr>
<td>AS</td>
<td>5 &lt; 6</td>
</tr>
<tr>
<td>AE</td>
<td>5 &lt; 5, 5 &gt; 1</td>
</tr>
<tr>
<td>AES</td>
<td>6 &lt; 6, 8</td>
</tr>
<tr>
<td>ACU</td>
<td>6 &lt; 6, 8, msg &gt;</td>
</tr>
<tr>
<td>APU</td>
<td>6 &lt; 6, 8, myrank &gt;</td>
</tr>
<tr>
<td>ASU</td>
<td>6 &lt; 6, 8, msg &gt;</td>
</tr>
</tbody>
</table>

3.3. Vali-Exec

The Vali-Exec module executes the instrumented program using the test data provided by the tester. During the execution, standard input flow, command line, program outputs, the execution traces and synchronization sequences are stored in separated files. The execution trace includes the paths executed in each process by the test input and it is used during the evaluation of test cases to determine which required elements were covered. After the program execution, the tester can visualize the outputs and also the execution trace to determine whether the obtained output is the same as expected. If it is not, an error was identified and must be corrected before continuing the test activity.

3.3.1. Controlled Execution. Controlled execution is a mechanism used to achieve deterministic execution, i.e., two executions of the program with the same input are guaranteed to execute the same instruction sequence [13]. It may also be used to try to force the execution of a desired synchronization sequence, aiming at increasing the coverage. Based on work of Tai et al. [13], the controlled execution was implemented in ValiMPI.

Synchronization sequences of each processes are gathered in runtime by the instrumented check-points of blocking and nonblocking send, receive and message request tests. The latter is also subject to nondeterminism, so each request is associated to the number of times it has been evaluated. This information and the other program inputs are used to achieve the deterministic execution, thus to allow test case replay.

3.4. Vali-Eval

The Vali-Eval module evaluates the coverage achieved by test sets with respect to a criterion, which can be selected by the tester. Vali-Eval uses the descriptors and elements generated by Vali-Elem and the paths executed by the test cases. The module implements the automata associated to the descriptors. A required element is covered if an executed path is recognized by its correspondent automaton [4]. The coverage score and the list of covered and not covered elements for the selected test criterion is provided as output.

4. Experimental Studies

To evaluate the implementation of ValiMPI, we conducted a study with four parallel programs: mdc, which calculates the greatest common divisor of three numbers [5]; phil, which implements the dining

![Figure 3. Nondeterministic PCFG](image-url)
philosophers problem and matrix, which implements multiplication of matrix. 

The results obtained for each program are shown in Tables 2-4, respectively. The study consisted of the following steps:
1. For each program, an initial test set $T_i$ was randomly generated. The cardinality is given by $|T_i|$. 
2. $T_i$ was submitted to ValiMPI and an initial coverage ($C_i$) was obtained for all the criteria implemented by the tool. The number of required elements for each criterion is $RE$. $T_e$ contains test cases that really contributed to cover elements in the executed order. 
3. Additional test cases ($T_a$) were generated to cover the elements required which was not covered by $T_i$. $T_f$ is given by $T_i \cup T_a$. $C_f$ is the coverage obtained by $T_f$.

In this step, the infeasible elements were manually detected with support of the controlled execution. The number of infeasible elements required for each criterion is given by $IE$.

### Table 2. Results for mdc, $|T_i| = 99$

| Crit. | $RE$ | $C_i$ | $|T_e|$ | $C_f$ | $|T_f|$ | $IE$ |
|-------|------|-------|--------|-------|--------|------|
| AN    | 62   | 100%  | 6      | 0     | 0      | 0    |
| AR    | 7    | 100%  | 2      | 0     | 0      | 0    |
| AS    | 7    | 100%  | 2      | 0     | 0      | 0    |
| AE    | 41   | 51.2% | 3      | 20    | 20     | 20   |
| AES   | 30   | 33.3% | 3      | 20    | 20     | 20   |
| ACU   | 29   | 100%  | 3      | 0     | 0      | 0    |
| APU   | 40   | 95%   | 7      | 100%  | 9      | 0    |
| ASU   | 66   | 24.2% | 7      | 28.8% | 10     | 47   |

### Table 3. Results for phil, $|T_i| = 15$

| Crit. | $RE$ | $C_i$ | $|T_e|$ | $C_f$ | $|T_f|$ | $IE$ |
|-------|------|-------|--------|-------|--------|------|
| AN    | 176  | 100%  | 2      | 0     | 0      | 0    |
| AR    | 11   | 100%  | 1      | 0     | 0      | 0    |
| AS    | 36   | 100%  | 2      | 0     | 0      | 0    |
| AE    | 356  | 21.3% | 2      | 280   | 280    | 280  |
| AES   | 325  | 13.8% | 2      | 280   | 280    | 280  |
| ACU   | 50   | 100%  | 2      | 0     | 0      | 0    |
| APU   | 148  | 78.4% | 3      | 81.8% | 4      | 27   |
| ASU   | 335  | 16.4% | 2      | 280   | 280    | 280  |

By analysing the results, we observe that the criteria are applicable and the tool might be used in practice. In spite of the great number of elements required for program matrix, the number of test cases do not grow proportionally. A low number of test cases was necessary to satisfy the criteria. With respect to the infeasibility aspect, a great number of infeasible elements were required for the criteria AES and ASU (based on communication flow). All infeasible edges of the programs mdc and phil are inter-process edges. In program matrix most of the infeasible elements are due to the master/slave processes that were coded in a single function. When proposing the criteria, we have adopted a conservative position by requiring all the possible interprocess edges, even if in practice the communication is not possible. This was adopted with the goal of revealing faults related in absent communications. However, we intend to conduct other experiments to explore efficacy aspects to propose changes in the way of generating the required elements and to avoid a large number of infeasible ones. In all cases, the evaluation of ASU was always the most expensive (in time). This is due to the combinatorial nature of the synchronization possibilities.

### Table 4. Results for matrix, $|T_i| = 1$

| Crit. | $RE$ | $C_i$ | $|T_e|$ | $C_f$ | $|T_f|$ | $IE$ |
|-------|------|-------|--------|-------|--------|------|
| AN    | 368  | 45.1% | 1      | 45.6% | 2      | 200  |
| AR    | 36   | 58.3% | 1      | 1      | 15     |      |
| AS    | 36   | 41.7% | 1      | 1      | 21     |      |
| AE    | 1032 | 4.7%  | 1      | 4.8%  | 2      | 982  |
| AES   | 972  | 2.78% | 1      | 1      | 945    |      |
| ACU   | 572  | 39.3% | 1      | 41.1% | 2      | 337  |
| APU   | 304  | 30.6% | 1      | 34.2% | 2      | 206  |
| ASU   | 1404 | 1.9%  | 1      | 2.1%  | 3      | 1375 |

By analysing the results, we observe that the criteria are applicable and the tool might be used in practice. In spite of the great number of elements required for program matrix, the number of test cases do not grow proportionally. A low number of test cases was necessary to satisfy the criteria. With respect to the infeasibility aspect, a great number of infeasible elements were required for the criteria AES and ASU (based on communication flow). All infeasible edges of the programs mdc and phil are inter-process edges. In program matrix most of the infeasible elements are due to the master/slave processes that were coded in a single function. When proposing the criteria, we have adopted a conservative position by requiring all the possible interprocess edges, even if in practice the communication is not possible. This was adopted with the goal of revealing faults related in absent communications. However, we intend to conduct other experiments to explore efficacy aspects to propose changes in the way of generating the required elements and to avoid a large number of infeasible ones. In all cases, the evaluation of ASU was always the most expensive (in time). This is due to the combinatorial nature of the synchronization possibilities.

### 5. Related Work

Most tools available for parallel software only aid the simulation, visualization and debugging; they do not support the application of testing criteria. Examples are TDC Ada [13] and ConAn [6], respectively, for ADA and Java. For message passing environments we can mention MDB [3] for PVM, XMPI [15] and Umpire [17] for MPI.

When considering criteria support, we can mention the Della Pasta [18] for Ada; Astral [10] and STEPS [7] for PVM programs. We did not find in the literature a tool similar to ValiMPI that implements criteria based on control, data and communication flows. Table 5 shows the main facilities of ValiMPI, compared to others tools.

### 6. Conclusion

This paper presented ValiMPI, a tool for testing MPI parallel programs written in C. As far as we know, this is the first test tool that supports test criteria specific to message passing environments in the context of MPI programs. ValiMPI can be used to support the test
data selection and to offer coverage measures that can be used to evaluate the quality of the test sets. ValiMPI provides the possibility of test case replay, a facility required for verifying the effectiveness of a fault correction. It is also possible to construct a synchronization sequence in order to cover a required element which may be difficult to cover due to the nondeterminism.

Results of an experiment showed the applicability of the criteria. The results showed a great number of required elements mainly for the communication-flow based criteria. This should be evaluated in future experiments and some refinements could be proposed to the criteria.

Future work includes the development of an intuitive graphical interface and the adaptation of ValiMPI to Fortran and C++. We also intend to conduct other experiments to better evaluate the cost/efficacy of the testing criteria.

References


Table 5. Existent Testing Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Data flow</th>
<th>Ctrl. flow</th>
<th>Test replay</th>
<th>Debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDC Ada$^a$</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConAn$^b$</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Della Pasta$^c$</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MDB$^d$</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STEPS$^e$</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Astral$^f$</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XMPI$^g$</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Umpire$^h$</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ValiMPI$^i$</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>

$^a$Ada, $^b$Java, $^c$shmem, $^d$PVM, $^e$MPI.