Verification of MPI Programs using CIVL

Ziqing Luo
Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark, Delaware, USA 19711
ziqing@udel.edu

Manchun Zheng
Pure Storage Company
California, USA
mzheng@purestorage.com

Stephen F. Siegel
Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark, Delaware, USA 19711
siegel@udel.edu

ABSTRACT

CIVL is a framework for verifying concurrent programs. The framework is built around a language, CIVL-C, that extends sequential C with general-purpose primitives that can be used to model a variety of concurrency dialects, including OpenMP, Pthreads, CUDA, and MPI. The framework automatically transforms programs using those dialects into CIVL-C so that static analysis and verification tools for CIVL-C can be applied. This paper describes how C/MPI programs are so transformed. The result is a verifier that can check, within finite bounds, a number of difficult properties of MPI programs, including functional correctness, deadlock-freedom, and adherence to rules specified in the MPI Standard.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Computing methodologies** → *Parallel programming languages*;

KEYWORDS

verification, MPI, model checking, symbolic execution, CIVL

ACM Reference format:
Ziqing Luo, Manchun Zheng, and Stephen F. Siegel. 2017. Verification of MPI Programs using CIVL. In *Proceedings of EuroMPI/USA '17, Chicago, IL, USA, September 25–28, 2017,* 11 pages.
https://doi.org/10.1145/3127024.3127032

1 INTRODUCTION

The problem of verifying correctness of concurrent programs has bedeviled researchers since the advent of concurrency. Early efforts, e.g., [16], focused on deductive approaches (proofs) carried out manually. In recent years more automated approaches have been introduced, based not only on deduction but also static analysis, model checking, and symbolic execution. Today a number of tools exist for verifying various aspects of different categories of concurrent programs. While most are research prototypes, many have nonetheless shown promise by detecting numerous defects in and verifying—with certain caveats—a wide variety of concurrent programs (cf. [15]).

One of the main barriers to further progress is the sheer number of different ways of expressing parallel programs. MPI (the "Message Passing Interface" [14]), OpenMP, CUDA, and Pthreads are just a few of the "concurrency dialects" used to write modern parallel programs. New dialects are introduced regularly, and the old ones constantly evolve. Furthermore, modern parallel architectures have necessitated the use of multiple dialects within a single program; these *hybrid* programs are especially difficult to reason about. Since it takes enormous effort to develop a practical verification tool, almost all of these efforts have targeted a single dialect; very few have targeted hybrid programs. Moreover, these tools are rarely updated as the API Standards evolve. At the same time, many of the underlying verification techniques used in these tools are very similar, differing mainly in the dialect they support. This results in redundant work, inadequate tool comparisons, and slow progress.

The CIVL (Concurrency Intermediate Verification Language) framework was developed to address these challenges [21]. The framework is centered around a common concurrent programming language named CIVL-C. The front-ends translate C programs that use MPI, OpenMP, CUDA, and Pthreads (alone or in combination) to CIVL-C. The back-end uses model checking and symbolic execution to verify CIVL-C programs. Ideally, a new dialect can be supported by just adding a new front-end; new verification techniques can be explored across a variety of dialects by adding a new

EuroMPI/USA '17, September 25–28, 2017, Chicago, IL, USA

ACM ISBN 978-1-4503-4849-2/17/09…$15.00
https://doi.org/10.1145/3127024.3127032

Funding for the CIVL project is provided by the U.S. National Science Foundation under awards CCF-1319571 and CCF-1346769. This material is also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number DE-SC0012566. This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

back-end. In reality, some additional work is usually needed in each case to achieve reasonable performance. However, the language and framework have been designed to facilitate such extensions while keeping the work to a minimum.

A high-level description of the CIVL framework appeared in [21], but that paper did not go into detail on the translation from any specific concurrency dialect to CIVL-C. The purpose of this paper is to describe the translation process for one dialect — MPI. (Papers on the other dialects are in preparation.)

In Section 2, we summarize the aspects of the CIVL-C language—including its "core library"—relevant to the translation of C/MPI programs. The structure of the translated code is explained in Section 3. A detailed description of the application of CIVL to two MPI examples is given in Section 4. Related work is discussed in Section 5. In Section 6, we evaluate the resulting tool in several ways: first, by enumerating desirable features of an MPI verification tool (Figure 8) and exploring how CIVL stacks up against other leading tools in its ability to provide these features; we also explore applications of CIVL to several realistic MPI applications; and finally we evaluate the usability and scalability of CIVL.

2 SUMMARY OF CIVL-C LANGUAGE

CIVL-C is an extension of the sequential part of C11 [8]. The names of the new keywords, types and functions all begin with `$`. The `$spawn` keyword may precede a function call, and indicates that a new *process* (or thread of control) should be created to execute the function call. The `$spawn` command returns immediately with a reference to the new process, an object of type `$proc`. The `$wait` function consumes an argument of type `$proc` and blocks until that process terminates.

CIVL-C allows function definitions to occur in inner (block) scopes, not just the file scope. Those functions can be spawned like any other function. This enables patterns such as the following:

```
1  void proc_f(int rank) {
2    void thread_f(int tid) {...}
3    $proc threads[nthreads];
4    $for (int i=0; i<nthreads; i++)
5      threads[i] = $spawn thread_f(i);
6    $for (int i=0; i<nthreads; i++)
7      $wait(threads[i]);
8  }
9  $proc procs[nprocs];
10 $for (int i=0; i<nprocs; i++)
11   procs[i] = $spawn proc_f(i);
12 $for (int i=0; i<nprocs; i++) $wait(procs[i]);
```

The code above generates `nprocs` processes, each of which spawns `nthreads` "threads"—the basic structure of a hybrid MPI-threads program. Note the memory hierarchy implicit in the design: variables declared in the body of `thread_f` are "thread-local": they are accessible only by the single thread. Variables declared within the `proc_f` scope are "process-local": they can be accessed by the code of that process and all the threads spawned by the process. Variables declared in the outermost scope can be accessed by all threads and processes.

The pattern on lines 9–12 is so common that CIVL-C provides a convenient short-hand:

```
$parfor (int i: 0 .. nprocs-1) proc_f(i);
```

Scopes are first-class objects in CIVL-C. A value of type `$scope` represents a *dynamic scope*, an object created when control enters the '`{`' that opens the scope and disappears when control reaches the matching '`}`'. The keyword `$here` evaluates to the current scope, hence the statement

```
$scope proc_scope = $here;
```

if inserted between lines 1 and 2 above would give a name to the process scope. In CIVL-C, every scope has its own heap, and the `$malloc` function takes an extra argument specifying the scope in which memory should be allocated. Hence in this example it is possible for each process to use its own separate heap—a faithful model of a real MPI program.

CIVL-C has `$assert` and `$assume` statements, which have their usual meanings. There are universally and existentially quantified expressions (`$forall` and `$exists`), as well as `$lambda` expressions. The latter can be used to initialize an array. For example,

```
double a[n] = (double[n])$lambda (int i) i*2.0;
```

allocates an array of doubles of length n in which the i-th element is set to $2i$, for $0 \leq i < n$.

The type qualifiers `$input` and `$output` can be applied to variables in the global scope. An `$input` variable is read-only and is initialized to an arbitrary value of its type if no initializer is given. (The verifier initializes such a variable with a fresh symbolic constant.) An `$output` variable is write-only. Two programs with the same input/output signature can be *compared* by the CIVL verifier. This checks that on any execution, if given the same input, the two programs will produce the same output.

CIVL-C provides ways to extend the core language. Ordinary functions are one extension mechanism. In CIVL-C, one can also add the function specifier `$atomic_f` to a function declaration. This means that a call to that function will happen in a single atomic step. In CIVL-C (as in C) the actual arguments are evaluated before the call, so their evaluation is not included in the atomic step; the step encompasses the call proper, the execution of the body, and the return. Such a call behaves very much like a new kind of statement.

CIVL-C also allows *system functions*. These are declared with the specifier `$system`, but definitions are not provided as CIVL-C code. Instead, the user provides a Java class which manipulates the state of the program directly. (The idea is similar to Spin's `c_code` facility.) System functions are always atomic. A system function may also have a *guard* which is specified in an annotation of the form `executes_when` *expr*`;` preceding the function declaration. A call to the function is enabled only in a state in which *expr* evaluates to *true*. By default, the guard is `$true`, i.e., a call is always enabled.

The problem with system functions is that verification tools have no idea what they can do. Without further information, a verifier must assume a system function could read or modify any part of the state. For model checkers, this can result in extreme state explosion. This is because model checkers use *partial order reduction* (POR) to reduce the state space explored without sacrificing soundness. POR requires knowledge of when two transitions are *independent*. (Two transitions are independent if, for any state in which they are both enabled, the state resulting from executing both is independent of the order in which they are executed.) Given a state s, for a model checker to restrict the search to a subset T of transitions enabled at s, it is necessary that on any execution departing from s, no transition dependent on a transition in T can occur without a transition in T occurring first [4].

A CIVL-C library developer can specify independence information for an atomic function with an annotation of the form `depends_on` *expr*`;`. One possible value of *expr* is `\nothing`, meaning a call to this function is independent of all transitions. Another is an expression of the form `\access`$(e_1,e_2,\ldots)$, where the e_i are expressions of pointer type. This annotation is understood as follows: given a state s, there is a directed graph G in which the nodes are the objects that exist in s and there is an edge from one node to another if the first object contains a pointer into the second. We say a process p can *access* an object o in s if there is a path in G from an object on p's call stack to o. The annotation specifies that the function call is independent of any transition executed by a process p that can *not* access any of the objects pointed to by the e_i. It is a fact of the CIVL-C language that if p cannot access o in s, then on any execution starting from s, p will never be able to access o unless some process that can access o executes first. Therefore, if there is some set of processes which cannot access the e_i objects, the model checker may be able to ignore the transitions from those processes. (There are other conditions that must be checked, but they do not require annotations.) Recall that the evaluation of actual arguments takes place before the function call; the dependence claim covers only the call, execution, and return—all of which happens as a single atomic step.

A dependence clause may be provided for any atomic function, not just system functions. This is useful when the independence holds for some subtle reason that the model checker cannot determine on its own. Of course, it is possible for a dependence clause to be wrong, i.e., for a function to depend on more than what is specified by the clause. In that case, the CIVL verifier could conclude that a property holds when it does not. However, this mechanism is intended only for "expert developers"—e.g., those that are developing a new front-end—not the end users. Moreover, it is also possible that independence relations that are hard-coded into model checkers are buggy. Placing this information in the source code makes transparent the assumptions about independence. Finally, in some cases unsoundness may be desirable, for example, in bug-finding tools.

3 TRANSFORMATION

The CIVL front-end preprocesses, parses, and merges source files to create a single abstract syntax tree (AST) representing the whole program. For each concurrency dialect, an AST-to-AST transformation then replaces dialect-specific code with equivalent CIVL-C code. In this section we describe the MPI transformation. But first we give a brief overview of MPI.

In threading dialects, a program typically starts with a single thread of control and then explicitly creates and destroys threads. MPI is different. A complete MPI program consists of the code that will be run by a single process. Conceptually, the program is executed by duplicating that code n times (where $n \geq 1$ is specified when the program is launched), with each copy running in its own process. The global variables in the original program essentially become process-local variables. All inter-process communication and synchronization is carried out by explicit calls to functions in the MPI library.

A *communicator* c is an MPI abstraction representing a "communication universe". Conceptually, c comprises an ordered set of m processes ($m \geq 1$). The processes are numbered from 0 to $m-1$; that number is the *rank* of the process in c. A process may belong to multiple communicators, and have a different rank in each one. However, MPI provides one default communicator, `MPI_COMM_WORLD`, which consists of all processes at system startup. The rank in `MPI_COMM_WORLD` can be thought of as a PID. By branching on this PID, process behavior can diverge in arbitrary ways. MPI provides many functions to produce new communicators from old ones.

A process refers to a communicator using an opaque handle, which is an object of type `MPI_Comm` (e.g., `MPI_COMM_WORLD`). All communication operations take an `MPI_Comm` argument. Messages sent on a communicator can only be received using that communicator. Conceptually, a communicator of size m encompasses m^2 FIFO channels—one for each ordered pair (i, j)—for buffering messages sent from rank i to j. (The strict FIFO ordering can be broken by associating integer *tags* to messages and specifying a tag in the receive call.)

In addition to the usual *point-to-point* operations for sending and receiving messages, MPI provides a number of *collective* operations that involve all processes in a communicator: barriers, broadcasts, etc. The collective and point-to-point buffers are completely disjoint.

Figure 1 shows the structure of the CIVL-C translation of an MPI program. The transformation introduces input variables for the number of MPI processes, as well as lower and upper bounds on that number. The CIVL verifier allows the user to specify concrete values for input variables on the command line, so one may verify the program, for example, for any number of processes between 2 and 10. (The default lower bound is 1). The command line arguments (not used in this example) are also represented as input variables.

An object of type `$mpi_gcomm` is a *global communicator object*. This represents the global state of an MPI communicator, including its sequence of member processes, and all of their buffered messages. The variable `_mpi_gcomm_world`

```
1 $input int _mpi_nprocs;
2 $input int _mpi_nprocs_lo = 1, _mpi_nprocs_hi, _civl_argc;
3 $input char _civl_argv[_civl_argc][];
4 $scope _mpi_root = $here;
5 $assume(_mpi_nprocs_lo <= _mpi_nprocs && _mpi_nprocs <= _mpi_nprocs_hi);
6 $mpi_gcomm _mpi_gcomm_world, _mpi_gcomms[]; // global communicators:
7 void _mpi_process(int _mpi_rank) {
8   $mpi_state _mpi_state = _MPI_UNINIT; // also: _MPI_INIT, _MPI_FINALIZED
9   MPI_Comm MPI_COMM_WORLD =  $mpi_comm_create($here, _mpi_gcomm, _mpi_rank);
10  int MPI_Send(void * buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm) {
11    $assert(_mpi_state==_MPI_INIT,  "can't call MPI_Send before MPI_Init");
12    $mpi_check_buffer(buffer, count, datatype);
13    return $mpi_send(buffer, count, datatype,  dest, tag, comm);
14  }
15  [more definitions of MPI functions]
16  [insert original source code here, but rename main _civl_main]
17  _civl_main(_civl_argc,  (char * [_civl_argc])$lambda (int i) _civl_argv[i]);
18  $mpi_comm_destroy(MPI_COMM_WORLD);
19 }
20 int main() {
21  _mpi_gcomm_world = $mpi_gcomm_create(_mpi_root,  _mpi_nprocs);
22  $seq_init(&_mpi_gcomms, 1, &_mpi_gcomm_world);
23  $parfor (int i: 0 .. _mpi_nprocs - 1) _mpi_process(i);
24  $mpi_gcomm_destroy(_mpi_gcomm_world);
25 }
```

Figure 1: Translation of C/MPI program to CIVL-C: high-level structure

stores a reference to the global communicator object for `MPI_COMM_WORLD`. Array `_mpi_gcomms` stores references to all communicators; initially it contains only `_mpi_gcomm_world`. These global communicator objects are allocated in the global scope (shared by all processes), and are essentially the only state data in that scope.

The entire original program is inserted into a function `_mpi_process` (lines 7–19). A process-local variable is introduced to record the "state" of the process, which starts as *uninitialized*, changes to *initialized* when `MPI_Init` is called, and then to *finalized* when `MPI_Finalize` is called. The handle `MPI_COMM_WORLD` is also allocated in the process scope; it wraps a reference to the global communicator object with the rank. CIVL's models of the MPI functions are all inserted in this scope as well, so they can access the PID and other process-local data. The original *main* function is renamed `_civl_main` and is invoked on the command line arguments.

Finally, a new main function is introduced in the global scope. It initializes the global communicators (lines 21–22), spawns and waits for the processes, each of which runs the `_mpi_process` function with a unique rank between 0 and `_mpi_nprocs-1` (line 23), and deallocates the global communicator upon termination (line 24).

The MPI library is implemented using a number of basic utility libraries that provide key data structures, such as the *bundle*. A bundle "wraps up" a region of memory into a single value and is used to represent the data of an MPI message. The final implementation of `mpi.h` uses these lower-level libraries, and is pure CIVL-C code (no system functions).

```
1 typedef struct MPI_Comm {
2   $comm p2p; // point-to-point communication
3   $comm col; // collective communication
4   $collator collator;
5   $barrier barrier;
6   int gcommIndex; // index of corresponding
7                   // global communicator
8 } MPI_Comm;
9 int $mpi_send(void *buf, int count,
10       MPI_Datatype datatype, int dest, int tag,
11       MPI_Comm comm) {
12   if (dest >= 0) {
13     int size = count*sizeofDatatype(datatype);
14     int place = $comm_place(comm.p2p);
15     $message out = $message_pack(place, dest,
16                                  tag, buf, size);
17     $comm_enqueue(comm.p2p, out);
18   }
19   return 0;
20 }
```

Figure 2: CIVL-C definitions of `MPI_Comm` and `$mpi_send`

MPI communicators are implemented using the `$gcomm` and `$comm` structures of the `comm` library. Figure 2 shows the definition of `MPI_Comm`. This structure has fields for two local comm handles (one for point-to-point and one for collective messages), a collator for checking consistency of collective calls on the communicator, a barrier, and the index of the communicator in the global list of communicators.

Figure 2 also shows the auxiliary function used to send a point-to-point message. A negative destination value is used

to represent `MPI_PROC_NULL`; sends to that destination are no-ops. Otherwise, the size of the send buffer is computed, the rank of the calling process in `comm` is obtained, and a message is created and enqueued in the point-to-point section of the communicator. Finally, `MPI_Send`, shown in Figure 1, simply checks that the process has been initialized and that the send buffer contains values of the appropriate type, and then calls this auxiliary function.

The collective functions are defined using auxiliary send/receive operations that access the collective section of the communicator. Simple, deterministic algorithms are used, unlike the case in a real MPI implementation.

4 EXAMPLES

A popular technique for detecting deadlocks in MPI programs is to change all `MPI_Send`s to `MPI_Ssend`s, forcing each send to execute synchronously with the matching receive. It is also known that this is not sufficent for detecting all deadlocks. The example of Figure 3 is particularly challenging for static analyzers because it will only deadlock if certain sends synchronize and others are buffered.

The program consists of 3 processes executing in two phases separated by a barrier. The first phase is a call to function `f`. If all communication is synchronous, the send from process 2 will match the first wildcard receive in process 0, process 2 then syncronizes with process 1, and finally the send from process 1 is matched by the second wildcard receive in process 0. Hence the value returned on process 0 is 0. If the send from process 2 is buffered, the wilcard receives could match the messages in either order, and the value returned on process 0 could be 0 or 1. This value is broadcast from process 0, so all processes now agree on the value of `x`.

The second phase, function `g`, is executed only if `x` is 1. It is a classic "head-to-head send" pattern, and will deadlock if and only if both sends are synchronous.

If all execution is synchronous, `x` is 0 and therefore `g` is not executed, so there is no deadlock. If all messages are buffered, `g` cannot deadlock, so again there is no deadlock. On the other hand, if the sends are buffered in `f` and 1 is returned, and the sends in `g` are synchronized, the program deadlocks.

Figure 4 shows the results of applying CIVL to this example. The command line options specify that *potential*—not just *absolute*—deadlocks should be detected; 3 MPI processes are to be used; and a *minimal* counterexample is to be found, i.e., a path of minimal length from the initial to a violating state. The output indicates that a deadlock was found. The call stacks of all processes are printed, and a *trace file* is saved to disc. The trace can be *replayed*, which guides the execution of the program down the saved path. The output resulting from the print statements makes the deadlocking execution clear.

The second example illustrates an ideal way CIVL could be used to verify exisiting code. Figure 5 shows a toy matrix library that provides a function to print a matrix and a function to multiply two matrices. The library consists of a header file and a sequential implementation. To apply CIVL to this library, a driver must be constructed to generate arbitrary symbolic input matrices (up to some bounded size) and call library functions. The driver is placed in its own translation unit, so no modification to the library code is required.

Figure 6 shows a second implementation of the libary interface. This one uses MPI and a manager-worker pattern to distribute the work [6]. A driver is included in a separate translation unit, as in the sequential version. Figure 7 shows how CIVL can be used to check the equivalence of the sequential and parallel matrix multiplication routines. In this case, equivalence is verified for all matrices with $N \in [1, 4]$, $L = M = 3$ and for `nprocs` = 3 in 9 seconds. The output from the `printf`s shows the symbolic values of variables; the printing does not affect the analysis but can be useful for understanding and debugging.

5 RELATED WORK

There has been much research on verification of MPI programs. Some of the earliest work (e.g., [13, 19, 20]) used the model checker Spin. While Spin's input language, Promela, provides some general concurrency primitives, using the language to model the complexities of an abstract MPI runtime is non-trivial. In that sense, the focus of the early work is similar to that of this paper. The main difference is that Promela is far removed from the C language, so it requires significant manual effort to abstract and transform a C/MPI program into a Promela model. This was ameliorated somewhat in MPI-Spin [19]—an extension to Spin which defines macros corresponding to many of the MPI functions. These functions include the blocking-mode standard point-to-point functions, all the *nonblocking* functions, and the collective functions—a larger subset of MPI than that covered by CIVL. MPI-Spin also provides a primitive symbolic execution facility which has been used to verify functional equivalence of MPI programs with corresponding sequential versions. Still, without pointers, procedures, and many other basic C constructs, producing a useful Promela model of a C/MPI program is difficult and error-prone.

Like Spin, TASS [23] is an explicit-state, stateful model checker, but its input language is a subset of C and MPI. Using symbolic execution, it can verify that properties hold for all possible inputs to a program, within some specified bounds on input sizes; it can also show that two programs are functionally equivalent. However it is MPI-specific and cannot be applied to other concurrency models or hybrid programs. It supports only a very limited subset of C.

ISP [25] and its distributed cousin DAMPI [26] are dynamic model checkers for MPI programs. They use a modified MPI runtime to explore schedules and communication matchings in a systematic way, and can verify properties such as deadlock-freedom. These tools do not perform symbolic execution, so require specific concrete inputs, similar to testing. They are stateless, so each interleaving is executed completely from beginning to end, which can reduce performance relative to

```
/* Receive into int buffer of length n from src */
void recv(int * buf, int n, int src) {
  MPI_Status s;
  MPI_Recv(buf, n, MPI_INT, src, 0, MPI_COMM_WORLD,
           &s);
  printf("Proc %d: received from proc %d:", rank,
         s.MPI_SOURCE);
  for (int i=0; i<n; i++) printf(" %d", buf[i]);
  printf("\n");
}
/* Send an int buffer of length n to rank dest */
void send(int * buf, int n, int dest) {
  printf("Proc %d: sending to proc %d:", rank, dest);
  for (int i=0; i<n; i++) printf(" %d", buf[i]);
  printf("\n");
  MPI_Send(buf, n, MPI_INT, dest, 0, MPI_COMM_WORLD);
}
/* if all sends synchronize, returns 0 on rank 0.
   otherwise, could return 0 or 1 on rank 0. */
int f() {
  int buf;
  if (rank == 0) {
    recv(&buf, 1, MPI_ANY_SOURCE);
    recv(&buf, 1, MPI_ANY_SOURCE);
    return buf;
  } else if (rank == 1) {
    recv(NULL, 0, 2); buf = 0; send(&buf, 1, 0);
  } else if (rank == 2) {
    buf = 1; send(&buf, 1, 0); send(NULL, 0, 1);
  }
  return 0;
}
void g() { // deadlocks iff both sends synchronize
  if (rank == 0) {
    send(NULL, 0, 1); recv(NULL, 0, 1);
  } else if (rank == 1) {
    send(NULL, 0, 0); recv(NULL, 0, 0);
  }
}
int main() {
  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  assert(nprocs >= 3);
  int x = f();
  MPI_Barrier(MPI_COMM_WORLD);
  MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
  if (x) g();
  MPI_Finalize();
}
```

Figure 3: Program `dy_buf_bad.c` may deadlock, but not if all sends are synchronous or if all sends are buffered.

a stateful model checker. On the other hand, they are quite robust and require no restrictions on the input language.

MOPPER [5] verifies deadlock-freedom in *single-path* MPI programs, including those that use `MPI_ANY_SOURCE` (wildcard) receives. It uses ISP to generate a single trace of the program and then reasons about all possible wildcard matchings that could occur in that trace. It does this very efficiently

```
$ civl verify -deadlock=potential -input_mpi_nprocs=3 \
   -min dy_buf_bad.c
... Violation 0 encountered at depth 401:
... kind: DEADLOCK, certainty: PROVEABLE ...
Call stacks:  ...
process 1:
  $mpi_send at civl-mpi.cvl:221.4-16 "$comm_enqueue"
    called from
  MPI_Send at mpi.cvl:87.9-17 "$mpi_send" called from
  send at dy_buf_bad.c:48.2-9 "MPI_Send" called from
  g at dy_buf_bad.c:74.4-7 "send" called from
  _civl_main at dy_buf_bad.c:91.9 "g"  ...
$ civl replay dy_buf_bad.c
Proc 2: sending to proc 0: 1
Proc 2: sending to proc 1:
Proc 1: received from proc 2:
Proc 1: sending to proc 0: 0
Proc 0: received from proc 1: 0
Proc 0: received from proc 2: 1
Proc 1: sending to proc 0:
Proc 0: sending to proc 1:
...
```

Figure 4: Applying CIVL to `dy_buf_bad.c`

by encoding the problem as a SAT formula. In our experience, however, it is rare that for a program using `MPI_ANY_SOURCE` to satisfy the single-path restriction. In a manager-worker pattern, for example, the manager will receive a task from any worker using a wildcard, and then send the next task back to that worker—violating single-path. On such programs, MOPPER's analysis is not necessarily sound.

There are also a number of dynamic checkers for MPI programs, such as MUST [7]. These are the most scalable of all the approaches discussed so far, but are limited to verifying a single run of a program, on concrete inputs.

All of the tools discussed above require a fixed or bounded number of MPI processes, and often bounds on other parameters (such as the size of input arrays). There has been some research into verifying properties for programs of arbitrary scale. Dataflow analyses on "parallel control flowgraphs" that generalize the traditional notions for sequential programs are one approach [1, 3, 18, 24]. While this is an active field of research, we are not aware of any publicly available tools based on it.

Another promising unbounded approach is the use of "session types" and automated theorem proving techniques to verify deadlock-freedom of MPI programs [11, 12, 17]. ParTypes [11] takes this approach. These tools currently require a good deal of manual effort: the user must provide a formal description of the communication pattern used in the MPI program, and insert annotations in the code linking program elements to that description. ParTypes also does not support wildcards (or use of multiple tags), or nonblocking operations. Hence it is restricted to a determinisitic subset of MPI.

The idea of representing a variety of concurrency dialects in a single generic language also arises in the INSIEME compiler framework and its intermediate representation INSPIRE [9].

mm_lib.h:

```
#define T double

/* Print matrix with nrows rows and ncols columns */
void printMatrix(int nrows, int ncols, T *m);

/* Multiply matrices a and b, storing result in c.
 * Dimensions: a:n*l, b:l*m, c:n*m */
void mat_mul(int l, int m, int n, T *a, T *b, T *c);
```

mm_lib.c:

```
void printMatrix(int nrows, int ncols, T *m) {
  for (int i = 0; i < nrows; i++) {
    for (int j = 0; j < ncols; j++) {
      printf("%f ", m[i*ncols + j]);
    printf("\n");
  }
  printf("\n");
}
void mat_mul(int l, int m, int n, T *a, T *b, T *c) {
  for (int i=0; i<n; i++)
    for (int j=0; j<m; j++) {
      T tmp = 0.0;
      for (int k=0; k<l; k++) tmp += a[i*l+k]*b[k*m+j];
      c[i*m+j] = tmp;
    }
}
```

mm_driver.cvl:

```
#include "mm_lib.h"
$input int NB=4, LB=4, MB=4, N, L, M;
$assume(0<N && N<=NB && 0<L && L<=LB && 0<M && M<=MB);
$input T a[N][L], b[L][M]; $output T c[N][M];
int main() {
  double local_c[N][M];
  mat_mul(L, M, N, a[0], b[0], (double*)local_c[0]);
  printMatrix(N, M, local_c[0]);
  $output_assign(c[0], local_c[0], sizeof(double)*M*N);
}
```

Figure 5: Sequential matrix library and CIVL driver

Like CIVL, INSPIRE is based on a general concurrency model, and includes primitives for spawning thread groups, nesting groups, and channel-based communication. Like CIVL, the Clang-based INSIEME front-end translates C programs using a variety of concurrency dialects to INSPIRE. Unlike CIVL, the goals are those of a traditional compiler—optimization and code-generation, and a runtime system—not verification. Moreover, INSPIRE is more "low-level" than CIVL-C, more comparable to the CIVL "model" IR [21]. CIVL-C is intended to be not only an IR, but a language that people will like to read and write directly. Finally, while [9] includes discussion of mapping MPI programs to INSPIRE, at the time of writing INSIEME supports OpenMP and OpenCL but not MPI, so a detailed comparison is not possible. The framework also includes a formal language for specifying transformations of ASTs, which allows for very succinct encoding of transformations [10].

```
void manager(int l, int m, int n, T *a, T *b, T *c) {
  T buf[m]; int t, nprocs; MPI_Status s;
  MPI_Comm_size(COMM, &nprocs);
  MPI_Bcast(b, l*m, MPI_DOUBLE, 0, COMM);
  for (t = 0; t < nprocs-1 && t < n; t++)
    MPI_Send(a+t*l, l, MPI_DOUBLE, t+1, t+1, COMM);
  for (int i = 0; i < n; i++) {
    MPI_Recv(buf, m, MPI_DOUBLE, MPI_ANY_SOURCE,
             MPI_ANY_TAG, COMM, &s);
    memcpy(c+m*(s.MPI_TAG-1), buf, m*sizeof(T));
    if (t < n) {
      MPI_Send(a+t*l, l, MPI_DOUBLE, s.MPI_SOURCE,
               t+1, COMM);
      t++;
    }
  }
  for (int i = 1; i < nprocs; i++)
    MPI_Send(NULL, 0, MPI_INT, i, 0, COMM);
}
void worker(int l, int m, int n) {
  T b[l*m], in[l], out[m]; MPI_Status s;
  MPI_Bcast(b, l*m, MPI_DOUBLE, 0, COMM);
  while (1) {
    MPI_Recv(in, l, MPI_DOUBLE, 0, MPI_ANY_TAG, COMM, &s);
    if (s.MPI_TAG == 0) break;
    for (int j = 0; j < m; j++) {
      out[j] = 0.0;
      for (int k=0; k<l; k++) out[j] += in[k]*b[k*m+j];
    }
    MPI_Send(out, m, MPI_DOUBLE, 0, s.MPI_TAG, COMM);
  }
}
void mat_mul(int l, int m, int n, T *a, T *b, T *c) {
  int rank; MPI_Comm_rank(COMM, &rank);
  if (rank == 0) manager(l, m, n, a, b, c);
  else worker(l, m, n);
}
```

Figure 6: MPI implementation of matrix library

```
$ civl compare -inputL=3 -inputM=3 -spec mm_driver.cvl \
mm_lib.c -impl -input_mpi_nprocs=3 mm_mpi_driver.cvl \
mm_mpi_lib.c
...
X_a[0][0]*X_b[0][0] X_a[0][0]*X_b[0][1] X_a[0][0]*X_b[0][2]
X_a[1][0]*X_b[0][0] X_a[1][0]*X_b[0][1] X_a[1][0]*X_b[0][2]
X_a[2][0]*X_b[0][0] X_a[2][0]*X_b[0][1] X_a[2][0]*X_b[0][2]
...
The standard properties hold for all executions.
```

Figure 7: Verification of equivalence of sequential and MPI matrix libraries

6 EVALUATION

We evaluated CIVL (v1.11.1) in three ways: (1) by performing a "feature comparison" with four leading MPI verification tools—MPI-Spin (v1.0), ISP (v0.3.1), MOPPER (based on ISP v0.2.0), and ParTypes (v1.0.3); (2) by applying CIVL to 5 more realistic MPI applications; and (3) by performing two scaling experiments involving all of the tools. Our testbed is

	Property and test programs	C	S	I	M	P
	An ideal verifier should verify the absence of ...					
DL	deadlocks: `absolute_dl`, `potential_dl`	+	+	+	+	+
AS	assertion violations: `assertion`	+	+	+	-	*
RO	messages that overflow the receive buffer: `rbuf_overflow`	+	+	-	-	+
TM	incompatible message and receive types: `type_mismatch_p2p`	+	+	-	-	-
CM	inconsistency in collective calls: `collective_mismatch`	+	+	+	+	+
FE	failure of an MPI program to be functionally equivalent to a sequential version of that program: `matmat`, `matmat_mw`	+	+	-	-	-
SE	divisions by zero, reads before definition, out of bound indexing, and illegal pointer dereferences: `seq`*`1..4*	+	-	-	-	-
	An ideal verifier should reason soundly about ...					
UB	arbitrary (unbounded) number of processes: `parallel_dot`	-	-	-	-	+
IN	all program inputs (of bounded size): `input_branch`	+	+	-	-	+
DD	control paths with dependencies on message data: `data_depend`	+	+	+	+	-
DB	dynamically changing blocking choices for `MPI_Send`: `dy_buf`	+	+	-	-	-
AR	all matchings at `MPI_ANY_SOURCE` receives: `any_src`	+	+	+	+	-
MP	control paths resulting from wildcard matchings: `not_single_path`	+	+	+	-	-
MT	multiple tags and `MPI_ANY_TAG`: `tags`	+	+	+	+	-
NB	non-blocking operations: `simple_nb`	-	+	+	+	-
MC	multiple communicators: `comm_dup`	+	-	+	-	-
HY	multithreaded MPI ("hybrid") programs: `mpithreads`	+	-	-	-	-
	An ideal verifier should ...					
US	be easy to use (automation): `parallel_dot`, `matmat_mw`	+	-	+	+	-
SC	scale well: `parallel_dot`, `matmat_mw`			*see §6.3*		

Figure 8: Desirable features of a tool for verifying MPI programs and corresponding tests; results of running C=CIVL, S=MPI-Spin, I=ISP, M=MOPPER,P=ParTypes.

an iMac with a 3.50 GHz Intel Quad Core i7-4771 CPU and 32 GB memory. ParTypes was run on a 64-bit Windows 7 VM with 1 GB memory on the testbed (since it only works with Windows); MOPPER was run on a 64-bit Ubuntu 14.04.5 VM on the testbed; the other tools were run on the testbed directly. All artifacts, including source programs, tool output, and instructions for reproducing the experiments, can be downloaded from http://vsl.cis.udel.edu/civl/eurompi17/.

6.1 Feature tests

In Figure 8, we list some of the most important features an MPI verification tool should provide. These include the kinds of properties the tool should check (or dually, the kinds of defects it should detect); the *scope* of program behaviors and constructs about which it should reason soundly; and usability and scalability. Working from this list, we constructed a suite of simple programs to test each feature. Each of these feature tests has at least one correct version and one version exhibiting a specific defect. We applied the five tools to each test program and used the results to conclude whether a tool provides a feature. For MPI-Spin, we constructed Promela models by hand; for ParTypes, we wrote appropriate protocols and added annotations; FE required a small amount of CIVL annotation.

As can be seen from Figure 8, CIVL supports a wide range of features. There are two main reasons for this: (1) the use of symbolic execution, and (2) the high fidelity of the translation from C/MPI source to the CIVL-C model. Consider, for example, TM. The symbolic execution mechanism associates a dynamic type to every value. When the data is decoded from a message, the type of the data is compared to that specified in the receive; if an inconsistency is discovered, an error is reported. Symbolic execution also enables features FE, SE, IN, and DD. The CIVL-C model of an MPI communicator encapsulates complete information about the state of buffered messages: their sources, destinations, tags, contents, and FIFO ordering; multiple communicators are maintained in a list, and can be created and destroyed dynamically, delivering feature MC. The symbolic execution approach does entail certain limitations: the number of processes and the sizes of input data structures such as arrays must be bounded. Moreover, CIVL does not yet support nonblocking operations.

In contrast, ParTypes can verify programs for any number of processes, but it does so for only a very limited subset of MPI. In particular, it does not deal with any-source (wildcard) receives, so the MPI programs it accepts are deterministic: given any input and number of processes, deadlock-freedom and assertion violations can be verified by examining a single synchronous execution. Verifying deadlock-freedom in

	Examples (lines of code)	
Tools	`parallel_dot.c` (73)	`matmat_mw.c` (63)
CIVL	3	13
ParTypes	63	N/A
MPI-Spin	98	350
ISP/MOPPER	3	0

Figure 9: Annotation burden measured by number of lines of additional code

programs with any-source receives is much more challenging, because multiple matchings and execution paths must be explored. Moreover, ParTypes has limited ability to reason about the content of messages. For example, our test `data_depend.c` (in pseudocode) is

```
int a[2];
if (rank == 0) { a[0]=1; a[1]=2; send(1,a); }
if (rank == 1) { recv(0,a);
                 if (a[0]!=1 || a[1]!=2) recv(0,a); }.
```

This program is deadlock-free (since the condition in the `if` statement must be *false*) but we could not find any protocol which allowed ParTypes to verify it.

An MPI program deadlocks "absolutely" when every process is either terminated or blocked at a receive for which no matching send or message is present, and at least one process has not terminated. However, this is not the only way deadlock can occur: a program in which every process is either terminated, blocked at an unmatched receive, or blocked at an unmatched `MPI_Send`, *may* deadlock. This is because any invocation of `MPI_Send` may be forced to synchronize (i.e., block until a matching receive operation is posted) or may be executed by buffering the message. These "potentially" deadlocked states are a superset of the absolutely deadlocked states and are particularly hard to detect.

All of the tools correctly detect simple examples of absolute and potential deadlock. However they diverge on more complicated examples, such as `dy_buf_bad.c`, described in Section 4. This program, which uses an any-source receive, contains a potential deadlock which can only be reached if one send is forced to synchronize and another is buffered. ISP and MOPPER cannot detect this deadlock. This is because they use a *fixed-capacity* channel model: the user may specify that every send must be synchronous, or that every message channel has a fixed positive number of messages it may hold. In contrast, MPI-Spin has a mode in which it makes a nondeterministic choice at each send—to either buffer or force synchronization. This is sound, but scales poorly. CIVL does better: in potential-deadlock-detection mode, it explores all possible states with infinite buffering, but checks for potential deadlocks at each state. The POR scheme is adjusted to be sound for potential deadlock, which means more states are explored than for absolute deadlock, but not as many MPI-Spin explores.

ISP and MOPPER require concrete inputs. If a defect only manifests on certain inputs, they cannot detect the defect unless executed on one of those inputs. In this regard, their approach is closer to testing, though unlike testing, they explore a wide range of possible executions for the given input. CIVL and MPI-Spin use symbolic execution and thus can reason about a wide range of inputs.

One measure of *lack* of usability is the amount of annotations or modeling code that must be written. Figure 9 gives the number of lines of additional such code for two examples. MPI-Spin is the worst by this measure, as it requires the user to write a complete Promela model of the program. ParTypes requires a protocol description and annotations in the original MPI program for matching C constructs and protocol types. CIVL requires only a small amount of extra code to specify bounds on the number of processes, and (depending on the problem) to specify input data. ISP and MOPPER are the best in this regard and required only 3 lines of annotation for specifying input data.

Another aspect of usability is the ability to provide useful feedback when something cannot be verified or a defect is discovered. MPI-Spin is based on Spin and thus provides traces as counterexamples, but the user must figure out how they relate to the original program. ParTypes reports errors as VCC assertion violations, without explanation. ISP, CIVL, and MOPPER provide step by step traces through the original source code; CIVL additionally displays the path condition, and the (symbolic) values of all variables involved in an assertion violation, or the call stacks of each process in a deadlock.

6.2 Verifying realistic MPI programs

In Figure 10, we present the result of using CIVL to verify the selected five real-world programs: 2d-diffusion, matrix multiplication using a manager-worker pattern, an MPI-Pthreads implementation of dot-product, Gaussian elimination on a row-distributed matrix, and a 1d-wave simulation with a subtle defect. For each program, CIVL checks its complete set of standard and MPI-specific properties as shown in Figure 8, including the validity of all assertions, which often involve complex numerical computations. In addition, for those marked with *c*, functional equivalence with a separate, trusted sequential version was also verified.

We use + to denote absence of property violations, and - for the opposite. We record the number of states and transitions explored by CIVL. The scale of each program is controlled by the number of processes `NP` and other input parameters, such as `NSTEPS`, which represents the number of time steps in a simulation.

CIVL was able to obtain the expected result (+ or -) on a range of configurations of these benchmarks. On the most realistic examples, the number of processes used was 4, and in the worst case time was 7 minutes: this occurred for the highly nondeterministic manager-worker matrix multiplication program.

6.3 Scalability

We analyzed how the tools scale by the number of processes (`NP`) while verifying deadlock-freedom for two examples, dot product and matrix multiplication, as shown in Figure 11.

program	SLOC	result	states	transitions	time(s)	scale
`diffusion2d.c` [22]	276	+	626185	626183	160	`NP=4, NSTEPS,NX,NY∈[1,5]`
`matmat_mw.c` [22]^c	89	+	1977538	1982825	355	`NP=4, N,L,M∈[4,5]`
`mpithread_both.c` [2]	96	+	233918	266436	37	`NP=MAXTHRDS=2, VECLEN=5`
`gauss_elim.c`^{c} [22]	295	+	276718	276683	88	`NP=ROW=COL=3`
`wave1dBad.c` [21]	190	-	497	496	5	`NP∈[1,4], NSTEPS,NX∈[1,5]`

Figure 10: Results of verifying C/MPI programs using CIVL. A program marked with *c* was verified by checking functional equivalence with a trusted sequential version.

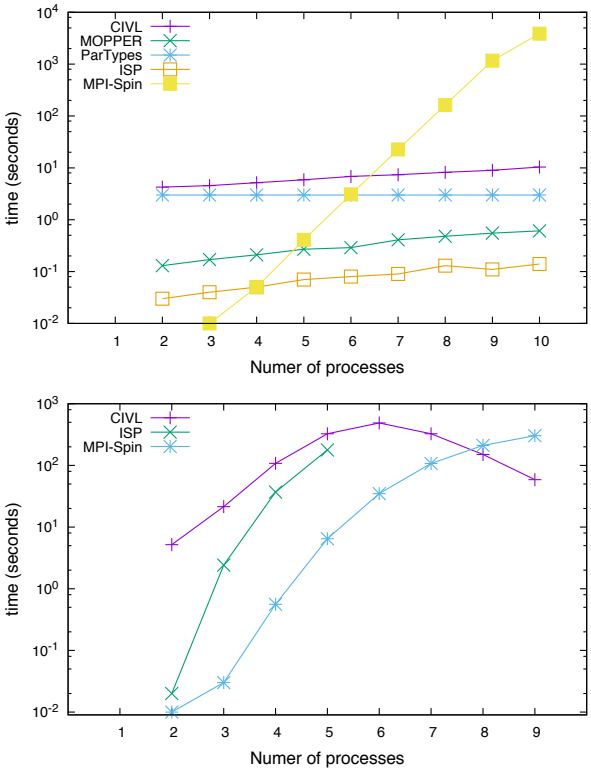

Figure 11: Time to verify deadlock-freedom. Upper: dot product, vector length 50. Lower: manager-worker matrix multiplication, 8×8 matrices.

For the dot product program, which is deterministic, ISP and MOPPER are much faster than CIVL (but CIVL does not require concrete input vectors). ParTypes verifies MPI programs against arbitrary `NP`, so it has constant performance. MPI-Spin executes transitions very quickly, but suffers from an extreme state explosion; this is due to the inserted C code in its model of the MPI library. Spin has no knowledge about the inserted code and must assume that it can access anything, and is therefore dependent on all other transitions, defeating Spin's POR. That explains why MPI-Spin becomes slower than CIVL when `NP` > 7 and cannot complete within 30 minutes when `NP` $= 10$.

The matrix multiplication example uses wildcard receives and is not single-path, so ParTypes and MOPPER cannot be applied. Of the remaining three tools, CIVL is the slowest over most of the domain. However, for `NP` ≥ 6, CIVL's time begins to decrease. This reflects the combinatorial nature of the problem: the initial `NP` tasks are sent to workers in a deterministic order, so as `NP` approaches the number of tasks (8), the amount of nondeterminism decreases. (There is still nondeterminism because the tasks can be returned in any order.) Also, for `NP` $= 6$, ISP cannot complete within 30 minutes. We believe the difference is due to the fact that CIVL is a stateful model checker—it saves the seen states and can backtrack as soon as a state has been seen before—while ISP is stateless. MPI-Spin also saves states but is hampered by the ineffective POR scheme. CIVL takes much longer to execute a transition, but uses a very precise POR algorithm that is aware of the semantics of each MPI function. It therefore explores significantly fewer states than MPI-Spin, and by `NP` $= 9$ is actually faster than MPI-Spin.

7 CONCLUSION

MPI and the C programming language are large and complex languages/APIs. Clearly any attempt to automatically translate C/MPI programs into the language of a generic verification tool will be a challenge. Nevertheless, we have found that CIVL manages this challenge reasonably well. By judicious use of a few basic extensions to the C language, and several general libraries, a model of a significant portion of an abstract MPI runtime has been constructed with moderate effort.

We presented a thorough evaluation of CIVL, including a comparison with other leading tools. We have shown that CIVL provides a number of features not provided by the other tools, including the ability to reason soundly over a wide range of nondeterministic behaviors of the MPI implementation. CIVL can be used to verify not only deadlock-freedom and correct use of MPI, but also the functional correctness of the computation performed by an MPI program. Other tools scale better than CIVL in some cases, but for realistic MPI applications, which often contain nondeterministic constructs, CIVL's performance is often competitive, thanks to the use of stateful model checking and precise partial order reduction techniques.

Future effort will focus on improving the performance of the CIVL verifier through parallelization and other optimizations, and supporting more MPI operations, such as nonblocking communication.

REFERENCES

[1] Sriram Aananthakrishnan, Greg Bronevetsky, and Ganesh Gopalakrishnan. 2013. Hybrid approach for data-flow analysis of MPI programs. In *International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10–14, 2013*, Allen D. Malony, Mario Nemirovsky, and Samuel P. Midkiff (Eds.). ACM, New York, 455–456. https://doi.org/10.1145/2464996.2467286

[2] Blaise Barney. 2017. Lawrence Livermore National Laboratory Message-Passing Interface (MPI) Exercise. https://computing.llnl.gov/tutorials/mpi/exercise.html. (2017). Accessed Aug. 5, 2017.

[3] Greg Bronevetsky. 2009. Communication-Sensitive Static Dataflow for Parallel Message-Passing Applications. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/CGO.2009.32

[4] Edmund M Clarke, Orna Grumberg, and Doron Peled. 1999. *Model checking*. MIT press, Cambridge, MA, USA.

[5] Vojtech Forejt, Daniel Kroening, Ganesh Narayanaswamy, and Subodh Sharma. 2014. Precise Predictive Analysis for Discovering Communication Deadlocks in MPI Programs. In *Proceedings of the 19th International Symposium on FM 2014: Formal Methods - Volume 8442 (Lecture Notes in Computer Science)*, Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun (Eds.), Vol. 8442. Springer, Cham, 263–278. https://doi.org/10.1007/978-3-319-06410-9_19

[6] Willaim Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA.

[7] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. 2012. MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In *International Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11–15, 2012*, Jeffrey K. Hollingsworth (Ed.). IEEE Computer Society Press, Los Alamitos, CA, USA, Article 30, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389037

[8] International Organization for Standardization and International Electrotechnical Commission. 2011. ISO/IEC 989:2011 N1570: Programming Languages – C. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf. (12 April 2011).

[9] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. 2013. INSPIRE: The Insieme Parallel Intermediate Representation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13)*. IEEE Press, Piscataway, NJ, USA, 7–18. http://dl.acm.org/citation.cfm?id=2523721.2523727

[10] Herbert Jordan, Peter Thoman, and Thomas Fahringer. 2014. A High-Level IR Transformation System. In *Euro-Par 2013: Parallel Processing Workshops, Aachen, Germany, August 26–27, 2013. Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 8374. Springer, Berlin, Heidelberg, 647–656. https://doi.org/10.1007/978-3-642-54420-0_63

[11] Hugo A. López, Eduardo R. B. Marques, Francisco Martins, Nicholas Ng, César Santos, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2015. Protocol-based verification of message-passing parallel programs. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, New York, NY, USA, 280–298. https://doi.org/10.1145/2814270.2814302

[12] Eduardo R. B. Marques, Francisco Martins, Vasco T. Vasconcelos, Nicholas Ng, and Nuno Martins. 2013. Towards deductive verification of MPI programs against session types. In Proceedings 5th Workshop on *Programming Language Approaches to Concurrency and Communication-cEntric Software*, Rome, Italy, 23rd March 2013 *(Electronic Proceedings in Theoretical Computer Science)*, Nobuko Yoshida and Wim Vanderbauwhede (Eds.), Vol. 137. Open Publishing Association, 103–113. https://doi.org/10.4204/EPTCS.137.9

[13] Olga Shumsky Matlin, Ewing Lusk, and William McCune. 2002. SPINning Parallel Systems Software. In *Model Checking of Software: 9th Intl. SPIN Workshop (LNCS)*, Dragan Bosnacki and Stefan Leue (Eds.), Vol. 2318. Springer, Berlin, Heidelberg, 213–220.

[14] Message-Passing Interface Forum. 2015. MPI: A Message-Passing Interface Standard, Version 3.1. http://www.mpi-forum.org/docs/docs.html. (4 June 2015).

[15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (March 2015), 66–73. https://doi.org/10.1145/2699417

[16] Susan S. Owicki and David Gries. 1976. An Axiomatic Proof Technique for Parallel Programs I. *Acta Inf.* 6 (1976), 319–340.

[17] César Santos, Francisco Martins, and Vasco Thudichum Vasconcelos. 2015. Deductive Verification of Parallel Programs Using Why3. In *Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015. (EPTCS)*, Sophia Knight, Ivan Lanese, Alberto Lluch-Lafuente, and Hugo Torres Vieira (Eds.), Vol. 189. ACM, New York, USA, 128–142. https://doi.org/10.4204/EPTCS.189.11

[18] Dale R. Shires, Lori L. Pollock, and Sara Sprenkle. 1999. Program Flow Graph Construction For Static Analysis of MPI Programs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1999, June 28 – July 1, 1999, Las Vegas, Nevada, USA*, Hamid R. Arabnia (Ed.). CSREA Press, 1847–1853.

[19] Stephen F. Siegel. 2007. Model Checking Nonblocking MPI Programs. In *Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007, Nice, France, January 14–16, 2007, Proceedings (LNCS)*, Byron Cook and Andreas Podelski (Eds.), Vol. 4349. Springer, Berlin, Heidelberg, 44–58.

[20] Stephen F. Siegel and George S. Avrunin. 2004. Verification of MPI-based software for scientific computation. In *Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004, Proceedings (LNCS)*, Susanne Graf and Laurent Mounier (Eds.), Vol. 2989. Springer, Berlin, Heidelberg, 286–303.

[21] Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, and Michael S. Rogers. 2015. CIVL: The Concurrency Intermediate Verification Language. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. ACM, New York, Article 61, 12 pages. https://doi.org/10.1145/2807591.2807635

[22] Stephen F. Siegel and Timothy K. Zirkel. 2011. FEVS: A Functional Equivalence Verification Suite for High Performance Scientific Computing. *Mathematics in Computer Science* 5, 4 (2011), 427–435.

[23] Stephen F. Siegel and Timothy K. Zirkel. 2011. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science* 5, 4 (2011), 395–426.

[24] Michelle Mills Strout, Barbara Kreaseck, and Paul D. Hovland. 2006. Data-Flow Analysis for MPI Programs. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP '06)*. IEEE Computer Society, Washington, DC, USA, 175–184. https://doi.org/10.1109/ICPP.2006.32

[25] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. 2008. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *Proceedings of the 20th International Conference on Computer Aided Verification (Lecture Notes in Computer Science)*, Aarti Gupta and Sharad Malik (Eds.), Vol. 5123. Springer, Berlin, Heidelberg, 66–79.

[26] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. 2010. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *International Conference on High Performance Computing Networking, Storage and Analysis, SC '10, New Orleans, LA, USA, November 13–19, 2010*. IEEE/ACM, IEEE Computer Society Washington, DC, USA, 1–10. https://doi.org/10.1109/SC.2010.7