Runtime correctness checking for emerging programming paradigms

Joachim Protze, Christian Terboven, Matthias Müller, Serge Petiton, Nahid Emad, Hitoshi Murai, Taisuke Boku

Runtime Correctness Checking for Emerging Programming Paradigms

Joachim Protze
RWTH Aachen University
Aachen, Germany
protze@itc.rwth-aachen.de

Christian Terboven
RWTH Aachen University
Aachen, Germany
terboven@itc.rwth-aachen.de

Matthias S. Müller
RWTH Aachen University
Aachen, Germany
mueller@itc.rwth-aachen.de

Serge Petiton
Maison de la Simulation
Saclay, France
University of Lille
Lille, France
serge.petiton@univ-lille1.fr

Nahid Emad
Maison de la Simulation
Saclay, France
University of Versailles
Versailles, France
nahid.emad@uvsq.fr

Hitoshi Murai
RIKEN AICS
Kobe, Japan
h-murai@riken.jp

Taisuke Boku
University of Tsukuba
Tsukuba, Japan
taisuke@cs.tsukuba.ac.jp

ABSTRACT

With rapidly increasing concurrency, the HPC community is looking for new parallel programming paradigms to make best use of current and up-coming machines. Under the Japanese CREST funding program, the post-petascale HPC project developed the XMP programming paradigm, a pragma-based partitioned global address space (PGAS) approach. Good tool support for debugging and performance analysis is crucial for the productivity and therefore acceptance of a new programming paradigm. In this work we investigate which properties of a parallel programing language specification may help tools to highlight correctness and performance issues or help to avoid common issues in parallel programming in the first place. In this paper we exercise these investigations on the example of XMP. We also investigate the question how to improve the reusability of existing correctness and performance analysis tools.

CCS CONCEPTS

• **Software and its engineering → Correctness**; **Parallel programming languages**; *Software maintenance tools*;

ACM Reference Format:
Joachim Protze, Christian Terboven, Matthias S. Müller, Serge Petiton, Nahid Emad, Hitoshi Murai, and Taisuke Boku. 2017. Runtime Correctness Checking for Emerging Programming Paradigms. In *Proceedings of Correctness'17: First International Workshop on Software Correctness for HPC Applications (Correctness'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3145344.3145490

Correctness'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5127-0/17/11.
https://doi.org/10.1145/3145344.3145490

1 MOTIVATION

In this paper we discuss three research questions: First, which properties of a language or parallelization paradigm are required to enable effective automatic correctness checking and possibly to avoid errors in the first place; second, how can existing specifications or APIs be extended to provide the necessary semantic information for the correctness checking tool; finally, to what extent can existing tools be reused and mapped to such a new programming paradigm.

We exercise this evaluation based on XcalableMP (XMP). XMP is a partitioned global address space (PGAS) approach driven by the Japanese exascale initiative.

As a PGAS aproach, XMP combines paradigms from shared and distributed memory programming; XMP also utilizes pragma based directives as well as API functions and base-language extensions. With these properties, XMP represents most features that a PGAS, distributed or shared memory programming paradigm would provide. This means similar analysis can be applied to many existing or up-coming parallel programming approaches.

For correctness analysis, we distinguish three classes of analysis:

- Static analysis: This kind of analysis can be done by solely inspecting the source code. Typically, this analysis is done while compilation, but could also be part of a stand-alone tool.
- Dynamic, local analysis: This kind of analysis needs information from an actual execution of the program. The number of executing parallel processes is one example for a value that is specific for an actual execution.
- Dynamic, global analysis: This kind of analysis needs information from multiple parallel executing processes. An example are values provided to some collective constructs, that are required to be the same between all participating processes.

If properties of a program can be checked statically, this should be done by the compiler during the compilation of the application. An alternative could be a separate tool, that parses the source code and

applies the analysis. Generally speaking, the effort for implementation, analysis and usage is lower if the analysis is integrated into the compiler, as the language specific parser can be reused and for the user it is a single invocation.

Dynamic analysis could be done by the runtime library implementation—if there is any—or by an external tool. Looking at MPI, that has quite a history in both specification and implementations, it has shown, that runtime implementations often refrain from applying runtime checks to arguments in favor of less runtime overhead and better preformance. We have seen some MPI implementations applying a base set of runtime checks and faulting with some error message. But the best, the programmer gets in these cases is an error code along with a short error description and the name of the MPI function in question. What the programmer doesn't get in most cases is debug level information like code line information, values of all arguments or description for opaque handles. While some MPI runtimes implement some local checks, we didn't see MPI runtimes implementing checks that need distributed or global knowledge.

Creating a third party tool for dynamic analysis of restrictions has several advantages:

- Reduction of implementation effort. A single tool implementation is reusable for multiple interface implementations.
- Runtime implementation can focus on performance and can assume correct usage of the interface.
- An observing tool can add more resources and create a global model of the execution on extra nodes. This helps to provide scalable analysis for collective restrictions.
- A tool would be attached and active during development. Production runs happen without the tool attached and therefore without the overhead of runtime analysis.

Based on our runtime correctness checking tool MUST [3] that analyzes MPI applications, we will try to reuse the tool with it's infrastructure and analyses. With that we investigate the question: To what extent is it possible to map the analysis of new parallel programming paradigms to existing tools?

2 BASIC CONCEPTS OF XMP

In this section we provide a very basic overview on key concepts of XMP [5, 10, 12]. XMP supports Fortran and C as base language. If not stated differently, examples in this paper are Fortran code.

2.1 Directives and Constructs

XMP is quite similar to OpenMP in the way that `#pragma xmp` statements in C respectively `!$xmp` prefixed comments in Fortran can be added to serial source code to express parallelism. The XMP specification differs between directives and constructs. Directives are used to derive descriptors for parallel execution and data. The descriptors created with directives compare with opaque handles in MPI, e.g., the handles for communicators. XMP constructs are used for work sharing, communication and synchronization. Most constructs reference descriptors or subsets of a descriptor to express the applicable domain for the construct.

2.2 XMP Programming Models

A typical XMP program executes in the SPMD model. Like an MPI program, each process starts with the same executable. Processes are called *nodes*, processes can be grouped into *node-sets*. Control flow of the execution is directed by *loop* directives, that work on distributed data and *task* directives that allow to execute code on smaller node-sets that are subset of the *current executing node-set*.

XMP distinguishes between two kinds of data, *global* and *local* data. Global data are data structures that are allocated in a distributed way and processes transparently work on the local share of the distributed data. For data distribution, XMP also provides mechanism to handle a halo area around the local share of the global data. Local data is only accessible locally.

2.3 Global-View Programming

The global-view programming model allows to introduce domain partitioning with distributed data structures that include a halo area just by adding a few pragmas to serial code. The key concepts are *data mapping*, *work mapping* and *synchronization*. Data mapping directives like *distribute* describe how data is distributed across a node-set. Work mapping directives like *loop* describe how work is distributed across a node-set. *Templates* allow to specify a distribution that can be used for both, data and work mapping. For global-view programming, all communication and synchronization is explicitly expressed by the programmer, XMP does not introduce additional communication.

```
 1  !$xmp nodes P(2,2)
 2  !$xmp template T(100, 100)
 3  !$xmp distribute T(block, block) onto P
 4      real G(100, 100)
 5  !$xmp align G(i, j) with T(i, j)
 6  !$xmp shadow G(1, 1)
 7
 8      do iter=1,10
 9  !$xmp loop (i, j) on T(i, j)
10        do i=1,100
11          do j=1,100
12            G(i, j) = .25 * (G(i-1, j-1) + G(i-1, j
       +1) + G(i+1, j-1) + G(i+1, j-1))
13          enddo
14        enddo
15  !$xmp reflect (G) width (/periodic/1,/periodic/1)
16      enddo
```

The example describes an execution with 2x2 processes (l.1). The template T is distributed block-wise in both dimensions (l.2-3). The array G is distributed according to template T and has a shadow, also known as halo, around each block of width 1 (l.4-6). The work is distributed along template T, so that each process works on the local share of G. The stencil access in line 12 reads from local memory and shadow area. After each iteration, the shadow area is updated in line 15.

In comparison to a serial version of the code, only several pragmas are necessary to run the code on distributed memory. Because

of the notion of shadow area, handling the periodic boundary condition is even easier than in serial code.

2.4 Local-View Programming

In the local-view programming model, each process works on local data. For data exchange between nodes, XMP supports *coarray* as defined in Fortran 2008. The first line declares two image arrays of 10 elements on each node of the currently executing node-set. The second line assigns a copy of the tenth image of B to the local image of A.

```
1  real :: A(10)[:], B(10)[:]
2  A = B[10]
```

The XMP specification provides a C language extension to use coarray notation also in C base language. The following code example depicts the usage of coarray notation in C, providing the same semantics as above Fortran code:

```
1  float A[10]:[*], B[10]:[*];
2  A[:] = B[:]:[10];
```

3 DERIVING CORRECTNESS QUESTIONS FROM API SPECIFICATION

The XMP language specification provides a list of restrictions for each of the specified directives and constructs. These restrictions provide the semantic specification of the XMP interface; they can be grouped into the three categories static, dynamic local, and dynamic global property. In this section, we report our evaluation about correctness analysis based on these restrictions. Additionally to these restrictions, we discuss the implications of data race and deadlock in XMP. All parallel programming paradigms face these two issues.

3.1 Examples of Restrictions in the XPM Spec

As an example, we exercise the restrictions that apply to the reflect construct. As sketched in the previous section is the reflect construct used to update the helo of a distributed global array. The XMP specification provides the following restrictions for the reflect construct:

a) The arrays specified by the sequence of *array-name*'s must be mapped onto the *executing node-set*.
b) The reflect width of each dimension specified by *reflect-width* must not exceed the shadow width of the arrays.
c) The reflect construct is global, which means that it must be executed by all nodes in the current executing node-set, and each local variable referenced in the construct must have the same value among all of them.
d) *async-id* must be an expression of type default integer, in XcalableMP Fortran, or type int, in XcalableMP C.

The selection of the executing node-set and the usage of the reflect construct might be in distinct compilation units. Further, both the node-set and array directives can be used with dynamic runtime values. So the restriction a) needs evaluation at runtime. Only local information is needed for this analysis.

Both, the width used in the reflection directive and the width used to specify the shadow of the array can be dynamic values. In that case runtime analysis with local information is necessary for restriction b). In the typical case where static values are used for the width, also static analysis can be applied.

Restriction c) already describes that it defines a global property. To analyze that all nodes of the executing node-set take part on the reflect operation and all nodes use the same values, global knowledge from all nodes is necessary.

Finally, for restriction d) compile time knowledge about the type of the expression is necessary. The type information is typically not contained in the executable, so this cannot be analyzed at runtime, on the other side, all necessary information is available for static analysis.

3.2 Common Threats of Parallel Execution

Parallel programming in general suffers from two major threats, deadlocks and data races. In the following we discuss how these may manifest in XMP.

3.2.1 Potential for Deadlock in XMP. Although the usage of pragmas makes it hard to generate deadlock, it is not impossible in XMP. One situation that can be seen as deadlock is derived from restriction c) in above example. If not all nodes of the executing node-set reach and execute a global construct like reflect, but execute a blocking construct instead, analysis can detect this as a deadlock. Such a blocking construct might be a *wait* construct that waits for a *post* construct on another node. If the other node reached the global construct, there is a cyclic dependency between these two nodes. The following listing depicts the situation where a single node reaches the wait construct, while all other nodes reach the reflect construct.

```
if(xmp_node_num().eq.1) then
  !$xmp wait (P(2),23)
endif
!$xmp reflect (G) width (/periodic/1,/periodic/1)
```

A cyclic dependency can also occur with two or more nodes reaching a *wait* construct as in:

```
!$xmp nodes P(*)
!$xmp wait (P((xmp_node_num()+1)%xmp_num_nodes()),23)
```

All nodes wait for an incoming *post* signal, but no node is ready to reach the post.

3.2.2 Potential for Data Race in XMP. From MPI programming we know two major classes of data race. While the MPI specification does not name these faults data race, it simply states that a program should not behave in that fashion. This is the case in the following classes:

- For non-blocking communication the application should not read or write the memory used as communication buffer before synchronization using wait,
- for one-sided communication (RMA) there are several patterns of unsynchronized remote memory access that could lead to inconsistencies and are specified as undefined behavior.

We can find both of these scenarios in XMP programs. In XMP, non-blocking communication is started by a construct with *async* clause and synchronized by the *wait_async* construct. The following example initiates the shadow area exchange. In the meantime, the nodes execute a stencil operation on the array. Depending on whether the node already got an update of the shadow or not, the node uses the old or the new remote value in the shadow cell.

```
!$xmp reflect (A) width (/periodic/1) async(10)
!$xmp loop on t(i)
do i = 1, 100
A(i) = A(i-1) ...
end do
!$xmp wait_async (10)
```

Remote memory access is available in XMP by using coarray. If multiple nodes access the same coarray image without synchronization, the outcome is non-deterministic. In the following code, all nodes write the local storage of A to the same image 10 of B, that is the image on the 10th node of the node-set:

```
real :: A(10)[:], B(10)[:]
A[10] = B
```

4 CONSTRUCTION OF XMPT API

In the previous section we presented the restrictions described by the XMP specification and how common threats of parallel programming might manifest in XMP programs. From this consideration we now derive a tools interface (XMPT) that is capable of transporting the necessary information to a runtime analysis tool. This tools interface is modeled after experiences from the OpenMP tools interface (OMPT). At runtime initialization time, the XMP runtime library is looking for an available tool, that implements the XMPT initialization function. During the initialization, the tool has the possibility to register callback functions for events of interest to the tool. During the execution of the program, the XMP implementation calls the registered callbacks when execution reaches an event that is coupled to the callback. The arguments provided to the callback transport the information necessary for the tool to apply the analysis.

4.1 Handling XMP Descriptors

As briefly touched in Section 2, directives are used to derive descriptors for *nodes*, *templates*, *distribution* and *arrays*. Detailed analysis of the restrictions provided for directives has shown that most restrictions describe static properties which need compile-time information. A few restrictions might need runtime analysis, if the values are not set statically; most of these restrictions state that the array elements need to be non-negative. This is a property that the runtime library can easily check during execution. On the other side, the XMP specification already provides inquiry functions to query the runtime for properties of descriptors.

Since there is no need to pass information from creation of descriptors to the tool for further analysis and all necessary information is available using inquiry functions, we decided that there is no need to describe events for directives.

But we introduce two new tools interface functions to enable tools to bind internal knowledge about descriptors to the descriptor handle:

```
int xmpt_desc_get_data(xmp_desc_t desc, void**data);
int xmpt_desc_set_data(xmp_desc_t desc, void *data);
```

The tool data field which is bound to the descriptor is initialized to 0 by the XMP runtime. This enables the tool to detect the first appearance of a descriptor within an execution. The tool then builds an internal representation of the descriptor and queries properties about the descriptor from the runtime. Finally, the tool binds the address of the object that stores the internal representation to the tool data field using `xmpt_desc_set_data`. If the descriptor was seen before, the tool directly has access to the internal representation using `xmpt_desc_get_data`.

4.2 Handling XMP Descriptor References

A common pattern in XMP constructs is the use of *nodes-ref* and *template-ref*. While the references can have various forms, they all can be transformed to the normalized form *nodes-name (nodes-subscript [, nodes-subscript]*)* with as many *nodes-subscript*s as *nodes-name* has dimensions. The other representations provided by the XMP specification are short-cuts to provide a more compact way of expression. The information to transport for a tool consists of the descriptor and the subscript description. For the subscript we use the following definition:

```
typedef struct xmpt_subscript_t
{
  int ndims;    /* number of dimensions */
  int omit;     /* flag omitted optional clause */
  int[] lbound; /* lower bound of the subscript */
  int[] ubound; /* upper bound of the subscript */
  int[] marker; /* mark periodic or step */
};
```

The *ndims* entry seems to be redundant, since the dimension can also be derived from the descriptor. But it improves the robustness of this part of the interface, since it describes the length of the subsequent arrays in this structure. For some constructs like the *barrier* construct it is optional to have a clause that uses a descriptor reference; for these constructs, the omit flag would be set on omission. For the three arrays in the structure, an XMPT implementation can decide whether the structure only contains pointers to the arrays—which might directly point to internal data structures—, or the structure contains an fixed-size array with the maximum supported array size—which might be 7 to be compatible with Fortran 90. The XMPT implementation just needs to write down the implementation decision in the `xmpt.h` file.

4.3 Handling XMP Work Mapping Constructs

The work mapping constructs in XMP are the *task*, *tasks*, *loop*, and *array* construct. The *tasks* construct doesn't have any arguments or restrictions, but a tool still needs to know about the tasks region to understand and reason about the concurrency semantics

of construct. The callback invoked for a tasks construct doesn't provide further arguments and only tells about the start and end of the tasks region. The *task* construct allows to limit execution of the task region to a subset of the currently executing node-set, therefore it takes a *nodes-ref* or *template-ref* as argument; the task callback consequently provides a descriptor handle and a subscript:

```
void xmpt_event_task_begin (
  xmp_desc_t desc,        /* descriptor for nodes or
    template */
  xmpt_subscript_t subsc, /* subscript */
  xmpt_tool_data_t* data  /* pointer to store tool
    specific data */
);
```

To support tools in matching begin and end events, all callbacks have a tool data pointer—tool data is defined to be a void pointer—, where the tool can store information at the begin event; the tool gets the information back in the callback for the end event. The other work mapping constructs *loop* and *array* are covered by callbacks similar to the callbacks for the *task* construct.

4.4 Handling XMP Communication and Synchronization Constructs

Most constructs in this category have an optional *async* clause, that takes an *async-id* value as identifier and makes the execution of the construct asynchronous. In this case, the construct is then synchronized by an *wait_async* construct with the same *async-id* value. In favor of a clean interface, XMPT defines two distinct callbacks for the synchronous and asynchronous version of the constructs. Both versions share the same end-event. The following listing shows the callback for the asynchronous version of the callback for the reflect construct, that is used to update the shadow area of an array as discussed in section 2.3:

```
void xmpt_event_reflect_begin_async (
  xmp_desc_t array_desc,    /* descriptor for the
    array to be updated */
  xmpt_subscript_t width,   /* reflect-width */
  xmpt_async_id_t async_id, /* async-id */
  xmpt_tool_data_t* data    /* pointer to store tool
    specific data */
);
```

In this callback, the subscript structure is reused to describe the width of the shadow to update. The width clause allows to only update a part of the shadow area as specified for the array and can be different on the lower and upper end of the array dimension. The marker gives information whether periodic boundary was specified for this dimension.

Since we found that we need to apply dynamic local or global analysis on all arguments provided to the clauses on these constructs, we decided to pass all arguments through the related callback to the runtime tool. Therefore, the other callback functions provide one or two descriptors along with a subscript and additional information specific for the related construct.

4.5 Handling XMP Coarray

With respect to correctness analysis, coarray access is mainly relevant to analyze potential data races or detect the use of uninitialized memory. For these analyses, the tool needs information about the memory access semantics, the order of accesses and about the relevant synchronization. In a coarray assignment the left side performs write, the right side performs read semantics to the coarray. For this case, XMPT provides two events, one for the read access and one for the write access. Furthermore, XMPT provides separate events for local and remote access. Listing 1 shows an example event for remote write access to a coarray. The XMP specification

Listing 1: Example of an XMP coarray event

```
void xmpt_event_remote_write(
  xmpt_coarray_id_t c,
  xmpt_subscript_t subsc,
  xmpt_subscript_t cosubsc,
  xmpt_tool_data_t* data
);
```

does not describe the concept of an descriptor for coarrays, therfore we introduced in XMPT with `xmpt_coarray_id_t` a descriptor for coarrays. Also, XMPT provides inquiry functions to derive information from a coarray descriptor. The *subsc* argument describes the array section of the memory access, the *cosubsc* argument describes image addressing for the remote coarray image. The callbacks for local coarray access do not contain the *cosubsc* argument. A tool can use the *data* argument to correlate the two memory access events of an assignment statement.

4.6 Other Use Cases for XMPT

XMPT was designed to not only be used for correctness checking, but also performance analysis in mind. Therefore most communication events provide a matching pair of begin and end events. These can be used for critical path analysis as well as estimating communication overhead or visualization of communication pattern. The performance analysis tools Scalasca [2, 13] and Extrae [4] announced early prototypes of XMPT support in their tools [1].

5 RUNTIME CORRECTNESS ANALYSIS

As earlier stated, one of the driving questions for this work is to which extent can we reuse existing tools to analyze emerging parallel programming paradigms. For our experiments, we build on our MPI runtime correctness checking tool MUST. For MPI this tool can analyze the execution behavior of an application and check whether the code executes within the parameters provided by the MPI spec. For MPI this covers trivial things like non-negative arguments, but also type matching, resource leaks, data races and deadlock. To answer the question about re-usability, we will revisit in the following the examples given in section 3 and discuss their analysis with MUST.

5.1 Analysis of Constraints from Restrictions

Restriction 3.1 a) requires for a reflect construct, the nodes-set to which the array is mapped to be equal to the currently executing

nodes-set. For this analysis the tool needs to track the executing nodes-set. While the concept of nodes-sets is similar to communicators in MPI, the explicit concept of an executing nodes-set is different. By tracking the events for work mapping constructs, the tool is able to track the currently executing node set. From the array descriptor provided in the reflect event, the tool can derive the nodes-set to which the array binds. For MPI our tool already tracks handles for communicators, groups, data types, and so on. Because of the similarity of nodes-set and communicators—especially Cartesian communicators—, the class which tracks communicators needs only small extension to track nodes-sets.

Restriction 3.1 b) requires the shadow width provided in the reflect construct to not exceed the shadow width declared for the array. This is an integer in range analysis which is also typical in the MPI interface; for example a rank provided in a communication call must be within the size of the provided communicator.

Restriction 3.1 c) requires that reflect—which is a global construct—is executed by all nodes of the currently executing nodes-set and all nodes provide the same arguments. This requirement is the same for MPI collective communication functions like `MPI_Bcast`. We add the information about the executing nodes-set as the "communicator" to the reflect event information and apply a collective completion analysis. This analysis aggregates incoming event information from neighboring processes, compares the provided arguments for consistency, keeps a representative event and makes sure to collect events from all relevant processes, before forwarding the representative event in the analysis tree towards the analysis root.

For other constructs, the restrictions are similar and most analyses map into the discussed classes. A few constructs have the additional restriction that the nodes-set specified by the *nodes-ref* or *template-ref* must be a subset of the currently executing node-set. This analysis is also new for the tool, since the MPI specification does not have such a requirement.

5.2 Analysis of Deadlock

For deadlock analysis MUST uses a two-fold strategy. A distributed state transition system is used to detect deadlock in a distributed and scalable way [7]; a centralized graph-based analysis provides root case information in form of a reduced waiting-for-graph which highlights the deadlock situation [8].

The state transition system processes the incoming events concurrently, that means the analysis selects the next available event from a random node for processing, if a transition rule is applicable for this event. The transition rules are defined in a way, that applicability of a rule implies that an MPI communication could finish. The state transition system detects deadlock, if no transition can be applied, but all processes have an event ready for analysis.

We extend the state transition system to accept and process relevant XMP specific events. This includes global constructs that have equivalent semantics as MPI collective communication calls. Constructs with async clause are handled like MPI-3 non-blocking collective communication [9]; the information is stored and activated by a matching wait construct. The *post* and *wait* construct express point to point synchronization. The analysis processes the post construct like a buffered send, the wait construct like a receive in MPI point to point communication.

For the *lock* and *unlock* construct a new kind of transition is necessary.

A node can enter the lock-acquiring state without precondition, when the lock-begin event is received. The lock-acquired state can be entered, if the lock-begin event is received and no other node is holding the lock. The latter condition enforces that the release event at the other node is processed before the acquired event. By entering the lock-acquired state, the lock is also marked to be owned by this node. A node can release the lock without precondition, when the node receives the unlock event; the lock is marked to be available in this transition.

The graph analysis needs extension to understand the additional dependencies coming from XMP communication semantics.

The graph analysis needs extension to understand the additional dependencies coming from XMP communication semantics.

5.3 Analysis of Data Race

For MPI, MUST applies data race analysis at various levels. At the level of non-blocking communication, the tool analyses concurrent use of overlapping buffers in independent communication calls. We are currently integrating MUST with the data race detection tool Archer [6], which is based on ThreadSanitizer that comes with recent versions of clang and gcc compilers. The goal of this integration is to detect various kinds of data races that come from multi-level parallelism and are currently not covered by any known tool. An example for such a data race is an unsynchronized memory access by a thread, while another thread is using the memory in inter-node communication. We are also working on support of data race analysis for one-sided MPI communication. The challenge here is, that the information about the concurrent remote memory accesses need to be sent around for analysis, while the target process has no idea of the remote memory access. We are confident that we will be able to feed these new MPI analyses with information coming from the XMPT interface and apply the same analysis for XMP applications. The events for coarrays provide similar information as an MPI function like `MPI_Put`, therefore the analysis should map quite easily.

6 CONCLUSIONS

In this paper we have shown the importance of a clear interface definition for a parallel programming paradigm as the basis for automatic correctness analysis. Based on the restrictions described in the interface specification of XMP we derived a classification for correctness analysis. The analyses divide into static analysis, local and global dynamic analysis. Because of the already existing inquiry functions for XMP descriptors, the tools interface XMPT only needs to provide events for work-mapping and communication constructs, but not for housekeeping directives. Regarding reusability, the study has shown that a tool can be reused to the extent to which the concepts of the language map. The concept of an executing nodes-set and also of locks has no related concepts in MPI. For these new concepts, also new analyses need to be introduced. Other analyses can be reused after being slightly adopted. At this point we cannot provide reliable overhead measurements, therefore we refer to the published results for the analysis of MPI applications. Overhead resulting from data race analysis is reported in [11] to be less than

80 % for a worst case benchmark with high-frequent collective communication, overhead for deadlock detection is reported in [7] to be less than 34 % in most cases, but might be higher in case where error is detected. For full data race analysis in multi-threaded XMP applications, we expect the overhead to be in the range of 2-20x as reported for Archer in [6].

ACKNOWLEDGMENTS

The authors would like to thank the funding agencies German Research Foundation (DFG), Agence nationale de la recherche (ANR) and Japan Science and Technology Agency (JST) for making this research possible as part of the German Priority Programme 1648 Software for Exascale Computing.

REFERENCES

[1] 2016. Developer tools for porting and tuning parallel applications on extreme-scale parallel systems. https://jlesc.github.io/projects/tool_pandt_project/. (2016).
[2] 2016. Scalasca web page. http://http://www.scalasca.org/. (2016).
[3] 2016. The MUST Project. https://www.itc.rwth-aachen.de/must. (2016).
[4] 2017. The Extrae Project. https://tools.bsc.es/extrae. (2017).
[5] 2017. XcalableMP web page. http://www.xcalablemp.org/. (2017).
[6] Simone Atzeni, Ganesh Gopalakrishnan, Zvonimir Rakamaric, Dong H. Ahn, Ignacio Laguna, Martin Schulz, Gregory L. Lee, Joachim Protze, and Matthias S. Müller. 2016. ARCHER: Effectively Spotting Data Races in Large OpenMP Applications. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*. 53–62. https://doi.org/10.1109/IPDPS.2016.68
[7] Tobias Hilbrich, Bronis R. de Supinski, Wolfgang E. Nagel, Joachim Protze, Christel Baier, and Matthias S. Müller. 2013. Distributed wait state tracking for runtime MPI deadlock detection. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*. 16:1–16:12. https://doi.org/10.1145/2503210.2503237
[8] 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 *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*. 30. https://doi.org/10.1109/SC.2012.79
[9] Tobias Hilbrich, Matthias Weber, Joachim Protze, Bronis R. de Supinski, and Wolfgang E. Nagel. 2016. Runtime Correctness Analysis of MPI-3 Nonblocking Collectives. In *Proceedings of the 23rd European MPI Users' Group Meeting, EuroMPI 2016, Edinburgh, United Kingdom, September 25-28, 2016*. 188–197. https://doi.org/10.1145/2966884.2966906
[10] Masahiro Nakao, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato. 2012. Productivity and Performance of Global-View Programming with XcalableMP PGAS Language. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012) (CCGRID '12)*. IEEE Computer Society, Washington, DC, USA, 402–409. https://doi.org/10.1109/CCGrid.2012.118
[11] Joachim Protze, Tobias Hilbrich, Andreas Knüpfer, Bronis R. de Supinski, and Matthias S. Müller. 2012. Holistic Debugging of MPI Derived Datatypes. In *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*. 354–365. https://doi.org/10.1109/IPDPS.2012.41
[12] XcalableMP Specification Working Group. 2016. XcalableMP Language Specification. http://www.xcalablemp.org/specification.html. (2016).
[13] Ilya Zhukov, Christian Feld, Markus Geimer, Michael Knobloch, Bernd Mohr, and Pavel Saviankou. 2015. Scalasca v2: Back to the Future. In *Proc. of Tools for High Performance Computing 2014*. Springer, 1–24. https://doi.org/10.1007/978-3-319-16012-2_1