MYX: Runtime correctness analysis for multi-level parallel programming paradigms

Joachim Protze, Miwako Tsuji, Christian Terboven, Thomas Dufaud, Hitoshi Murai, Serge Petiton, Nahid Emad, Matthias Müller, Taisuke Boku

MYX: Runtime Correctness Analysis for Multi-Level Parallel Programming Paradigms

Joachim Protze, Miwako Tsuji, Christian Terboven, Thomas Dufaud, Hitoshi Murai, Serge Petiton, Nahid Emad, Matthias S. Müller, and Taisuke Boku

Abstract In recent years the increasing compute power is mainly provided by rapidly increasing concurrency. Therefore, the HPC community is looking for new parallel programming paradigms to make the best use of current and up- coming machines. Under the Japanese CREST funding program, the post-petascale HPC project developed the XcalableMP programming paradigm, a pragma-based partitioned global address space (PGAS) approach. To better exploit the potential

J. Protze (✉)
RWTH Aachen University, Aachen, Germany
e-mail: protze@itc.rwth-aachen.de

M. Tsuji
RIKEN-CCS, Kobe, Japan
e-mail: miwako.tsuji@riken.jp

C. Terboven
RWTH Aachen University, Aachen, Germany
e-mail: terboven@itc.rwth-aachen.de

T. Dufaud
University of Versailles, Versailles, France
e-mail: thomas.dufaud@uvsq.fr

H. Murai
RIKEN-CCS, Kobe, Japan
e-mail: h-murai@riken.jp

S. Petiton · N. Emad
University of Versailles, Versailles, France
e-mail: serge.petiton@univ-lille.fr; nahid.emad@uvsq.fr

M. S. Müller
RWTH Aachen University, Aachen, Germany
e-mail: mueller@itc.rwth-aachen.de

T. Boku
University of Tsukuba, Tsukuba, Japan
e-mail: taisuke@cs.tsukuba.ac.jp

H.-J. Bungartz et al. (eds.), *Software for Exascale Computing - SPPEXA 2016–2019*, Lecture Notes in Computational Science and Engineering 136,
https://doi.org/10.1007/978-3-030-47956-5_18

concurrency of large scale systems, the mSPMD model was proposed and implemented with the YvetteML workflow description language. When introducing a new parallel programming paradigm, good tool support for debugging and performance analysis is crucial for the productivity and therefore the acceptance in the HPC community. The subject of the MYX project is to investigate which properties of a parallel programming 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 XcalableMP and YvetteML.

1 Introduction

Exascale systems are expected to consist of tens of thousands of compute nodes, complemented by specialized accelerators, resulting in system architectures which are heterogeneous on multiple levels. Such architectures challenge the programmer to write multi-level parallel programs, which means employing multiple different paradigms to address each level of parallelism in the system [2]. This ranges from inter-node parallelism in the form of distributed memory parallelism, over shared-memory parallelism to exploit multi-core processors and acceleration units, to vector-style parallelism to target corresponding hardware units. The long-term challenge is to evolve existing and develop new programming models to better support the application development on exascale machines. For different domains and different abstraction levels, various programming models have gained momentum. While there is ongoing research on how to make the currently predominant HPC programming model—namely MPI+X—scale well on such systems, the emerging and more high-level PGAS programming models have shown to deliver high productivity for users and certain types of codes [10]. The JST-CREST funded post-petascale HPC project developed the XcalableMP (XMP) programming paradigm, which combines local and global view PGAS concepts.

The multi-level programming paradigm FP3C [13] as described later in this paper is a solution for post-petascale systems targeting a huge number of processors and the attached acceleration devices. Programmers can express high-level parallelism in the YvetteML (YML) workflow language and employ parallel components written in SPMD programming paradigms like XMP or MPI. Since YML drives and executes multiple SPMD tasks at the same time, this is characterized as mSPMD.

The MYX project aims to combine the know-how and lessons learned of different areas to derive the input necessary to guide the development of future programming models and software engineering methods. Therefore we are developing correctness checking techniques for the XMP programming paradigm and make this analysis also available for the multi-level programming paradigm FP3C.

The contributions of this work are:

- Identify possible correctness issues of XMP applications,
- define an XMP tools interface to provide runtime state and event information to runtime tools,
- extend MUST by XMP specific runtime correctness analyses,
- extend YML to soundly support innovative numeric techniques like UCGLE, and
- provide a workflow to analyze YML+XMP applications driven by the FC2P framework.

The structure of the remaining paper is as follows. In Sect. 2 we introduce the concept of runtime correctness checking and provide an overview of the general implementation of MUST. In Sect. 3 we provide a brief overview of the concepts of the XMP programming paradigm based on an example code. In Sect. 4 we highlight potential correctness issues in XMP applications and what information is needed to analyze those errors. To perform such runtime analysis, a tool like MUST needs state and event information from the XMP runtime system. In Sect. 5we, therefore, provide a brief overview of the tools interface that we proposed as an extension of XMP to the XMP specification consortium. In Sect. 6 we introducethe concepts of the YML workflow language based on an example code. The uniteand conquer method described in Sect. 7 represents an example use case for an mSPMD program implemented with YML for the coarse-grained parallelism and XMP for the implementation of the individual YML tasks. To implement such a method, some extensions of YML are necessary, we also discuss the implications for correctness. In Sect. 8 we present the FP2C framework, which provides a YML+XMP implementation targeted to HPC systems. As MPI is basically the standard for distributed memory HPC systems and those systems also prefer fixed-width jobs, i.e., jobs with a fixed number of processes the FP2C framework is implemented with MPI and dynamically launches MPI processes to fill the requested number of process slots. Finally, in Sect. 9 we present the challenges and solutions to provide runtime correctness analysis in MUST for such a dynamicruntime system.

2 Runtime Correctness Analysis for Parallel Programs

Other than serial programs, parallel programs are affected by non-determinism as an effect of the concurrent execution of multiple threads or processes. For defect programs, this non-determinism can manifest as data races or deadlocks which are not known in serial programming. Different approaches to identify and remove the defects in those programs include static code analysis, model checking, and runtime or post-mortem analysis. Here we want to discuss runtime correctness analysis, where the error detection is performed during the execution of the program.

MUST performs runtime correctness checking for MPI parallel applications. The application developer executes the application under the control of MUST, which

checks at execution time whether the usage of MPI is valid according to the MPI specification. For MPI applications we have shown, that the execution overhead for such runtime analysis is below 20% for the typical use case [8]. Although we aim at complete coverage of the MPI specification, the focus is currently on communication functions.

The overhead of runtime correctness analysis depends significantly on the granularity of the analysis. For MPI, the granularity is quite coarse-grained: there is typically a lot of calculation between MPI function calls which is not analyzed. For data race detection in multithreaded applications, the granularity of analysis is much more fine-grained, as each individual memory access is subject to analysis. Therefore we see a two to hundredfold runtime overhead for data race analysis.

2.1 Runtime Analysis in MUST

For runtime analysis of distributed memory applications, we distinguish three kinds of analyses as shown in Fig. 1. *Local analysis* only needs information from a single application process and can be performed within the application process to avoid unnecessary data transfer. In a multi-threaded application, this analysis potentially needs information from multiple threads. We, therefore, spawn an additional

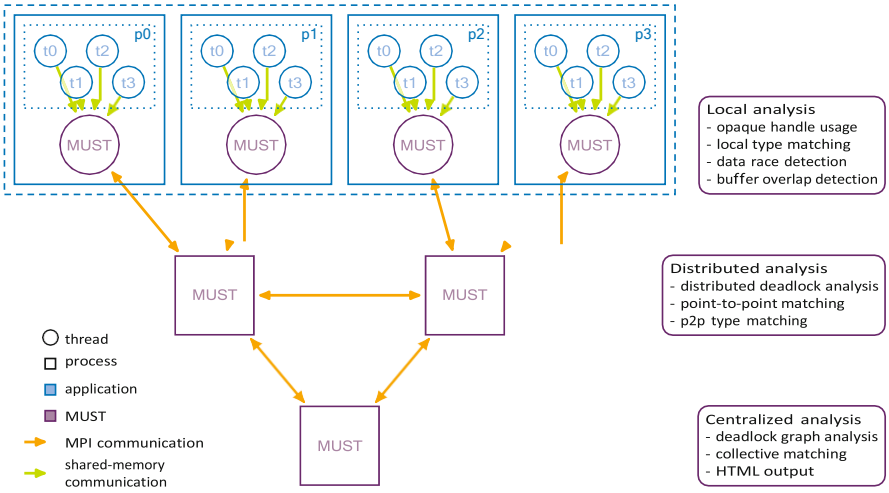

Fig. 1 MUST applied to a hybrid parallel application with four processes $p_0 \ldots p_3$ and four threads $t_0 \ldots t_3$ each. MUST spawns an extra tool thread in each process, which communicates with the additional tool processes using a tree-based overlay network (TBON). Each analysis is performed on the first tool layer that has sufficient information to perform the specific analysis.In the typical setup, communication between MUST processes is performed using MPI. The communication between the threads uses shared memory communication

analysis thread to have all necessary information available. In a single-threaded application or when only the master thread communicates, the local analysis is performed in the application thread and no additional tool thread is spawned. Asan example, local type matching compares compile-time information about types used by the application on variable declaration with the runtime information on the corresponding MPI data types used in communication [9].

Distributed analysis needs information from more than one application process. For scalability reasons, the analysis is distributed in the analysis tree and performed in the node where sufficient information is available. As an example, the distributed deadlock detection analysis runs on the parent node of each application process. It works with a distributed state transition system, which is fed with send and receive information for the specific process, as well as completion notification for collective communication [8].

Centralized analysis needs global information from multiple or all application processes. For scalability, a tree reduction analysis is applied where possible, so that the centralized analysis is just the last step in such reduction. An example of such tree reduction is collective matching analysis, where each tree node comparesthe parameters in collective communication for all child nodes and finally passes one representative to the parent tree node. An example of completely centralized analysis is graph-based deadlock analysis, which we use to visualize the circular dependencies causing a deadlock, but also to verify the presence of a deadlock in the time-out based deadlock detection. This analysis needs information on all pending communication operations but is only executed when a deadlock is detected or suspected.

2.2 Underlying Tool Infrastructure of MUST

MUST intercepts events in the execution of a targeted application to apply the analysis based on the information from these application events. Initially, these events were MPI function calls, but this is now extended to OpenMP and XMP events that are delivered to registered callback functions. Within an application process or thread, the tool can only get active when such an event is delivered. MUST builds on a tree-based overlay network (TBON) communication subsystem, as depicted in Fig. 1. Since the tool cannot make any assumptions, when it will be active on an application process or thread, those nodes communicate only towards the root of the tree. For use cases as point-to-point matching and distributed deadlock detection, the classical TBON communication scheme was extended by horizontal communication within a tool layer. In the current default configuration of MUST, all processes are started together as MPI processes with a common `MPI_COMM_WORLD`. Using the MPI interception layer PMPI, MUST then makes sure, that only the processes intended to execute the application code will continue execution. The tool processes remain in a run loop which waits for incoming events to process. Whenever application code uses `MPI_COMM_WORLD` in an MPI

function call, this communicator is replaced by a sub-communicator representing the application processes. This is transparent for the application, which will never see the additional analysis processes. For performance reasons, the tree layout—including the number of layers and placement of analysis functions— is hard-coded and compiled into a specific instance of the tool. This means that with the current tool infrastructure a dynamic reconfiguration of the tree is not possible at runtime.

3 XcalableMP

XcalableMP (XMP) is a directive-based language extension for Fortran and C languages. Like in OpenMP, parallelism is introduced by the use of directives. If all the directives are ignored by the compiler, a serial program with the same semantics and results should remain. XMP targets parallel programming for distributed memory systems, in contrast to OpenMP targeting shared-memory parallelism. The implementation of XMP in the OmniCompiler is a source to source transformation, which translates the directives into additional code and calls into the XMP runtime library. The XMP runtime library communication is mainly performed using MPI. Therefore, it is in general also possible to use MPI communication in XMP programs or link a library written with XMP into an MPI application.

Listing 1 shows an example of an XMP distributed parallel source code. This example assumes the execution with four processes which are assigned to the nodeset `p`. In XMP, the distribution of a virtual `array` onto nodes is defined as a `template`. An array is then associated with a template using the `align` statement. This defines the distribution of the array over the nodes. Also, the distributed processing is defined by applying the template to a `loop` directive. The iterations of the loop are executed on the different processes according to the distribution assigned to the template `t`.

For parallelization of stencil codes on distributed memory systems, there is typically the need to use a halo as temporary copy for the calculation of the boundary in the local share. XMP supports such behavior with two directives. The `shadow`

Listing 1 Global-view programming: distribute data and work to processes (*nodes*)

```
#pragma xmp nodes p(4)
#pragma xmp template t(0:11)
#pragma xmp distribute t(block) onto p
int B[12];                              // Data Mapping
#pragma xmp align B[i] with t(i)

#pragma xmp loop (i) on t(i)
for(i=0; i<12; i++){                    // Work Mapping
  B[i]=B[i]*2;
}
```

Listing 2 Local-view programming: use coarray notion in C code

```
int a[10]:[*], b[10], c[10];
#pragma xmp nodes p(4)

int me = xmpc_this_image();
int right = (me + 1) % 4, left = (me + 3) % 4;
for(i=0; i<10; i++){
  b[i]=me;                  // initialize
}
a[:]:[right] = b[:];        // put to right neighbor
xmp_sync_all(NULL);         // barrier and sync memory
b[:] = a[:];                // local copy
c[:] = a[:]:[left];         // get from left neighbor
```

directive allows to specify the width of the halo for a specific distributed array and the `reflect` directive is to perform the update of the halo.

The functionality described so far is called as global-view programming in XMP. In global-view programming, the application programmer does not need to care where data is located. The array is transparently distributed and accessed. In the suggested workflow, the work is performed where the data is located.

Furthermore, XMP extends Coarray Fortran and makes this functionality also available in C. In XMP this is called local-view programming. To access memory on a different process in local-view programming means to explicitly specify the target process, that holds the image of interest.

The code example in Listing 2 demonstrates how XMP allows using the concept of Coarray in C code. The array `a` is declared as a Coarray of size 10 with an image on each process. The image selector is separated with a colon in the declaration. A classical, local array `b` is initialized in the for loop and then assigned to the Coarray `a`. The slice notation `b[:]` similar to Fortran allows assigning a whole array at once. The assignment to the remote image `right` is semantically a *put* operation. Therefore, the slice notion is not only a convenience feature but allows to perform a single memory transfer in comparison to a for loop, which assigns each array element individually.

4 Correctness Checking for XMP Programs

In this section, we will discuss possible programming errors in XMP applications and how to detect those errors in the code.

4.1 Programming Errors in XMP Programs

For global-view programming we identified a range of possible programming errors that violate restrictions provided for the specific XMP construct. As an example, the `barrier` construct has the following restriction:

- The nodeset specified by the `on` clause must be a subset of the executing nodeset.

The code example in Listing 3 violates this restriction, because the `task` construct limits the executing nodeset to process `p(0)`, while the nodeset specified by the `on` clause is the complete nodeset `p`. The code presents also the MPI idiom with the same semantic. Since only the process with the rank number 0 reaches the barrier, this will finally result in a deadlock for the MPI code.

Besides violations against restrictions imposed by the XMP specification, we also identified possible data races for asynchronous communication. The code example in Listing 4 initializes a distributed array, which is defined with a surrounding halo. The update of the halo is performed asynchronously, because of the `async` clause.

Listing 3 Only a subset of processes participates in a collective barrier operation

```
#pragma xmp task on p(0)
{
  printf("Only executed on rank 0");
  #pragma xmp barrier on p
}
if(rank == 0)
{
  printf("Only executed on rank 0");
  MPI_Barrier(MPI_COMM_WORLD);
}
```

Listing 4 Asynchronously updating the halo can result in a data race

```
int a[16];
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t onto p
#pragma xmp align a[i] with t[i]
#pragma xmp shadow a[1]

#pragma xmp loop (i) on t[i]
for(int i=0;i<16;i++)
  a[i] = i * 4;

#pragma xmp reflect (a) width (/periodic/1) async(100)
for(int i=0;i<16;i++)
  a[i] = a[i-1] + a[i+1];
#pragma xmp wait(100)
```

Before the asynchronous execution is finished, the stencil access already reads this halo value. It is unclear whether the old or the new value is read, but also whether the old or the new value is sent to the neighbor.

Another possible error arises from the use of the `orthogonal` clause with the `reflect` construct. With this clause, only the orthogonally adjacent halo will be updated, but not the corners or edges of a multidimensional halo. For most stencil applications, this is sufficient and saves a lot of communication, because the corner is located on a different process. If the application nevertheless needs and accesses the value, it will see an uninitialized value there.

For local-view programming, the main risk is data race in the remote memory access. This can occur if different processes access the same memory in the same image without synchronization and one of them modifies the memory. Revisiting the code example in Listing 2, we would have a data race on `a` if we remove the function call to `xmp_sync_all`. The `left` neighbor updates the local image `a` of a process, which would then not be synchronized with the local access to `a` and also not synchronized with the read by the `right` neighbor.

4.2 Correctness Analysis for XMP Programs

Since XMP programs translate to MPI programs in the implementation provided by the omni-compiler, we can apply native MPI correctness analysis to XMP programs. For an application which implements Listing 3, MUST detects a deadlock between an `MPI_Barrier` implementing the XMP barrier directive and an `MPI_Barrier` inserted by the XMP compiler at the end of the task region. Figure 2 shows the deadlock as reported by MUST. The left diagram depicts the cyclic dependency detected by MUST, where `MPI_Barrier` is called with two different communicators. The MUST report provides further details about these communicators, which are created by the XMP implementation. The right diagram provides additional information on the function stack for the two conflicting `MPI_Barrier` calls. `_XMP_Barrier` is the XMP runtime implementation for any explicit or implicit barrier. The graph also shows that this XMP barrier is calledfrom two different locations—lines 15 and 23—in the source code, although the original source code only has 15 lines.

This example emphasizes, that correctness analysis for XMP applications can be done at the MPI level, but is not too useful for the application developer. In other previous work [1, 11] we have seen that we can achieve better results concerning precision and recall if we base the analysis on the semantics of the high-level parallel programming paradigm. Furthermore, the analysis at a higher abstraction level can help to provide more meaningful error reports. In the following, we will see how this applies to XMP.

Analysis in Global-View Programming For the errors, where XMP code might violate restrictions imposed by the XMP specification, we distinguish between static

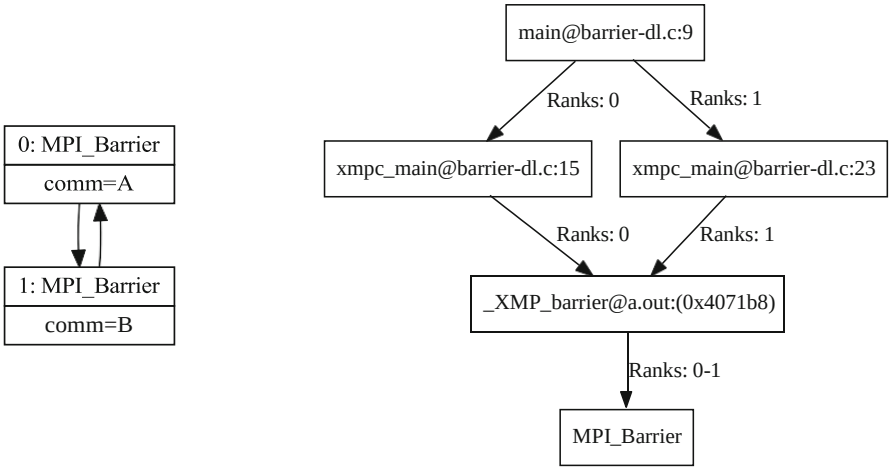

Fig. 2 Deadlock detected by MUST for Listing 3 when only looking at MPI

and dynamic properties of the code. Most of the restrictions on the standalone declarative directives like `nodes`, `template`, or `align` have an impact on static properties of the code. Those restrictions include self-reference in a declaration or lexical name conflicts of handle names with other symbols in a scoping unit. Such restrictions should be enforced by the compiler and result in meaningful compile-time error messages.

Restrictions on dynamic properties depend on the specific value a variable has at runtime. We distinguish between those that involve only the local process and those that involve multiple or all processes. For global-view programming the latter is only the case for collective operations, which require consistent clauses and values among all contributing processes. As for MPI collective communication functions, we analyze this as a reduction analysis, where each node in the TBON compares all incoming events and passes one representative event to the parent node.

All other restrictions on dynamic properties can be analyzed locally. Various XMP constructs have the same restriction as mentioned in Sect. 4.1 for the `barrier` construct. The *nodeset* used for the construct needs to be a subset of the currently executing nodeset. The executing nodeset is the set of nodes executing the current XMP region. The `loop` construct as well as the `task` construct allowto restrict the currently executing nodeset. To perform runtime analysis for such a subset requirement, an analysis tool needs to understand the concept of nodeset and how they can be derived. Listing 5 provides some examples for slicing nodesets. The nodeset `p` consists of the eight processes executing the application, it is also called *entire nodeset*. The nodeset `q` skips the first node in `p` and recruits nodes twoto five from `p`. The nodeset `r` is a two-dimensional nodeset, which can be used from two-dimensional domain decomposition. The nodeset `s` is also two-dimensional, but uses only every other node in `p`. Now, checking whether `q[2]` is subset of `s`

Listing 5 Deriving nodesets in XMP

```
#pragma xmp nodes p(8)                  // entire nodeset
#pragma xmp nodes q(4)=p(2:5)           // slicing nodes 2-5
#pragma xmp nodes r(2,4)=p(1:8)         // 2-dimensional nodeset
#pragma xmp nodes s(2,2)=p(1:8:2) // skip every other node
```

is a non-trivial question. One method to perform the subset analysis is to expand the given nodeset to the entire nodeset while marking participating nodes. This expansion is not scalable for large numbers of processes, but for some comparison of nodesets we cannot avoid the full expansion.

Analysis in Local-View Programming For local-view programming, our main focus is on data race detection in remote memory access. A data race is commonly understood as concurrent access of multiple execution entities to the same data in memory while at least one access is writing to memory. Concurrent access implies that there is no synchronization between the two memory accesses. We can observe two ways of access to a coarray in XMP, both can be found in the code example in Listing 2. The remote image access denoted by `a[:]:[target]` has write semantics on the target memory for the put operation and read semantics for the get operation. The local image access denoted by `a[:]` or by `a[1]` has the memory access semantics as suggested by the base language. In general, an application might access and modify the local image through a pointer to the local image. Especially, when the array is passed to a library, as a linear algebra library, the access to the local image is out of control of the XMP compiler or runtime system. To detect data races on remote memory access, we need to instrument the local memory accesses as well as tracking all remote memory accesses. Since the conflicting memory access might occur in the library, also the library needs to be instrumented for the runtime data race analysis. This is particularly difficult if the library is only available as a binary.

For data race analysis in MUST, we build on ThreadSanitizer as logging and analysis backend. Memory access instrumentation is performed by clang or GNU compiler during compilation. In addition, we provide high-level synchronization, memory access, and concurrency semantics into the ThreadSanitizer analysis. Therefore we extend the annotation interface used by ThreadSanitizer and Valgrind to feed all necessary information into the analysis. An access to the local image by a remote process should be seen concurrent to any previous access by a different process, that is not synchronized with the current access. Synchronization in XMP is possible with global synchronization, e.g., using the `sync all` directive respectively an `xmp_sync_all()` call, or point to point synchronization, e.g., using the `sync image` directive respectively an `xmp_sync_image()` function call.

5 Tools Interface for XcalableMP

To provide XMP specific runtime information to analysis tools, we designed and implemented the XMP tools interface—XMPT. The interface builds on experiences from the OpenMP tools interface. As an example, the specific environmental variable `XMP_TOOL_LIBRARIES` allows loading an XMP specific tool, when the XMPT interface is available during application execution. This imitates the OMPT specific environmental variable `OMP_TOOL_LIBRARIES` and allows building portable tools, which dynamically adapt to the parallel programming paradigm used by a program.

During startup of an XMP application, the XMP runtime tries to find an XMPT tool, which is identified by the exported function `xmpt_initialize`. Other than in OMPT we don't need a three-way handshake for tool initialization, as the XMP runtime doesn't need to adopt the own initialization in case an XMPT tool is present. Once a tool is found, the XMP runtime calls this function and the tool has the chance to register callbacks for certain XMP events. The current implementation of XMPT provides callbacks for all global-view directives and constructs as well as for coarray memory access and synchronization in local-view programming.

The data mapping identifiers like *node-name*s and *template-name*s are identified by their opaque XMP descriptor handles. To recognize such a descriptor and store information on the descriptor, the XMPT interface allows binding tool data to each XMP descriptor.

The OpenMP specification restricts the OMPT tool to only use OMPT runtime functions, but not to call OpenMP runtime routines like `omp_get_num_threads`, nor to use OpenMP pragmas to implement OMPT callback functions or signal handlers. Without this restriction, the OMPT tool might cause a deadlock in the execution of an OpenMP application, because the OpenMP runtime could hold a lock that it tries to acquire again when the OpenMP runtime function is called. The main difference in this particular aspect is that XMP is initialized explicitly at an early point in the execution by calling `xmp_init`, while OpenMP implementations tend to lazy initialize when the first OpenMP construct or runtime routine is called. For thread-safe initialization, the OpenMP runtime might acquire an initialization lock at any entry to the runtime.

For XMPT there is no restriction on the use of XMP runtime functions so that an XMPT tool can use the variety of inquiry functions to collect all necessary information about the opaque XMP descriptor handles. This allows to query information on XMP specific entities on demand and avoids to transport all available information as arguments to the callbacks. This makes the interface both more compact and more efficient.

6 YvetteML

YvetteML (YML) is a workflow description language for technical or scientific calculation that describes dependencies among tasks.

YML interprets low source code and dependencies between tasks to generate the indicated DAG and execute the task according to the DAG. YML of the original casing is, P2P tasks that are written sequentially in the language it was assumedto run in an environment or a small cluster, of tasks written in a parallel language. By using YML it became possible to run the application on a large scale system. We also developed middleware for porting. The middleware used to implement the mSPMD programming model is OmniRPC-MPI [13]. It provides Remote Procedure Call (RPC) based on MPI and is an extension of the library OmniRPC. Our OmniRPC-MPI middleware is a workflow scheduler to control remote programs which are created for task execution by use of `MPI_Comm_spawn` on request. Control and data flow is implemented using MPI functions such as and `MPI_Send`and `MPI_Recv`.

Listing 6 shows a simple example for a YML program. It invokes a function `add` which takes two double arguments and on return provides the sum in the first argument. The execution starts sequentially, at first *`result`* = 1 + 2 is calculated. Then execution continues parallel with three concurrent code blocks, separated by `//`. The first code block is just to satisfy the dependency on `ping[0]`, the other two concurrent code blocks execute five parallel iterations each. We can interpret each of the iterations as a task, the `wait` and `notify` statements express dependencies. The YML interpreter generates a DAG as depicted in Fig. 3, where each parallel block and each parallel loop iteration becomes a task. Each of the leaf tasks executes

Listing 6 YvetteML example

```
compute add(result, 1.0, 2.0);   # result <- 1 + 2
par
    notify(ping[0]);
//
    par(i:=0;4)
    do
        wait(ping[i]);
        compute add(result, result, result);
        notify(pong[i]);
    enddo
//
    par(i:=0;4)
    do
        wait(pong[i]);
        compute add(result, result, result);
        notify(ping[i+1]);
    enddo
endpar
```

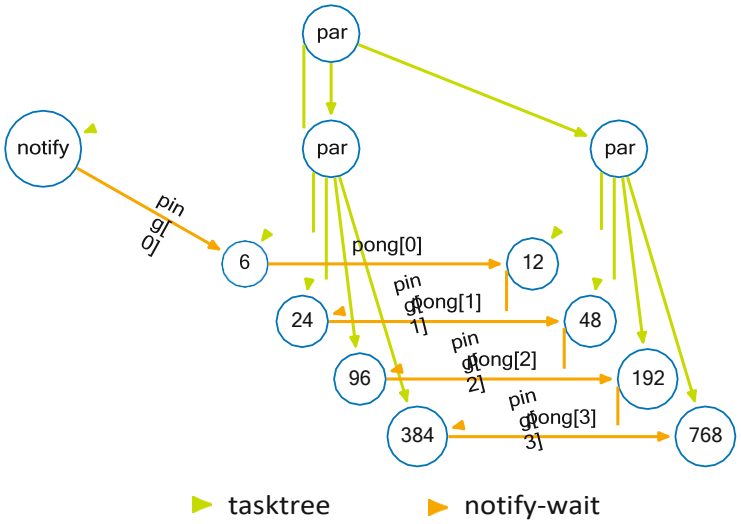

Fig. 3 Graph of tasks as defined by the YvetteML code in Listing 6. The task nodes representing the inner task executing the `compute add` are represented by the result of their computation

one of the `add` functions and the vertices represent the dependencies expressed by the `notify` and `wait` statements. Due to the alternating dependencies between the tasks, this program at the end executes sequentially.

7 Unite and Conquer Approach Using YvetteML

The Unite and Conquer approach was introduced by Emad et al. [6]. The principle of this approach is to make the collaboration of several iterative methods to accelerate the convergence of one of them. This approach can be seen as a model for the design of numerical methods by combining different computational componentsto work for the same objective, with asynchronous communication among them. Unite implies the combination of different computational components, and conquer represents different components work together to solve one problem. Different independent components with asynchronous communications can be deployed on various platforms such as P2P, cloud and supercomputer systems. The idea of mixing asynchronously restarted Krylov methods using distributed and parallel computing was initially introduced by Guy Edjlali and Serge Petiton [4, 5]. They experimented those hybrid Krylov methods asynchronously on networks of heterogeneous parallel computers (e.g., using two Connection Machines, a CM5 and a CM200 and a network of workstations).

Dividing iterative methods into components coupled with asynchronous communication, as suggested in the Unite and Conquer approach, introduces both numerical and parallel benefits for the components.

Numerical benefits: for conventional deflation and polynomial preconditioned methods, the information used is obtained from previous Arnoldi reduction, and it might be difficult to explore larger subspace. Therefore, the convergence might be slowed down. For the methods implemented with the proposed paradigm, the solving and preconditioning parts are independent. This information applied to the deflation or polynomial preconditioned Solver Components can be different from their own Arnoldi reduction, which improves the flexibility of the algorithms,e.g., much more eigenvalues and larger searching space for the deflation. Hence the limitation of spectral information caused by restarting might be broken down, and faster convergence might be obtained. The numerical benefits for linear and eigensolver are already respectively discussed in [7, 14].

Parallel benefits: parallel performance of iterative methods can be improved by the asynchronous promotion and reduction of synchronizations and global communications, especially the synchronization points for the preconditioning. Separating components improves also the fault tolerance and reusability of algorithms.

7.1 UCGLE

UCGLE (Unite and Conquer GMRES/GMRES-LS method) is a linear equation solver implementation based on the Unite and Conquer approach. It composes mainly three computing components: ERAM, GMRES (Generalized Minimal Residual method), and LS (Least-Squares polynomial method). The GMRES component is used to solve the systems, the LS and ERAM components work as the preconditioning part. The asynchronous communication of this hybrid method among three components reduces the number of overall synchronization points and minimizes global communication. The work-flow of UCGLE with three computing components: The ERAM component computes the desired number of dominant eigenvalues, and then sends them to LS component; the LS component uses these received eigenvalues to generate a new residual vector, and sends it to the GMRES component; the GMRES component uses this residual as a new restarted initial vector for solving the non-Hermitian linear systems. Figure 4 shows the better convergence acceleration of UCGLE compared with preconditioned GMRES. The convergence of UCGLE is accelerated by the LS polynomial preconditioning.

For the use-case of multiple right-hand sides, Wu and Petiton extend this method to m-UCGLE [14]. The m-UCGLE approach furthermore splits the problem into blocks, which are solved individually while feeding their results asynchronously into the computation of the other blocks. This loosely synchronized blocking approach is supported by the general asynchronous feedback loop in the UCGLE approach. Overall this method shows better scalability than other approaches while still profiting from the improved convergence behavior of the UCGLE method.

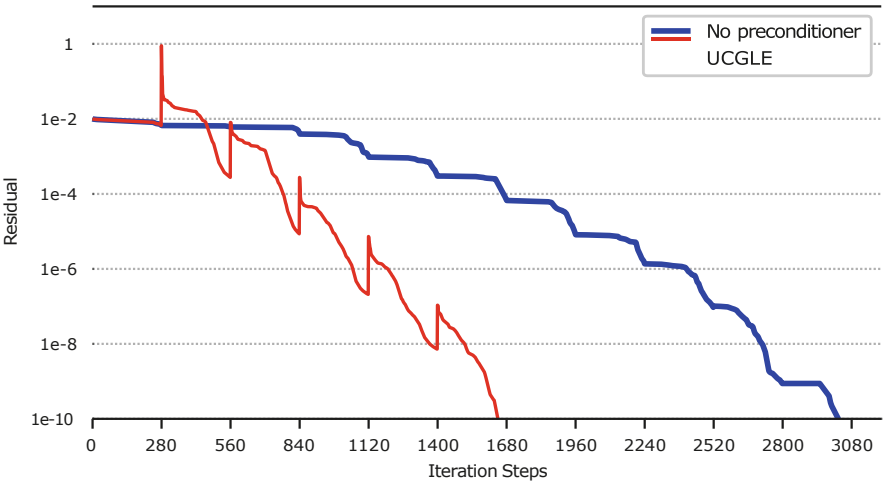

Fig. 4 Convergence comparison of UCGLE method vs. classic GMRES

7.2 Extending YvetteML to Support m-UCGLE

With the current version of YML, the implementation of an m-UCGLE method is not possible due to two limitations: There is no mean for asynchronous communication in YML as needed for the asynchronous feedback loop. YML also provides no way to break early from a YML loop. The latter would be needed to stop iteration at a convergence condition. To make the control flow depending on asynchronous communication, it is necessary to break at multiple levels. Therefore, we propose different kinds of exiting a parallel branch in YML:

1. the application may exit the parallel branch if all the running tasks are completed, e.g., if there are several BGMRES components in parallel to solve linear systems, this parallel section should be exit if all the BGMRES component achieve the convergence;
2. the application may exit the parallel branch if only one task among all is completed, e.g., in the MERAM algorithm, several ERAM components are executed in parallel to approximate the eigenvalues of a matrix, if one of these components approximates enough eigenvalues, the whole parallel section should be exited;
3. the application may exit the parallel branch if only several tasks among all are completed;
4. for the application with multi-level parallelism, we may decide to exit several levels of parallel branches; and
5. the application may exit with saving selected data into the local filesystems, which will improve its fault tolerance and reusability, e.g., lsparams generated by

the B-LSP Component could be saved into local, which will be used for solving the linear systems in future.

By use of the different ways to exit a parallel branch, a unite and conquer algorithm could be implemented with YML. The latter point would even introduce resilience to the YML implementation allowing efficient checkpoint and restart to be defined in the YML description.

8 FP2C

FP2C (Framework for Post-Petascale Computing) is a development and execution environment which supports multi-program methodologies across multiple architectural levels as suggested by Dufaud et al. [3]. FP2C integrates XMP to describe tasks into the workflow environment YML. Therefore FP2C is an implementation of YML to be executed on classical HPC clusters. FP2C is composed of three layers:

1. workflow programming,
2. parallel and distributed programming, and
3. shared-memory parallel programming/accelerator.

The tasks are expected to be executed on sub-clusters or groups of nodes which are tightly connected. These tasks would be hybrid programs with distributed and shared programming models. The workflow scheduler among the sub-clusters or groups invokes and manages the tasks.

The YML backend implementation used for this configuration is OmniRPC-MPI to allow dynamic creation and control of MPI processes needed to executed the YML tasks on an HPC cluster. OmniRPC-MPI is an extension of OmniRPC [12], which supports remote procedure call (RPC) in a grid environment. When the OmniRPC-MPI receives requests to invoke remote programs or to execute taskson the remote programs, then it handles the requests by calling MPI function such as `MPI_Comm_spawn` to create new processes for the task or `MPI_Send` to notify existing, available processes about the new task.

Figure 5 depicts the execution of a workflow with FP2C. Initially, mpirun only starts the process for the YML scheduler. The scheduler loads the task graph and starts executing the YML program by creating and scheduling YML tasks. Using `MPI_Comm_spawn`, the scheduler creates remote programs with the required number of processes to execute a specific task. To avoid the overhead of process startup and shutdown, the scheduler can reuse an existing group of processes to schedule another task, when the previous task is finished like depicted for task2 and task3. By the use of MPI point-to-point communication, the remote program is informed about the next task to execute and also communicates back about the completion of a task. If some YML tasks need a different number of parallel processes than the previously finished task, FP2C will terminate the remote program to spawn new remote programs as depicted for task1, which is replaced by smaller remote programs to execute task5 and task4.

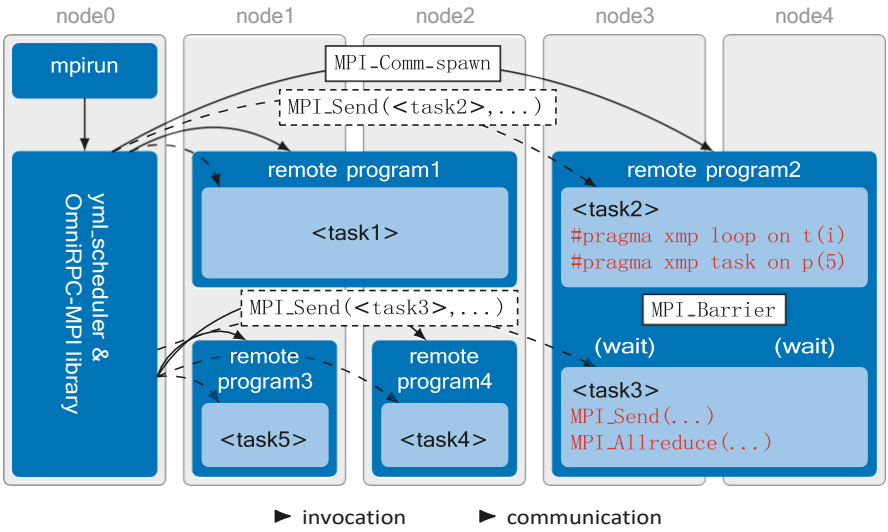

Fig. 5 Execution of an mSPMD application described in the YML control and data flow language in the FP2C implementation

9 Correctness Checking for YML

Correctness checking touches YML at multiple levels:

- The language Yvette, that expresses the graph semantics within YML, is quite similar to the hardware description language Esterel and suffers from similar correctness issues. In this section, we will especially cover potential deadlocks and data races as well as some semantic issues that come with this language.
- The runtime system implementation of YML could also be subject of correctness analysis. The challenge is then to distinguish the behavior of the YML runtime system from application behavior to minimize the analysis overhead.
- Finally, since YML expresses a workflow and runs various modules, it can be of interest to analyze the individual modules separately for correctness.

9.1 Programming Errors in YML Description

With the current specification of YML, the graph defined by a YML graph description can be statically built and therefore also statically analyzed. We identified various possible error patterns in graph descriptions. Possible errors include the use of undefined variables, type miss-match for a variable, but also deadlock due to wait conditions which never receive a signal. Due to the static and self-contained nature of the graph description language, even the possible deadlocks can be identified

statically with data flow analysis. The analysis for those programming errors should be integrated into the YML compiler.

9.2 Challenges of Analyzing the YML Runtime System

Analyzing the runtime system of YML has two mayor challenges for a runtime analysis tool like MUST, which is developed to support the analysis of common HPC applications. The first challenge is to understand the difference between code that represents the YML runtime system and code that belongs to the application code, in this context the YML task code. The bigger challenge is the dynamic MPI characteristic of the YML runtime system, which dynamically creates processes using `MPI_Comm_spawn`, that are then integrated into the execution and should also be supervised by the analysis tool. The analysis tool would also need to understand the resulting new MPI communicators as well as the communication patterns with those spawned processes.

Supporting an application that exposes such dynamic behavior is currently not supported by MUST and the underlying TBON communication layer. The tool would dynamically need to decide about additionally needed analysis processes to extend the TBON. Creating a TBON infrastructure which supports such dynamic application behavior might be subject of a future project.

9.3 Correctness Checking Integrated into FP2C

Since each YML task, invoked by the YML runtime, can be a complete parallel program, such task can have any issue which can also be found in parallel programs. Therefore a developer might want to analyze individual tasks for parallel correctness to identify issues like deadlocks or data race within a task. We introduce a new option for the definition of compute functions into the YML description, which allows applying an analysis tool like MUST to specific YML tasks.

For those selected tasks, the YML scheduler needs to launch additional processes to execute the distributed and centralized analysis of MUST as depicted for remote program2 in Fig. 6. In this specific example, MUST executes both kinds of analysis in a single process. Before launching the FP2C application, the MUST infrastructure needs to be prepared for the execution with each task configuration, which would be done by the `mustrun` execution wrapper for a normal MPI or XMP application. For the execution of a YML task with applied MUST analysis, the remote program controlled by FP2C then needs to select the appropriate prepared configuration of MUST, which is typically done by exporting some environmental variables.

Since we specifically want to analyze the YML task, but not the YML infrastructure, the MPI functions called to implement the FP2C command and control workflow should be ignored by the analysis tool. Some of those MPI functions are

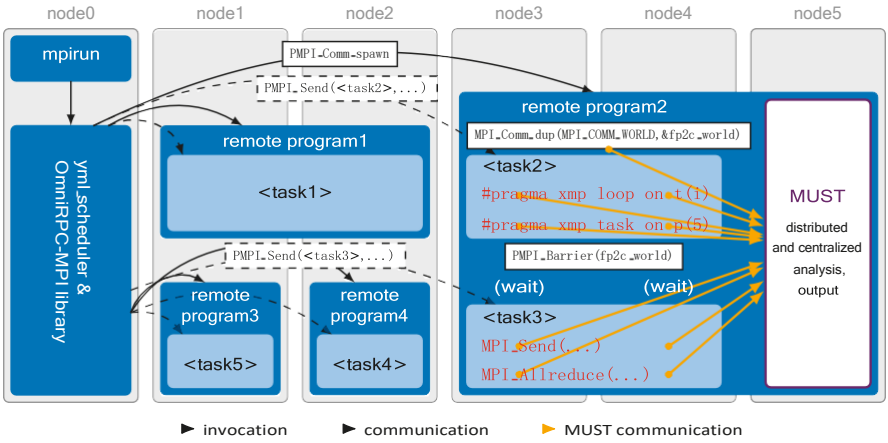

Fig. 6 Integration of MUST into FP2C: the YML scheduler launches selected YML tasks under the control of MUST runtime correctness checking. MUST analyses are applied only to parallel programming constructs used in the task code, but not to MPI communication utilized by the FP2C command and control workflow

used to communicate with the YML scheduler, which is outside of the process controlled by MUST. Such communication to the outside would confuse some analysis performed by MUST. We can avoid analysis of such functions by directly calling into the PMPI interface for functions that implement FP2C functionality. Circumventing MUST analysis for all FP2C-owned MPI communication can result in a deadlock: As Fig. 5 shows, FP2C will execute a barrier at the end of task execution to ensure that all processes finished the execution of the task. For native FP2C execution, it is valid to use `MPI_COMM_WORLD` for this barrier. The MUST analysis process does not know about the barrier and the execution will therefore stall. With FP2C using PMPI calls, MUST will not be able to replace `MPI_COMM_WORLD` by a communicator representing the application processes asit was described in Sect. 2.2. The application processes cannot pass the barrier and the MUST process waits for new messages from the application processes. We could fix this issue by deriving `fp2c_world` from `MPI_COMM_WORLD` using `MPI_Comm_dup` as shown in Fig. 6. The `fp2c_world` communicator can then safely be used by FP2C in PMPI communication calls which are limited to the application processes.

Another challenge when applying MUST to YML tasks is to deal with the output files of MUST. By default, MUST assumes that it is applied to a single MPI application and will write an output file to the current working directory. With FP2C we apply MUST to various YML tasks. To enable the application developer to associate the error report to a specific YML task, we should write the MUST output to a different file per task. The MPI specification defines `int MPI_Pcontrol(const int level, ...)` to allow flexible interaction between MPI application and PMPI tool. It is the responsibility

of the tool to interpret and define the arguments passed to this variadic function. For our use case, we defined usage with `level=8192` and signature `int MPI_Pcontrol(const int level, const char* filename)` to indicate that the analysis results of subsequent application events should be written to the new file name. We also want to make sure that distributed analysis for application events before the pcontrol call write the report into the old file. Therefore we require this pcontrol function call to be collective on the whole application. Currently, we do not require to finish all MPI communication at this point. In the future, we might add some additional pcontrol commands to express certain runtime assertions. Such assertions might include that no outstanding messages are expected or all MPI handles should be released at a certain point.

10 Conclusion

In this paper, we discussed how we can apply runtime correctness checking to emerging multi-level parallel programming languages which try to encounter the challenges of multi-level concurrency of exascale systems. Specifically, we looked into possible correctness issues in XMP applications, which represent the field of PGAS languages. We described how we integrated runtime correctness analysis for XMP applications into the runtime correctness checking tool MUST and therefore specified the new tools interface XMPT for XcalableMP. The workflow description language YML allows to introduce another level of high-level concurrency and therefore better exploit the massive available concurrency of exascale systems. As an example application for such a high-level concurrency workflow, we introduced the unite and conquer method UCGLE. This method improves the convergence behavior of certain solvers of linear equation systems by asynchronously exchanging intermediate results of preconditioner and solver. We introduced FP2C as an implementation of YML targeting HPC systems. We showed how we could integrate MUST runtime analysis to be applied to certain tasks scheduled by the FP2C runtime system and discussed solutions for challenges on the way to a successful workflow.

Acknowledgments The research reported here received funding by the German Research Foundation (DFG) through the priority program 1648 SPPEXA, by the Agence nationale de la recherche (ANR) and by the Japan Science and Technology Agency (JST). The authors would like to thank them for making this research possible.

References

1. Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Ahn, D.H., Laguna, I., Schulz, M., Lee, G.L., Protze, J., Müller, M.S.: ARCHER: Effectively spotting data races in large openmp applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, May 23–27, 2016, pp. 53–62 (2016)
2. Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.C., Barkai, D., Berthou, J.Y., Boku, T., Braunschweig, B., Cappello, F., Chapman, B., Chi, X., Choudhary, A., Dosanjh, S., Dunning, T., Fiore, S., Geist, A., Gropp, B., Harrison, R., Hereld, M., Heroux, M., Hoisie, A., Hotta, K., Jin, Z., Ishikawa, Y., Johnson, F., Kale, S., Kenway, R., Keyes, D., Kramer, B., Labarta, J., Lichnewsky, A., Lippert, T., Lucas, B., Maccabe, B., Matsuoka, S., Messina, P., Michielse, P., Mohr, B., Mueller, M.S., Nagel, W.E., Nakashima, H., Papka, M.E., Reed, D., Sato, M., Seidel, E., Shalf, J., Skinner, D., Snir, M., Sterling, T., Stevens, R., Streitz, F., Sugar, B., Sumimoto, S., Tang, W., Taylor, J., Thakur, R., Trefethen, A., Valero, M., Van Der Steen, A., Vetter, J., Williams, P., Wisniewski, R., Yelick, K.: The international exascale software project roadmap. Int. J. High Perform. Comput. Appl. **25**(1), 3–60 (2011). https://doi.org/10.1177/1094342010391989
3. Dufaud, T., Tsuji, M., Sato, M.: Design of data management for multi SPMD workflow programming model. In: Proceedings of the 4th International Workshop on Extreme Scale Programming Models and Middleware, ESPM2@SC 2018, Dallas, November 11–16, 2018, pp. 9–18 (2018)
4. Edjlali, G., Emad, N., Petiton, S.: Hybrid methods on network of heterogeneous parallel computers. In: Proceedings of the 14th IMACS World Congress, Atlanta (1994)
5. Edjlali, G., Petiton, S., Emad, N.: Interleaved parallel hybrid Arnoldi method for a parallel machine and a network of workstations. In: Conference on Information, Systems, Analysis and Synthesis (ISAS'96), pp. 22–26 (1996)
6. Emad, N., Petiton, S.G.: Unite and conquer approach for high scale numerical computing. J. Comput. Sci. **14**, 5–14 (2016). https://doi.org/10.1016/j.jocs.2016.01.007. https://doi.org/10.1016/j.jocs.2016.01.007
7. Emad, N., Petiton, S., Edjlali, G.: Multiple explicitly restarted arnoldi method for solving large eigenproblems. SIAM J. Sci. Comput. **27**(1), 253–277 (2005)
8. Hilbrich, T., de Supinski, B.R., Nagel, W.E., Protze, J., Baier, C., Müller, M.S.: Distributed wait state tracking for runtime MPI deadlock detection. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, November 17–21, 2013, pp. 16:1–16:12 (2013). https://doi.org/10.1145/2503210.2503237. https://doi.org/10.1145/2503210.2503237
9. Hück, A., Lehr, J., Kreutzer, S., Protze, J., Terboven, C., Bischof, C.H., Müller, M.S.: Compiler-aided type tracking for correctness checking of MPI applications. In: 2nd IEEE/ACM International Workshop on Software Correctness for HPC Applications, CORRECTNESS@SC 2018, Dallas, November 12, 2018, pp. 51–58 (2018). https://doi.org/10.1109/Correctness.2018.00011. https://doi.org/10.1109/Correctness.2018.00011
10. Nakao, M., Lee, J., Boku, T., Sato, M.: Productivity and performance of global-view programming with xcalablemp pgas language. In: Proceedings of 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid, pp. 402–409 (2012). https://doi.org/10.1109/CCGrid.2012.118
11. Protze, J., Schulz, M., Ahn, D.H., Müller, M.S.: Thread-local concurrency: a technique to handle data race detection at programming model abstraction. In: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2018, Tempe, June 11–15, 2018, pp. 144–155 (2018)
12. Sato, M., Hirano, M., Tanaka, Y., Sekiguchi, S.: Omnirpc: A grid RPC facility for cluster and global computing in openmp. In: OpenMP Shared Memory Parallel Programming, International Workshop on OpenMP Applications and Tools, WOMPAT 2001, West Lafayette, July 30–31, 2001 Proceedings, pp. 130–136 (2001)

13. Tsuji, M., Sato, M., Hugues, M., Petiton, S.: Multiple-SPMD programming environment basedon pgas and workflow toward post-petascale computing. In: Proceedings of 42nd InternationalConference on Parallel Processing, ICPP, pp. 480–485. IEEE, Piscataway (2013). https://doi. org/10.1109/ICPP.2013.58
14. Wu, X., Petiton, S.G.: A distributed and parallel asynchronous unite and conquer method to solve large scale non-hermitian linear systems. In: Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPC Asia 2018, Chiyoda, January 28–31, 2018, pp. 36–46. ACM, New York (2018). https://doi.org/10.1145/3149457.3154481