Building Correctness Analysis on Top of XMPT

Joachim Protze (protze@itc.rwth-aachen.de)

3rd Workshop on Parallel Programming Models - Productivity and Applications

High Performance Computing
RWTH AACHEN UNIVERSITY

Motivation

- Increasing concurrency in HPC needs new concepts of programming
- MPI + X is one candidate
  - Highlights the need for multilevel parallelism
  - Potentially even more levels of parallelism
- Multiple PGAS approaches were developed
  - Did any of these gain acceptance?
  - Why?

Key to success might be tool support

Portable runtime correctness checking

- Can we reuse existing tools and apply them to new programming paradigms?
- What are the common challenges?
- What are the limitations?
- How big is the effort to integrate analysis for pragma-based PGAS approach XMP into existing MPI tool MUST?
- Data race and deadlock are major threats in parallel
  - Can we define an abstract interface, that provides sufficient information to analyse arbitrary parallel paradigms?

Agenda

- What is XMPT? And why do we want it?
- Experiences using XMPT
  - Lessons to learn for tool interfaces
- Results of applying MUST to XMP applications

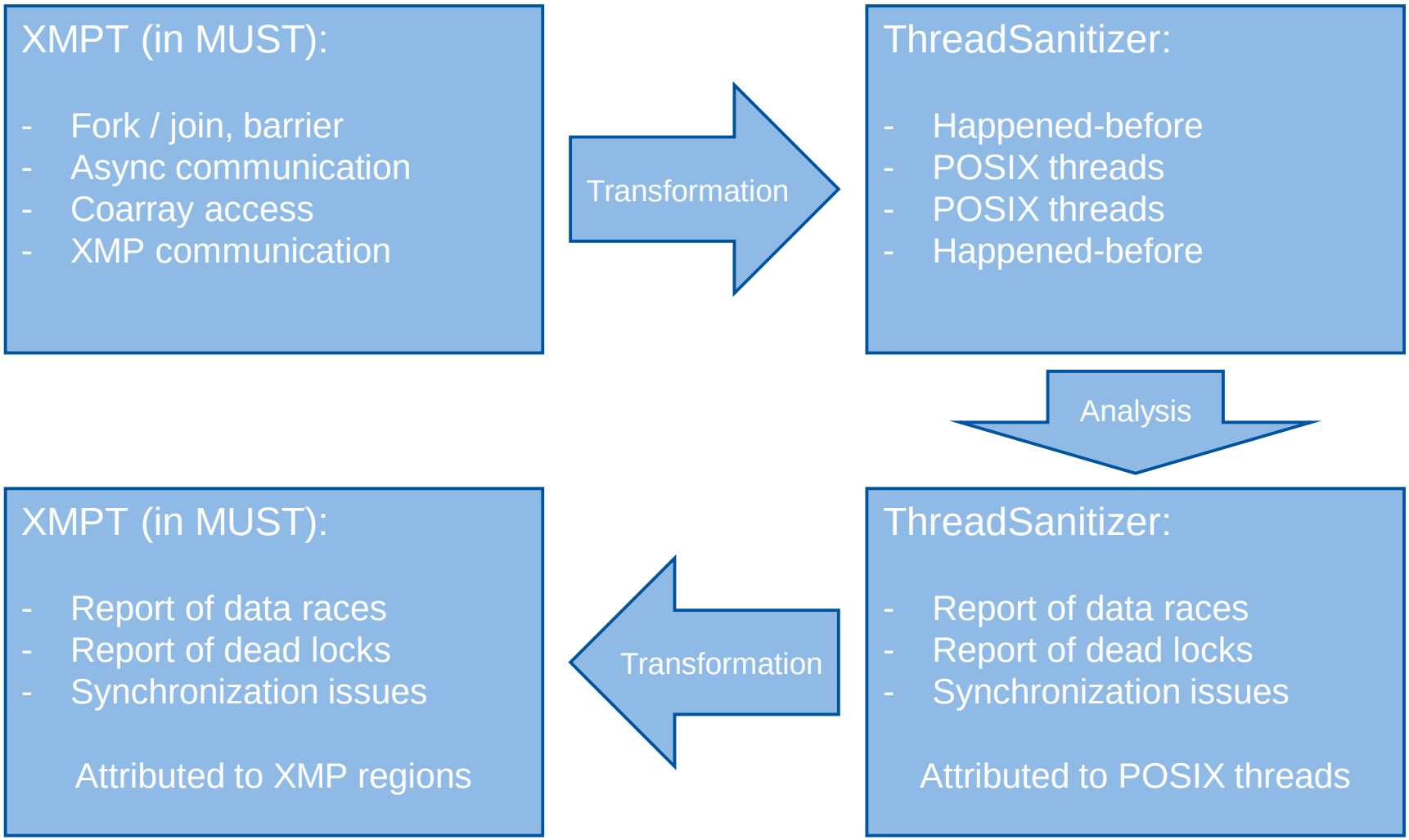
Tools interfaces

- PMPI
  - Tools interface for MPI
  - MPI spec describes wrapping of MPI functions
- OMPT
  - Tools interface for OpenMP
  - Latest OpenMP spec describes events, tool gets notification about encountered events
- XMPT
  - Tools interface for XMP, follows the specification of OMPT
  - Development of XMPT started with OMPT @TR2 level

XMPT

- What events are needed?
  - Events for the begin and end of XMP regions
- What information is needed?
  - Essentially, all information possibly provided to the XMP pragma
  - To allow stateless implementation of the tool, the runtime stores a tool data with scopes
  - XMP already provided functions to derive information from handles
- Is this information sufficient for performance analysis?

Example: Data race detection for XMP



Experiences using XMPT

Interface between XMP runtime and tool

… or what can go wrong with visibility of symbols?

- Main difference between XMP runtime library and most OpenMP runtime libraries:
  - XMP runtime library is statically linked into the application

- For OMPT the tool startup has significantly evolved since TR2:
  - `ompt_start_tool(`*`lookup-function, version`*`)` is the single public interface symbol
  - OpenMP runtime tries to find this function in
    - The application
    - Any loaded dynamic library
    - Any library listed in `OMP_TOOL_LIBRARIES`
  - The tool uses the *lookup-function* to find all other OMPT functions

Interface between XMP runtime and tool

… and what is the actual problem?

- XMPT relies on many xmp-functions
  - These function are not visible for a dynamicly loaded tool
    - Compile application with `-rdynamic` or
    - Provide a lookup-function or
    - Ensure to add all necessary function to the dynamic symbol table of the application

Information on descriptors

- XMP provides functions to query all kinds of attributes from descriptors
- A tool can store this information in an object and bind the object to the descriptor.
- But there is no way to detect the end of lifetime for descriptors

Source code location

- For debugging the source code location is important
- OMPT added codeptr_ra to events, this provides informations where the pragma is placed

MUST implementation

How can we do analysis based on XMPT events?

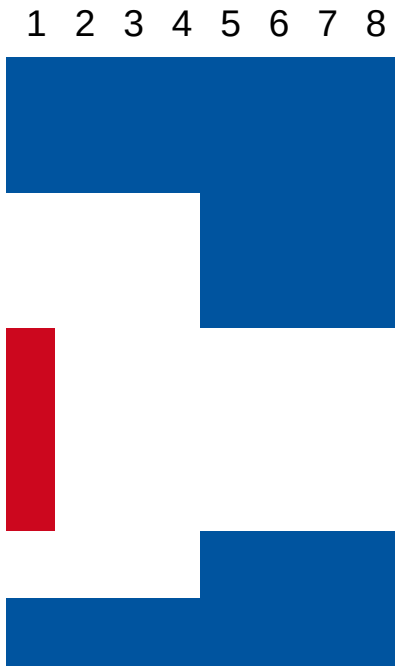
```
#pragma xmp nodes p(8)
int main(int argc, char** argv){
  xmp_init(&argc, &argv);
#pragma xmp task on p(5:8)
  {
#pragma xmp task on p(1)
    {
      printf("PASS\n");
    }
  }
  return 0;
}
```

xmp-task-begin (desc=p, subscr=(5:8), scope-data=A)

xmp-task-begin (desc=p, subscr=(1:1), scope-data=B)

xmp-task-end (scope-data=B)

xmp-task-end (scope-data=A)

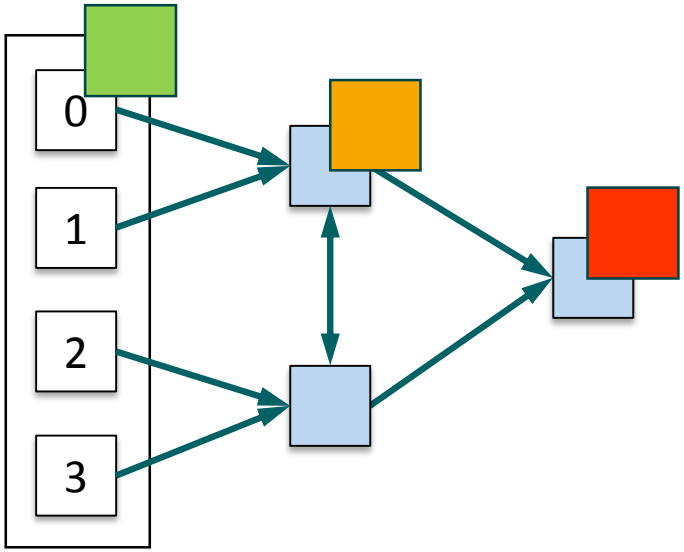


- There is no event for the nodes directive.
- The first time p is seen, MUST queries all available information on p
- For task-begin
  - We analyse whether (desc/subsc) is in the executing node-set
  - We update the executing node-set to be (desc/subsc)

Storing information next to the descriptors

Good idea, but unfortunatelly not applicable for MUST

- We need the information not only for local analysis, but also on other nodes for distributed analysis
- When MUST finds an unknown descriptor, we create a GTI event that is propageted in the tree

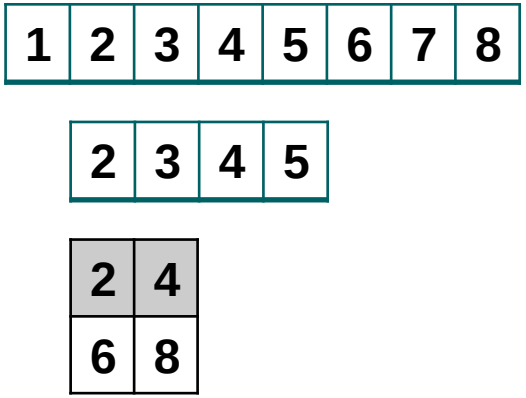


Tracking executing nodeset

… can be quite difficult in general case

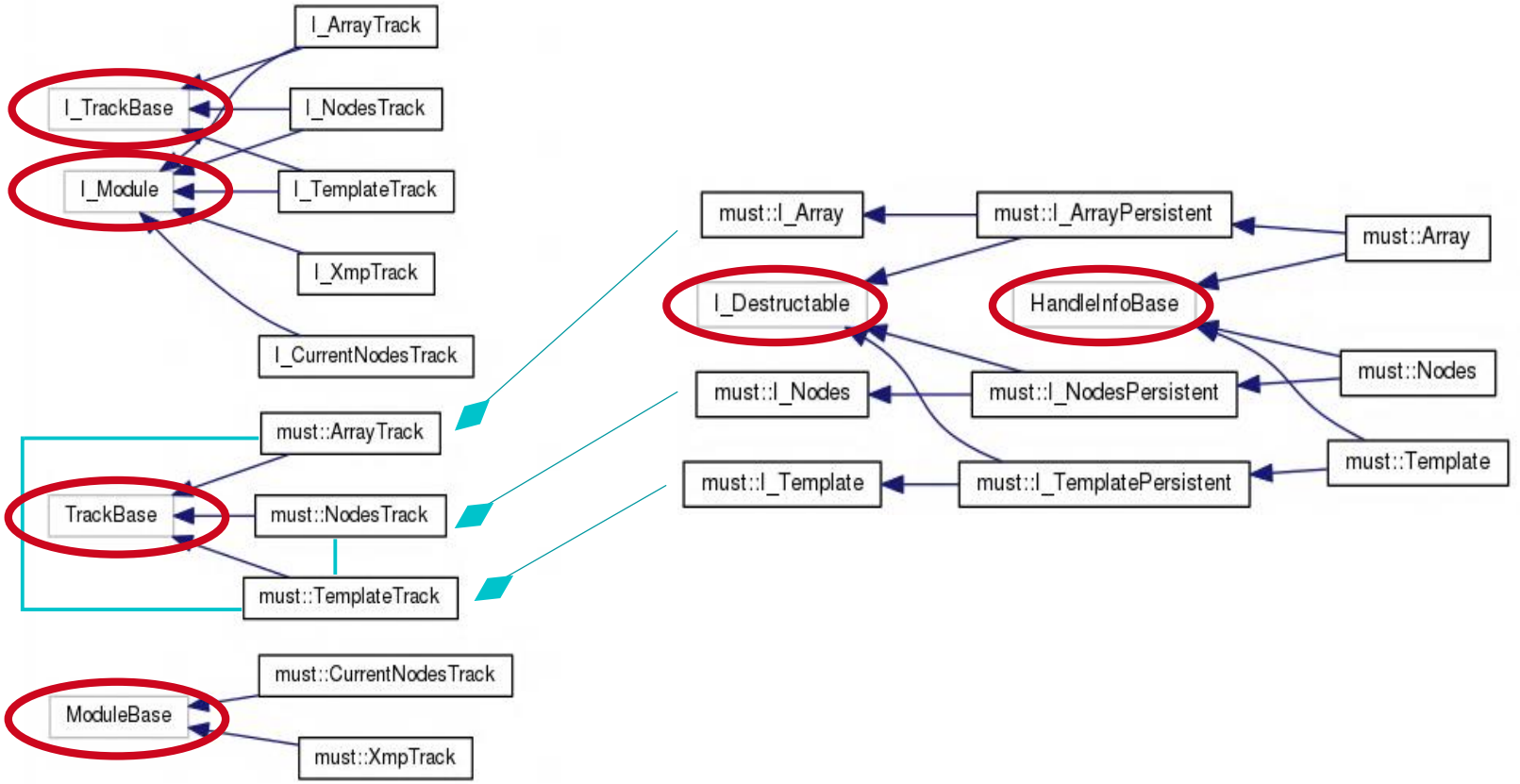
```
#pragma xmp nodes p(8)
#pragma xmp nodes q(4)=p(2:5)
#pragma xmp nodes r(2,2)=p(2:8:2)

#pragma xmp task on r(2,:)
#pragma xmp task on p(1)
printf("PASS\n");
```

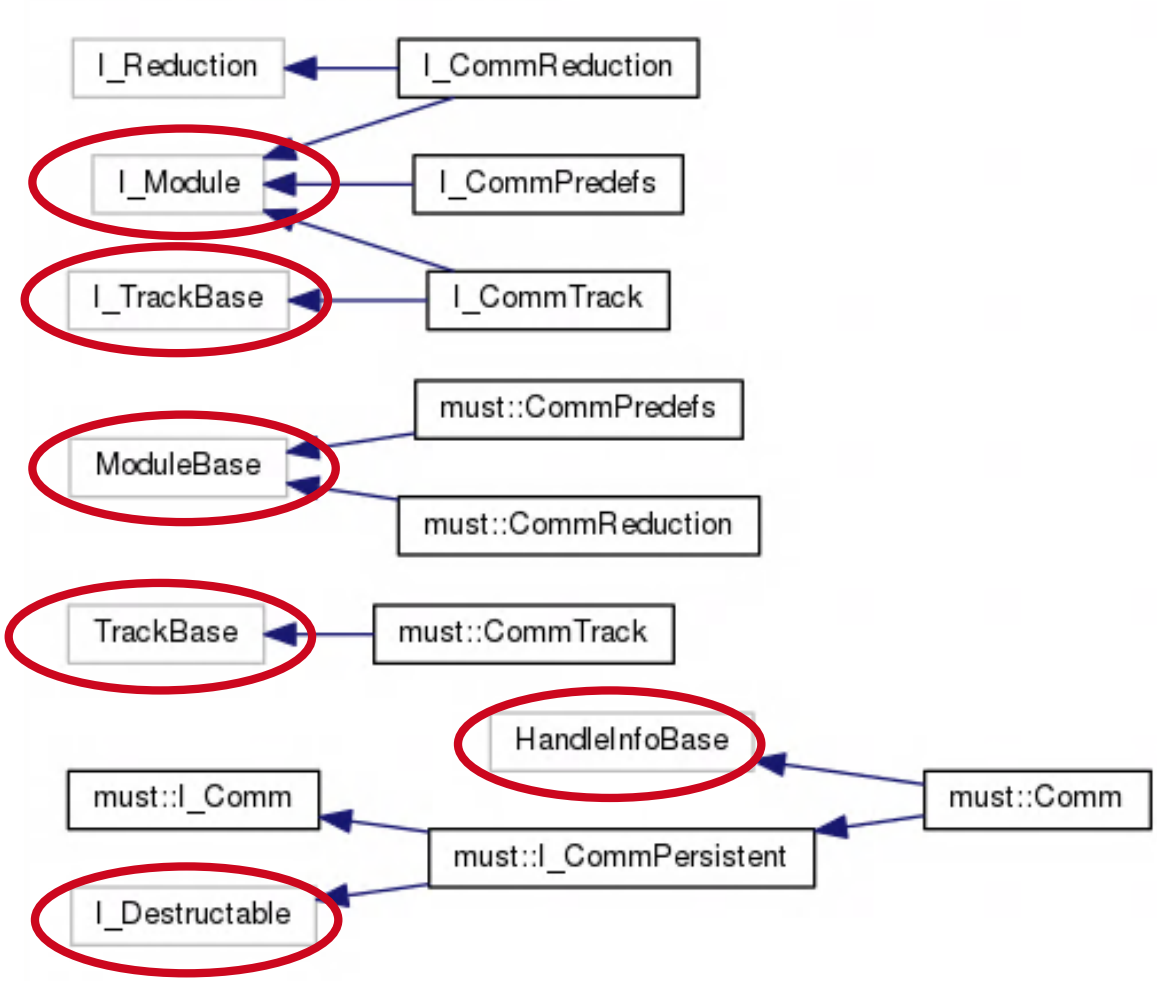


- **Is p(1) in r(2,:)?**
- For the general case we recursively create a bitfield which marks active nodes

Tracking information on descriptors



Similar as tracking information on MPI handles



XMPT based analyses in MUST

- We currently have most analyses implemented, which are possible with local knowledge
- Still working on analysis of asynchronous communication

- Next is collective consistency (all do „the same thing“)

- Finally integrate Simon’s work for XMP coarray

Summary

- Correctness tools are important
- Tools are always steps behind the development of new languages
- Tools interface is important for easier porting of tools

- Sometimes we here the question:
  - Why not use a language, that prevents the issues?
  - How many HPC codes are written in RUST?

Thank you for your attention.