Content

MUST
- MPI usage errors
- MUST reports
- MUST usage
- MUST features
- MUST future

Archer
- OpenMP data race detection
- Archer usage
- Archer GUI

% git clone https://git.rwth-aachen.de/protze/tools-tutorial.git

Slides/Handson
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    int rank, size, buf[8];
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INTEGER, &type);
    MPI_Recv (buf, 2, MPI_INT, size - rank, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send (buf, 2, type, size - rank, 123, MPI_COMM_WORLD);
    printf("Hello, I am rank %d of %d\n", rank, size);
    return 0;
}

No MPI_Init before first MPI-call
Fortran type in C
Recv-recv deadlock
Rank0: src=size (out of range)
Type not committed before use
Type not freed before end of main
Send 4 int, recv 2 int: truncation
No MPI_Finalize before end of main

% git clone
https://git.rwth-aachen.de/protze/tools-tutorial.git

Slides/Handson
MPI usage errors

• MPI programming is error prone
• Bugs may manifest as:
  – Crashes
  – Hangs
  – Wrong results
  – Not at all! (Sleeping bugs)
• Tools help to detect these issues
MPI usage errors

- Complications in MPI usage:
  - Non-blocking communication
  - Persistent communication
  - Complex collectives (e.g. Alltoallw)
  - Derived datatypes
  - Non-contiguous buffers

- Error Classes include:
  - Incorrect arguments
  - Resource errors
  - Buffer usage
  - Type matching
  - Deadlocks
MUST: Tool design

- Application
- Additional tool ranks
- MPI Library
- Split-comm-world
- Force status wrapper
- GTI: event forwarding, network
- Local Analyses
- Non-Local Analyses
- Rewrite-comm-world
- MPI Library
MUST: Tool design

GTI: event forwarding, network

Local Analyses

Non-Local Analyses

0 → 1 → 4 → 6

1 → 4

2 → 3 → 5
MUST: Hands-on / Demo

- Load MUST

- Download the correctness examples:

  ```
  % git clone https://git.rwth-aachen.de/protze/tools-tutorial.git
  ```

- Compile and execute the MPI example:

  ```
  % mpicc -g must-example.c -o example.exe
  % salloc -ppdebug
  % mustrun --must:mpiexec srun -n4 -ppdebug ./example.exe
  ```
Must detects deadlocks

Who?

What?

Where?

Details

The application issued a set of MPI calls that can cause a deadlock! A graphical representation of this situation is available in a detailed deadlock view.

Click for graphical representation of the detected deadlock situation.
Graphical representation of deadlocks

Rank 0 waits for rank 1 and vv.

Simple call stack for this example.
Sometimes fixing one defect introduces several new ones

```c
#include <mpi.h>
#include <stdio.h>

int main (int argc, char** argv)
{
    int rank, size, buf[8];

    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);

    MPI_Datatype type;
    MPI_Type_contiguous (2, MPI_INT, &type);
    MPI_Type_commit (&type);

    MPI_Request request;
    MPI_Irecv (buf, 2, MPI_INT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

    MPI_Send (buf, 1, type, size - rank - 1, 123, MPI_COMM_WORLD);

    printf ("Hello, I am rank %d of %d.\n", rank, size);

    MPI_Finalize ();

    return 0;
}
```

Deadlock was fixed by non-blocking recv
MUST detects data races in asynchronous communication.

Data race between send and asynchronous receive operation.

- Missing MPI_Wait is diagnosed as resource leak.
Graphical representation of the race condition

Message
Overlap in communication buffers! The graph below shows details on this situation. The first colliding item is highlighted.

Datatype Graph

- MPI_Send: send(buf = 0x7fff43c973a0)
- MPI_Type_contiguous(count=2)
- MPI_Irecv: recv(buf = +0x0)
- [0]
- MPI_INT
MUST found no issue

No further error detected

Hopefully this message applies to many applications
MUST – Basic Usage

• Load MUST module/dotkit, e.g.:

% module load must

• Apply MUST with an mpiexec wrapper, that’s it:

• Instead of

% $MPICC source.c -o exe
% $MPIRUN -np 4 ./exe

• Replace:

% $MPICC -g source.c -o exe
% mustrun --must:mpiexec $MPIRUN -np 4 ./exe

or:

% mustrun -np 4 ./exe

• After run: inspect “MUST_Output.html”
Remember the analysis tree?

• By default MUST uses a single extra node
• In batch: Allocate the extra process(es)!

• Query information about required processes:

  % mustrun --must:info -np 4 ./exe

• For distributed analysis
  – Either:

    % mustrun --must:distributed -np 4 ./exe

  – Or:

    % mustrun --must:fanin 16 -np 4 ./exe

  – The latter allows you to specify a branching factor for the tree
Application crash handling

- Fatal error might stop the execution before report is written 😞

- Default: crash-safe centralized analysis
  - MPI call is only executed, when tool process finished analysis
  - Serializes MPI communication (overhead might be significant)
  - Assert that application does not crash (allows asynchronous analysis):

  ```
  % mustrun --must:nocrash -np 4 ./exe
  ```

- Distributed: can handle crashes, but MPI might be dead
  - Use alternative communication layer for MUST:

  ```
  % mustrun --must:nodesize 8 -np 4 ./exe
  ```
  
  - nodesize must be divider of processes scheduled per node
  - One process per nodesize becomes tool process
  - Might have some minor influence to the communication behavior
Analysis of multi-threaded MPI applications

- Default: MUST limits to MPI\_THREAD\_FUNNELED
  - Only the master thread can call MPI
  - The application must respect this thread-level

- Hybrid analysis: MUST requires MPI\_THREAD\_MULTIPLE
  - The application can use any MPI thread-level
  - MUST raises the requested thread-level to multiple

\begin{verbatim}
% mustrun --must:hybrid -np 4 ./exe
\end{verbatim}

- MUST adds an analysis thread to each rank
  - Potentially oversubscribes the node? Should not matter for most apps.
- The additional tool processes are single-threaded
  - Does not fill those nodes, intelligent batch system might help in future?
Mustrun modes: Separate prepare and run

- Prepare MUST for execution with specific config (on frontend):
  
  ```
  % mustrun --must:mode prepare -np 4 ./exe
  ```

- Execute with MUST in a batch job (after prepare):
  
  ```
  % mustrun --must:mode run -np 4 ./exe
  ```

- Enforce new tool configuration/building and start execution:
  
  ```
  % mustrun --must:mode preparerun -np 4 ./exe
  ```
Upcoming features based on LLVM/clang

• Data type checking
  ```c
  int array[10];
  MPI_Send(array, MPI_FLOAT, 10, ...);
  ```
  – Needs compile time information
  – Prototype implemented, needs to be integrated into release

• Data race detection
  ```c
  MPI_Isend(array, MPI_FLOAT, 10, ..., &req);
  array[5]++;
  MPI_Wait(&req,...);
  ```
  – Integration of MUST and ThreadSanitizer/Archer
Installing MUST (⇒ Sysadmins)

- Configure with Cmake
  - Activate stack trace, if Dyninst is installed:
    - -DUSE_CALLPATH=on
    - -DSTACKWALKER_INSTALL_PREFIX=<dyninst-install-path>
  - Set a default mpiexec command:
    - -DMPIEXEC=srun

- Make / Install:
  - make -j16 install install-prebuilds

- Prebuilds are preconfigured tool configurations, that are installed with the tool
  - Runs --must:mode prepare for common tool configurations
  - Tool configuration not covered by the Prebuilds will trigger some tool configuration / compilation during “mustrun”
Archer: OpenMP data race detection
Data race detection tool: Archer

- Error checking tool for
  - Memory errors
  - **Threading errors** (OpenMP, Pthreads)
- Based on ThreadSanitizer (runtime check)
- Available for Linux, Windows and Mac
- Supports C, C++ (Fortran in work)
- Synchronization information based on OMPT
- More info: [https://github.com/PRUNERS/archer](https://github.com/PRUNERS/archer)
- Will hopefully be part of the 9.0 release of LLVM
  - Most probably missed the deadline 😞
Archer - Usage

- Compile the program with `-g` and `-fsanitize=thread` flag
  - `clang -g -fsanitize=thread -fopenmp myprog.c -o myprog`

- Run the program under control of ARCHER Runtime
  - `export OMP_NUM_THREADS=...`  
    - `./myprog`
  - Detects problems only in software branches that are executed

- Understand and correct the threading errors detected

- Edit the source code
- Repeat until no errors reported
Archer - Result Summary

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int a = 0;
    #pragma omp parallel
    {
        if (a < 100) {
            #pragma omp critical
            a++;
        }
    }
}
```

WARNING: ThreadSanitizer: data race
Read of size 4 at 0x7fffffffdcdc by thread T2:
- #0 .omp_outlined. race.c:7 (race+0x000000004a6dce)
  #1 __kmp_invoke_microtask <null> (libomp.so)

Previous write of size 4 at 0x7fffffffdcdc by main thread:
- #0 .omp_outlined. race.c:9 (race+0x000000004a6e2c)
  #1 __kmp_invoke_microtask <null> (libomp.so)
Hands-on / Demo

• Load Archer module

$ cd ~/tools-tutorial/Debug-examples
$ clang -fopenmp -g prime_omp.c -lm

Try:

$ OMP_NUM_THREADS=2 ./a.out
$ OMP_NUM_THREADS=4 ./a.out
$ OMP_NUM_THREADS=8 ./a.out
Hands-on / Demo

- Compile with data race detection:

  $ clang -fsanitize=thread -fopenmp -g prime_omp.c -lm

- Make Archer library available (could be done by module):

  $ export OMP_TOOL_LIBRARIES=libarcher.so

- Execute with some threads:

  $ OMP_NUM_THREADS=2 ./a.out

Fix the issues, recompile, test again
Hands-on / Demo

• To verify that ARCHER is active, you can make ARCHER verbose:

$ ARCHER_OPTIONS="verbose=1" OMP_NUM_THREADS=2 ./a.out
Archer detected OpenMP application with TSan, supplying OpenMP synchronization semantics
Usage for Fortran-code

- No Fortran compiler frontend with ThreadSanitizer in LLVM
- But we can use gfortran for compilation:

\[
gfortran -fsanitize=thread -fopenmp -g -c app.f
\]

- Still use clang for linking:

\[
clang -fsanitize=thread -fopenmp -lgfortran app.o
\]

\[
OMP_NUM_THREADS=2 ./a.out
\]

For OpenMP programs, always use the clang delivered with ARCHER to avoid false alerts
Advanced: use annotations for custom synchronization

OMP2012/371.applu331/src/synchs.f90:

```fortran
subroutine sync_left( ldmx, ldmy, ldmz, v )
...
if (iam .gt. 0 .and. iam .le. mthreadnum) then
  neigh = iam - 1
  do while (isync(omp_get_thread_num() - 1) .eq. 0)
    !$omp flush(isync)
    end do
    CALL AnnotateHappensAfter(__FILE__, __LINE__,
                               isync(omp_get_thread_num() - 1))
    CALL AnnotateHappensBefore(__FILE__, __LINE__,
                               isync(neigh))
    isync(neigh) = 0
    !$omp flush(isync,v)
  endif
```
Upcoming: Archer GUI

- Implemented by LLNL summer student: Sam Thayer
- Archer report is redirected into json output
- Aggregated report is presented in GUI

$ archer-gui <directory with Archer report files>

- Aggregates across threads and (MPI) processes
Upcoming: Archer GUI

- Implemented by LLNL summer student: Sam Thayer
- Archer report is redirected to JSON output
- Aggregates reports into user-defined categories (across threads and MPI processes)
- Displays reports in ToolGear UI (implemented by LLNL employee John Gyllenhaal)
  - The same UI as other debugging tools such as MemCheckView
Upcoming: Archer GUI

- Offers simple options sufficient for most use cases
- Alternately, offers detailed control of report aggregation
Thank you for your attention.