OpenMP data race detection with Archer

46th VI-HPS Workshop
Joachim Jenke, Felix Tomski

i12 High Performance Computing
RWTH AACHEN UNIVERSITY

Data Race

C++

- The execution of a program contains a **data race** if it contains **two** potentially **concurrent** conflicting actions, at least one of which is **not atomic**, and **neither happens before** the other, [...]. Any such data race results in **undefined behavior**.

OpenMP

- **Multiple threads** access the same memory **unordered**, at least one thread writes. If a data race occurs then the result of the program is unspecified.

- There is **no benign** data race in C/C++. It is always UB!

Data Race vs. Race Condition

- Data race is UB, race condition is not.
- Sprinkling atomics into the code avoids UB, but not necessarily race conditions

```
#include <stdio.h>
#include <math.h>

int main(int argc, char** argv){
  int a = 0, l = 100; double sum = 0;
  if (argc > 1)
    l = atoi(argv[1]);
#pragma omp parallel num_threads(8) reduction(+:sum)
  while (a < l)
    sum += sin(a++);
  printf("a=%i, sum=%lf\n", a, sum);
}
```

```
#include <stdio.h>
#include <math.h>
#include <atomic>
int main(int argc, char** argv){
  std::atomic<int> a{0}; int l = 100; double sum = 0;
  if (argc > 1)
    l = atoi(argv[1]);
#pragma omp parallel num_threads(8) reduction(+:sum)
  while (a < l)
    sum += sin(a++);
  printf("a=%i, sum=%lf\n", a.load(), sum);
}
```

Detect data races in OpenMP target code

- Add thread sanitizer flag for compilation:
  - `clang++ -fopenmp -fno-omit-frame-pointer -fsanitize=thread -g`
  - Also works with latest Intel compilers (icx/icpx/ifx)

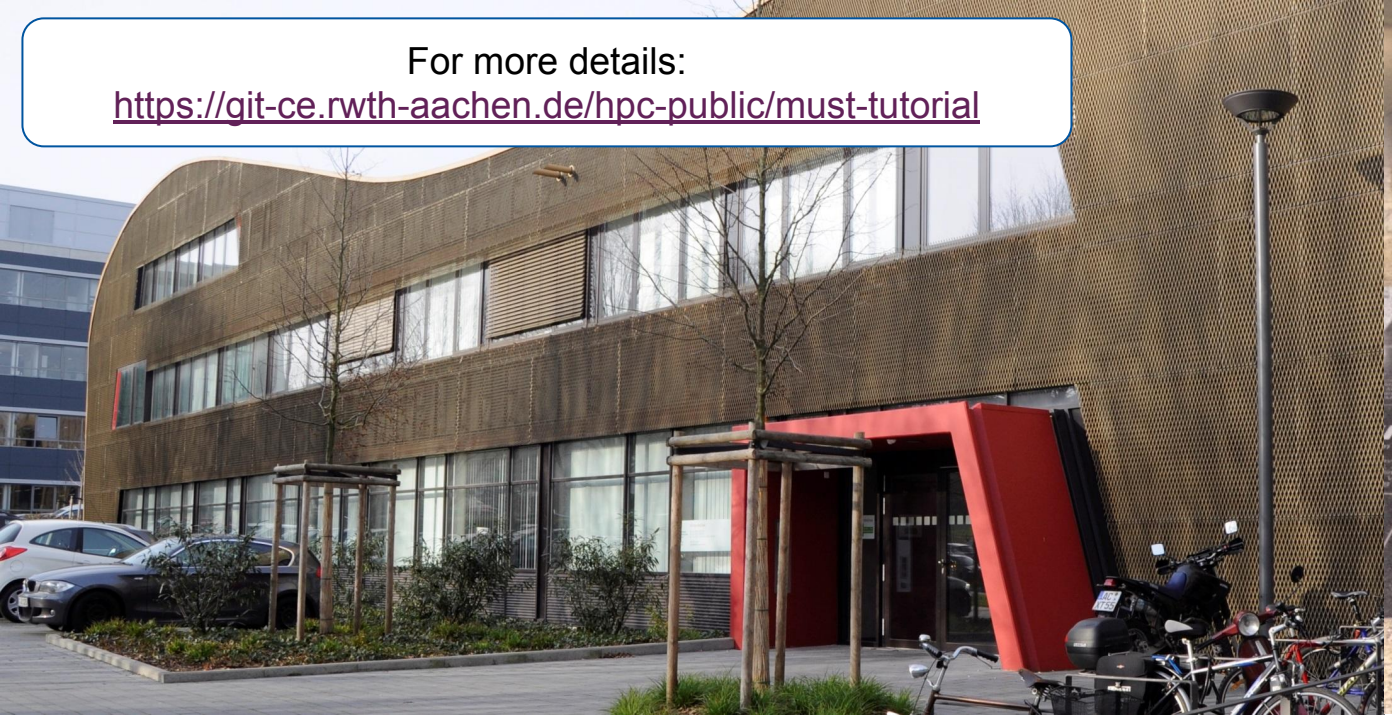
- Clang allows offloading to host:
  - `clang++ -fopenmp -fopenmp-targets=x86_64-pc-linux-gnu`
- add TSan instrumentation:
  - `clang++ -fopenmp -fopenmp-targets=x86_64-pc-linux-gnu -fsanitize=thread -g`

- Available with all llvm-based compilers (AMD, Intel, …)
- For Intel compiler make sure to explicitly load Archer:
  - `OMP_TOOL_LIBRARIES=libarcher.so`

Using Sanitizers With Fortran Code

- GCC supports most sanitizers (other than MSan)
- Compile Fortran codes with `gfortran` or `ifx` + sanitizer flags
- Support in `flang` is possible, but not yet integrated
- For Archer support, make sure to link the LLVM OpenMP runtime, not GNU runtime:
  - `$ gfortran -lomp -fopenmp -fsanitize=thread -fno-omit-frame-pointer test.f90`
  - `$ gfortran -fopenmp -c -fsanitize=thread -fno-omit-frame-pointer test.f90`
    `$ clang -fopenmp -fsanitize=thread --gcc-install-dir=<path-to-gfortran>`
    `-lgfortran test.o`
  - New: use `ifx`
  - The latter links the LLVM TSan runtime which has significantly lower runtime overhead

For more details:
https://git-ce.rwth-aachen.de/hpc-public/must-tutorial



MPI + GPU Correctness Checking with MUST

46th VI-HPS Workshop
Joachim Jenke, Felix Tomski, Alexander Hück, Simon Schwitanski

High Performance Computing
RWTH AACHEN UNIVERSITY

How many issues can you spot in this tiny example?

At least 8 issues in this code example!

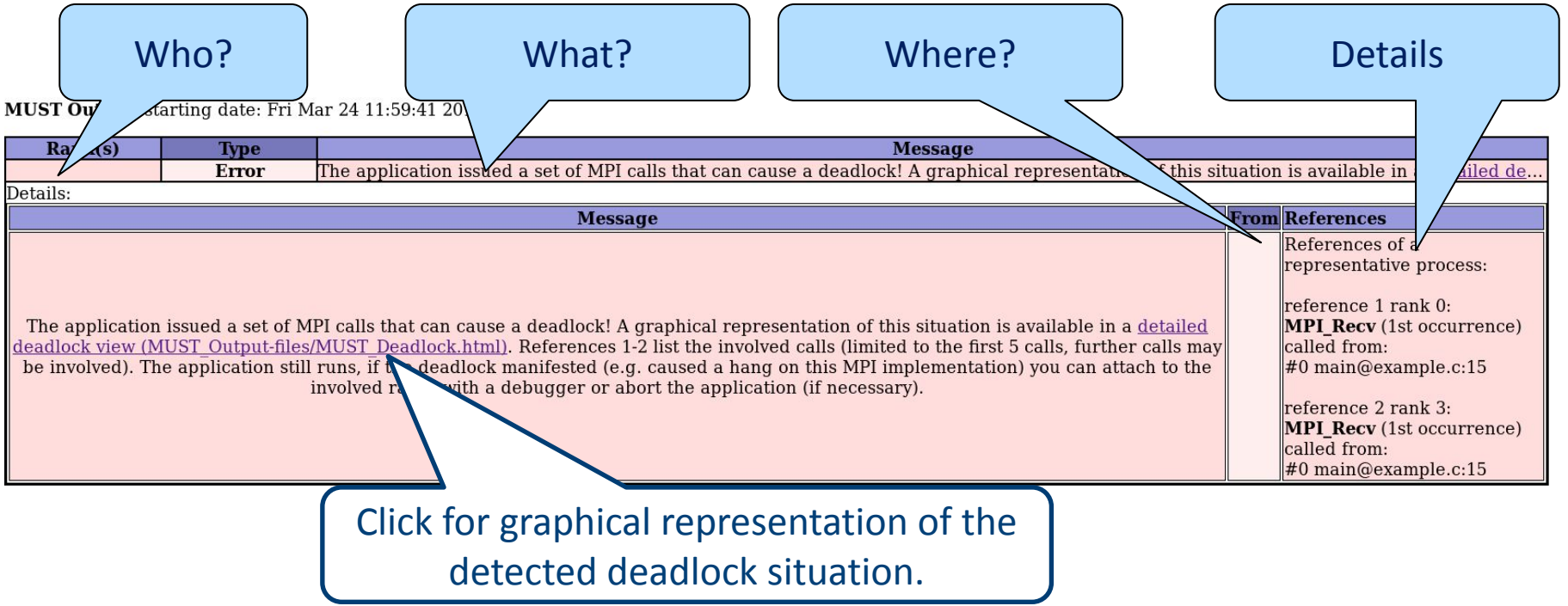
```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char** argv)
{
    int rank, size, buf[8];

    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;
}
```

Motivation

- MPI programming is error prone
- Portability errors (just on some systems, just for some runs)
- Bugs may manifest as:
  - Crash
  - Application hanging
  - Finishes
- Questions:
  - Why crashing/hanging?
  - Is my result correct?
  - Will my code also give correct results on another system?

- Tools help to pin-point these bugs

Must error report structure



Finally

Rank(s)	Type	Message
	Information	MUST detected no MPI usage errors nor any suspicious behavior during this application run.

Details:

Message	From	References
MUST detected no MPI usage errors nor any suspicious behavior during this application run.		

No further error detected

Hopefully this message applies to many applications

MUST - Basic Usage

- Apply MUST as an mpiexec wrapper, that's it:

```
% mpicc source.c -o exe
% $MPIRUN -n 4 ./exe
```

```
% mpicc -g source.c -o exe
% mustrun --must:mpiexec $MPIRUN -n 4 ./exe
```

or simply

```
% mustrun -n 4 ./exe
```

- After run: inspect "MUST_Output.html"
- "mustrun" (default config.) uses an extra process:
  - I.e.: "mustrun -np 4 …" will use 5 processes
  - Allocate the extra resource in batch jobs!
  - Default configuration tolerates application crash; BUT is slower (details later)

Advanced Usage

MUST - At Scale (highly recommended for >10 processes)

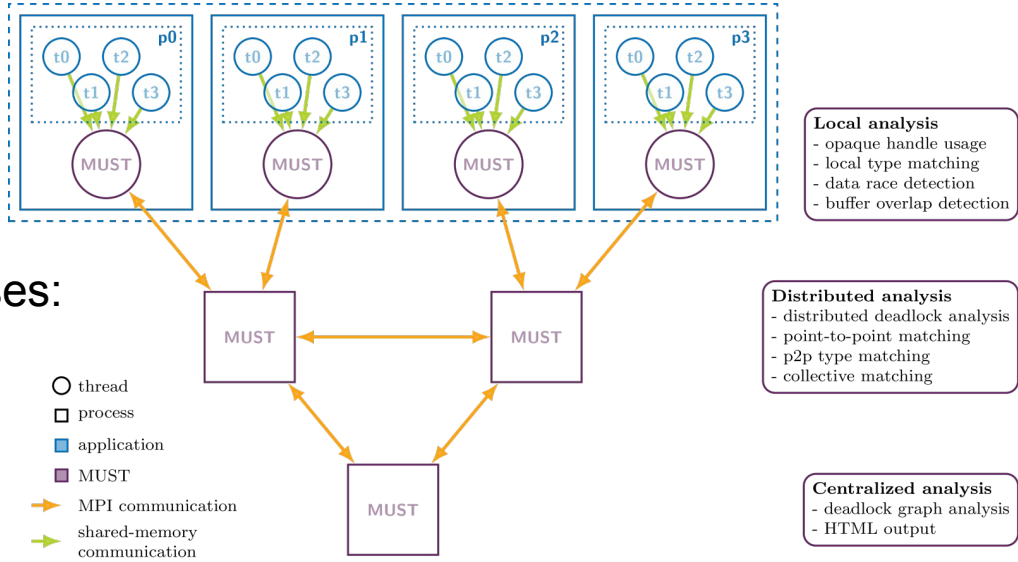
- Provide a branching factor (fan-in) for the tree infrastructure:

```
% mustrun -n 40 ./exe \
  --must:fanin 8
```

- Get info about the number of processes:

```
% mustrun -n 40 ./exe \
  --must:fanin 8 --must:info
```

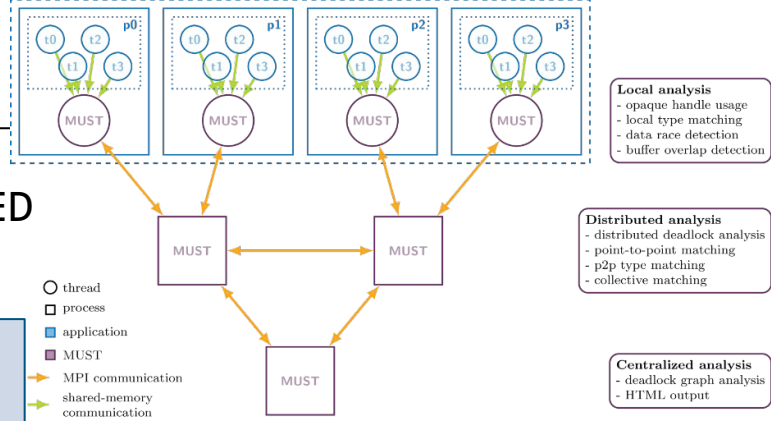
➢ This will give you the number of processes needed with tool attached



MUST – Execution Modes

	Application might crash	Application never crashes
Centralized analysis		`--must:nocrash`
	▪ 1 extra process ▪ Blocking communication	▪ 1 extra process ▪ Non-blocking communication
Distributed analysis	`--must:nodesize 8`	`--must:fanin 8`
	▪ 1 extra process per 7 application processes + tree ▪ Nodesize must be divisor of ranks sharing memory	▪ 1 extra process per 8 app processes + tree

MUST - Multithreading Support



- By default, MUST supports `MPI_THREAD_FUNNELED`
- For higher threading levels:

```
% mustrun -n 40 ./exe --must:hybrid
```

- This will raise the required level to `MPI_THREAD_MULTIPLE`!
- Some MPI might need env like: `MPICH_MAX_THREAD_SAFETY=multiple`
- Get info about the resources needed:

```
% mustrun -n 40 ./exe --must:hybrid --must:info
```

➢ This will give you the number of processes needed with tool attached

MPI runtime correctness checking with MUST – Advanced Usage

- We use Backward-cpp (or Dyninst) as an external lib for stacktraces
- Collecting stack traces can be costly. Select with `--must:stacktrace [backward|dyninst|none]`
- Supposed your application has no faults you won't need stacktraces ☺

Rank(s)	Type	Message	From	References
	Information	MUST detected no MPI usage errors nor any suspicious behavior during this application run.		

From
Representative location: **MPI_Init_thread** (1st occurrence) called from: #0 MAIN__@bt.f:90 #1 main@bt.f:319
Representative location: **MPI_Comm_split** (1st occurrence) called from: #0 MAIN__@bt.f:90 #1 main@bt.f:319

MUST – Filter file

- Use filter files to selectively exclude error/warning messages (avoid cluttered output)
- Format: `messageType:MUST_MESSAGE_TYPE:source`
  - `MUST_MESSAGE_TYPE`: kind of message to ignore (e.g. `MUST_WARNING_COMM_NULL`)
  - `source`: specific file (`filename.c`), specific function (`function_name`) or all sources (`*`)

- Example: Ignore `NULL` comm. warnings originating from `main.c` (needs stacktraces)

```
messageType:MUST_WARNING_COMM_NULL:src:main.c
```

- Example: Ignore all data type leak errors

```
messageType:MUST_ERROR_LEAK_DATATYPE:*
```

- Define and use a filter file:
  - `--must:filter-file <path-to-filter-file>`

MUST – More options

- Print help:
  - `--must:help`
- Select output format:
  - `--must:output {html|json|stdout}`
- Use with ddt:
  - Record error message information:
    - `--must:capture`
  - Replay under control of ddt:
    - `--must:reproduce --must:ddt`

MUST - Summary

- MPI runtime error detection tool
- Open source (BSD license)
  http://www.itc.rwth-aachen.de/MUST/
- Wide range of checks, strength areas:
  - Overlaps in communication buffers
  - Errors with derived datatypes
  - Deadlocks
- Largely distributed, able to scale with the application

Compiler-Aided Analysis

Issues with GPU offloading and Cuda-Aware MPI

```
#pragma omp parallel master
  {
#pragma omp task depend (inout: inbuf) depend (out: request)
    MPI_Irecv (inbuf, N, MPI_FLOAT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

#pragma omp task depend (inout: request)
    MPI_Wait (&request, MPI_STATUS_IGNORE);

#pragma omp task depend (inout: inbuf)
    Kernel<<<1, N, 0, stream[2]>>>(inbuf, outbuf);

#pragma omp task depend (in: outbuf) depend (out: request)
    MPI_Isend (outbuf, N, MPI_FLOAT, size - rank - 1, 123, MPI_COMM_WORLD, &request);

#pragma omp task depend (inout: request)
    MPI_Wait (&request, MPI_STATUS_IGNORE);
  }
```

Toolchain to detect errors for MPI + Cuda offloading

- CuSan: instrument CUDA kernel invocation + CUDA API
- MUST: instrument MPI memory accesses
- Archer/TSan: instrument local memory accesses

Using MUST + Archer

- Compile the MPI(+OpenMP) application just like described for Archer
- Using clang to compile with OpenMPI/IntelMPI/MPICH:
  - `export OMPI_CC=clang; export MPICH_CC=clang;`
  - `export OMPI_CXX=clang++; export MPICH_CXX=clang++;`
- Run MUST with TSan support:
  - `--must:tsan`

- For integration of TSan output into the MUST report, a helper-library must be linked into the application (not necessary for the setup on cosma):
  - `-Wl,--whole-archive ${MUST_ROOT}/lib/libonReportLoader.a -Wl,--no-whole-archive`

CuSan

CuSan is a tool to find races between CUDA calls and the host

- Consists of
  - a LLVM compiler plugin to instrument CUDA memory accesses & synchronization
  - runtime that passes information to ThreadSanitizer

Compiling with CuSan

make

- Often compiler selection possible with env variable

Use mpic++ also for C codes!

Makefile content:

```
MPICC ?= cusan-mpic++
demo: example.c
    $(MPICC) -fsanitize=thread -O1 -g -c $< -o $@.o
    $(MPICC) -fsanitize=thread $@.o -o $@
```

```
$> MPICC=cusan-mpic++ make -j 1
```

CMake

- During the configuration, CMake executes internal compiler checks, where we do not need CuSan instrumentation:
  - Need to disable wrapper

```
$> CUSAN_WRAPPER=OFF cmake .. \
    -DCMAKE_C_COMPILER=cusan-mpic++

$> make -j 1
```

Running MPI applications with MUST+CuSan

Slides gracefully provided by Dr. Alexander Hück

- MUST annotates memory accesses caused by MPI calls
  - Does not require special compilation process
- CuSan annotates CUDA memory accesses
- Enable MUST's race detection and compile application with CuSan
  - Detect races between CUDA kernels and MPI calls

```
$> mustrun --must:tsan -np 2 <app>
```

Rank(s)	Type	Message
0(1)	MUST_WARNING_DATARACE	Data race between a read of size 8 at MPI_Send@1 and a previous write of size 8 at __device_stub__kernel(int*, int)@2.

Details:

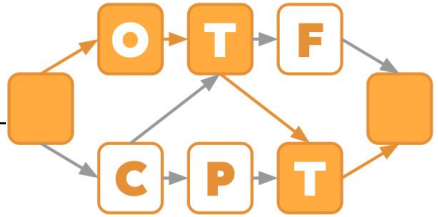
Message	From	References
Data race between a read of size 8 at MPI_Send@1 and a previous write of size 8 at __device_stub__kernel(int*, int)@2.	Representative location: **MPI_Send** (0th occurrence) called from: #0 MPI_Send@PnMPI/src/pnmpi/wrapper_c.c:22431 #1 main@send_ss_user_malloc_w_r_nok.c:54	References of a representative process: reference 1 rank 0: **MPI_Send** (0th occurrence) called from: #0 MPI_Send@PnMPI/src/pnmpi/wrapper_c.c:22431 #1 main@send_ss_user_malloc_w_r_nok.c:54 reference 2 rank 0: **__device_stub__kernel(int*, int)** (0th occurrence) called from: #0 __device_stub__kernel(int*, int)@send_ss_user_malloc_w_r_nok.c:11

On-the-fly Performance Model Factors for Multi-Level Parallelism

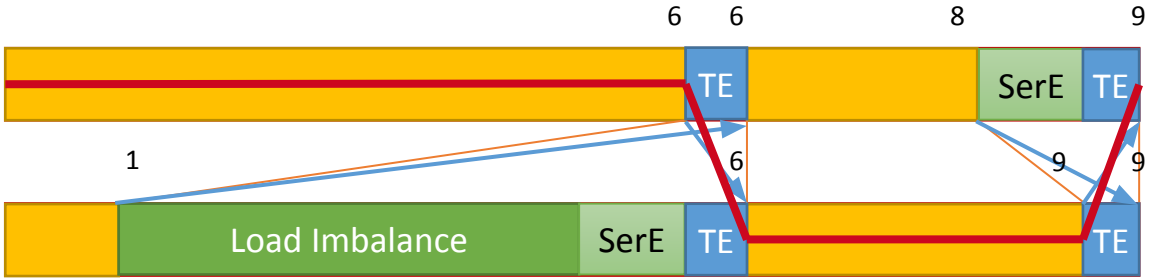
Dr. Joachim Jenke (jenke@itc.rwth-aachen.de)

i12 High Performance Computing
RWTH AACHEN UNIVERSITY

O-T-F critical path analysis for hybrid model factors



- Forward-only analysis
  - we only need the metrics of the critical path, but not the concrete path
- Treat time metrics as Lamport clock and implement the necessary propagation of this clock (MPI communication, OpenMP synchronization)
- Relevant metrics: useful computation, time outside the OpenMP runtime
- Relevant critical paths: global, process-local, thread-local
  - Formulation of MPI-specific and OpenMP-specific model factors in the paper



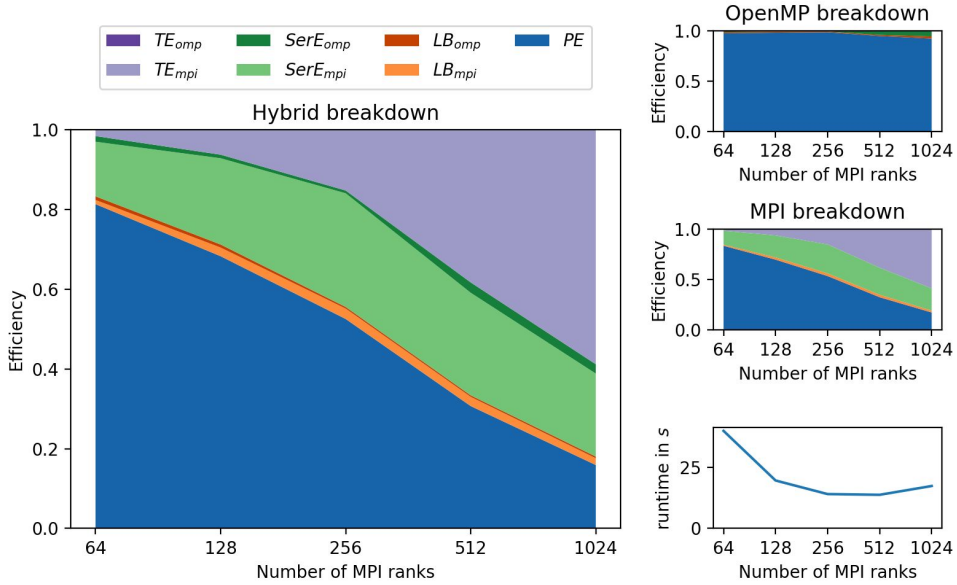
Perform OTF-CPT Analysis

- Execute the application with OTF-CPT:
  - `mpirun -np 8 env OTFCPT_OPTIONS=log_path=$PWD/data/data-8x12.txt OMP_NUM_THREADS=12 OMP_PLACES=cores OMP_PROC_BIND=close OMP_TOOL_LIBRARIES=libOTFCPT.so LD_PRELOAD=libOTFCPT.so ./a.out`
- For scaling experiment, vary num procs or threads
- Create metric plots:
  - `python3 CPT-plot.py data`

OTF-plot output

Generates 4 files (2x pdf, 2x png)

	64	128	256	512	1024
Parallel Efficiency	81.4	68.3	52.6	30.7	15.9
Load Balance	97.6	95.9	94.6	91.9	88.3
Communication Efficiency	83.3	71.2	55.6	33.4	18.0
Serialisation Efficiency	84.6	76.0	65.5	54.1	43.8
Transfer Efficiency	98.4	93.8	84.8	61.7	41.2
MPI Parallel Efficiency	83.5	69.7	53.4	32.4	17.3
MPI Load Balance	98.7	96.9	95.3	93.0	90.3
MPI Communication Efficiency	84.6	72.0	56.1	34.9	19.1
MPI Serialisation Efficiency	85.9	76.7	66.1	56.5	46.4
MPI Transfer Efficiency	98.5	93.8	84.9	61.8	41.2
OMP Parallel Efficiency	97.5	97.9	98.4	94.7	92.1
OMP Load Balance	98.9	98.9	99.3	98.9	97.7
OMP Communication Efficiency	98.5	99.0	99.1	95.7	94.3
OMP Serialisation Efficiency	98.6	99.0	99.2	95.8	94.3
OMP Transfer Efficiency	100.0	100.0	99.9	99.9	100.0



NHR4CES NHR for Computational Engineering Science

Performance Optimisation and Productivity
A Centre of Excellence in HPC

i12 High Performance Computing

RWTH AACHEN UNIVERSITY