Performance and Correctness with the Vampir Tool-Suite and MUST

Dr. Matthias S. Müller
Tobias Hilbrich
Email: {matthias.mueller, tobias.hilbrich}@tu-dresden.de
Content

- Welcome & Motivation
- Performance Optimization with Vampir
- Runtime Error Detection with MUST
- Conclusions & Wrap-Up
Part I: Welcome & Motivation
High performance computing without tools:

Questions:
- Did I utilize the machine efficiently?
- Are my results correct?
- Why aren’t I getting a result?

Tools help developers understand HPC systems
We got tools to assist you:

- Vampir: Performance optimization
- MUST: Correct MPI usage
HPC systems evolve; Tools need to adapt:

Paradigms

MPI+ Accelerator+ Threads
Co-Array Fortran
SHMEM
OpenCL

Heterogeneous/ Hybrid

MPI+CUDA
PGAS+CUDA
MPI+Threads

Fault Tolerance
Towards Exascale ...

TOP 500 – Largest Machine

Scalability

Year

#Cores

Make all this accessible!

Challenges

Matthias Müller and Tobias Hilbrich
We want to cover:

- How can our tools help you
- Basic tool usage
- New features and success stories
- Roadmaps
- Detailed training in individual sessions
Part II: Performance Optimization with Vampir

Dr. Matthias S. Müller
Tobias Hilbrich

Email: {matthias.mueller, tobias.hilbrich}@tu-dresden.de
Content

- Overview
- Technology
- Example
- Tracing
- Visualization
- Meeting the Challenges
- Score-P
- Roadmap
Content

- Overview
- Technology
- Example
- Tracing
- Visualization
- Meeting the Challenges
- Score-P
- Roadmap
Overview

- Vampir: tool to optimize application performance
- Approach: Event tracing
- Insights with Vampir include:
  - Communication/Computation ratio
  - Communication statistics (Bandwidth/Latency)
  - Inefficient communication patterns
  - Excessive synchronization time
  - Inefficient IO, or GPGPU usage
- However: Vampir is no automatic tool
  - A GUI guides the user to performance issues
Content

- Overview
- Technology
- Example
- Tracing
- Visualization
- Meeting the Challenges
- Score-P
- Roadmap
Content

- Overview
- Technology
- Example
- Tracing
- Visualization
- Meeting the Challenges
- Score-P
- Roadmap
Example – COSMO-SPECS

- COSMO-SPECS is a coupled model system
  - COSMO: regional weather forecast model developed at the German Weather Service (DWD), formerly known as “Lokal-Modell” (LM)
  - SPECS: detailed cloud microphysics model developed at the Leibniz Institute for Tropospheric Research, Leipzig (IfT)
- Used for detailed study of the interactions between aerosols, clouds, and precipitation
Example – COSMO-SPECS Domain Decomposition

- COSMO computational grid:
  - 3D regular rectangular

- COSMO domain decomposition:
  - 2D (horizontal) decomposition into MxN processor domains
  - No dynamic load balancing

- SPECS uses the same data structures and decomposition
  - Technically “just” a replacement of COSMOs original cloud model
  - But causes **huge load imbalance**, since (heterogeneous) cloud cover determines local work load
Example – COSMO-SPECS Trace

- SGI Altix 4700, 100 cores
- Forecast time: 30min
- Run time: ~2:55h
- Usage of VT API to switch tracing off. Only the
  - first 3 time steps,
  - one time step at 25%, 50%, and 75% of the time steps, and
  - the last 3 time steps are recorded.
- Groups
  - METEO: COSMO weather model
  - MP: SPECS cloud microphysics calculation
  - MP_UTIL: Utilities for cloud microphysics (advection, clipping)
  - COUPLE: Adjustment of flux fields of COSMO (for microphysics)
Example – COSMO-SPECS Trace Visualization

First 3 time steps of COSMO-SPECS run

SPECS performs 20 sub-time steps per COSMO step

Last 3 time steps of COSMO-SPECS run

Everything ok

Heavy load imbalance

Cloud grows in grid cells of these MPI ranks

56% MPI!
Content

- Overview
- Technology
- Example
- **Tracing**
- Visualization
- Meeting the Challenges
- Score-P
- Roadmap
VampirTrace is a monitor for event tracing

**VampirTrace:**
- Open source, BSD license
- Part of every OpenMPI distribution

**Usage:**
- Attach VampirTrace library to application
- Various control and tuning options
Tracing – Instrumentation

Data Sources

**Source level**
- Compiler
- Manual
- OpenMP with Opari
- Tau PDT

**Binary level**
- Dyninst

**MPI Profiling Interface**
- MPI

**Runtime/Library**
- MPI Correctness
- Pthreads
- NVIDIA CUDA
- OpenCL
- 3rd party libraries
- External counters
- Plugin counters
- Java tracing

**Operating System**
- Resource usage
- Memory allocation

**Hardware**
- Performance counters
- I/O
- CPU ID
Tracing – The Hard Part

- Event tracing requires trade-off’s:
  - Only add the data sources you need
  - Limit granularity (i.e., filtering)
- Good practice: start with a profile run
Tracing – Usage

Compiler wrappers provide instrumentation:

CC=icc
CXX=icpc
F90=ifort
MPICC=mpicc

CC=vtcc
CXX=vtcxx
F90=vtf90
MPICC=vtcc -vt:cc mpicc

Basic VampirTrace usage:
- Exchange compiler commands
- Re-compile & re-link
- Run as usual (details follow)

This adds compiler based instrumentation
NAS Parallel Benchmarks

- Activate the VampirTrace compiler wrappers:

  ```
  % vi config/make.def
  #F77 = mpif77
  F77 = vtf77 -vt:f77 mpif77
  ```

- Build a kernel:

  ```
  % make bt CLASS=W NPROCS=4
  ```
Tracing – Example (2)

- Remember: Start with a profile run
- Statistics run with VampirTrace:

```bash
% cd bin
% export VT_MODE=STAT
% mpirun -np 4 bt.W.4
...

excl. incl. excl. time incl. time name
  time  time  calls  /call  /call
2.867s 2.867s  2412  1.189ms  1.189ms MPI_Wait
1.060s 1.060s   1  1.060s   1.060s MPI_Init
0.765s 0.765s  202  3.788ms  3.788ms MPI_Waitall
0.483s 0.483s 1678149  0.287us  0.287us binvcrhs_
0.304s 0.304s 1678149  0.181us  0.181us matmul_sub_
0.298s 0.635s  402  0.741ms  1.578ms x_solve_cell_
0.297s 0.664s  402  0.740ms  1.651ms z_solve_cell_
0.264s 0.571s  402  0.656ms  1.419ms y_solve_cell_
0.210s 0.210s 1678149  0.125us  0.125us matvec_sub_
0.163s 0.163s  202  0.807ms  0.807ms compute_rhs_
```

Displayed 10 from 65 functions.

Tracing very short but frequent calls is impractical

⇒ Candidates for filtering
Tracing – Example (3)

Let's trace with filtering:

% nano filter.txt
binvcrhs_ -- 0
matmul_sub_ -- 0
matvec_sub_ -- 0
binvrhs_ -- 0
exact_solution_ -- 0
% export VT_MODE=TRACE
% export VT_FILTER_SPEC=filter.txt
% mpirun -np 4 bt.W.4

NAS Parallel Benchmarks 3.3 -- BT Benchmark ...

Resulting trace file “bt.W.4.otf”:

% vampir bt.W.4.otf
High MPI ratio due to 4 ranks on 2 cores
Tracing – Adding Detail

- We traced:
  - Function calls
  - MPI calls

- We might add (examples):
  - More buffering for the trace (for longer runs):
    \[ \% \text{export VT BUFFER SIZE=1000M} \]
  - PAPI performance counters:
    \[ \% \text{export VT METRICS=PAPI FP OPS:PAPI L2 TCM} \]
  - IO or memory allocation tracing:
    \[ \% \text{export VT MEMTRACE=yes} \]
    \[ \% \text{export VT IOTRACE=yes} \]
Tracing – Compiler Instrumentation Limits

- Default instrumentation: compiler instrumentation
  - Instruments EVERY function call
  - Even with filter: VampirTrace invoked per function call
  ⇒ Can cause high overhead

- More advanced instrumentation techniques:
  - Manual (laborious, but always available)
  - Binary instrumentation with Dyninst
  - Source-to-source instrumentation with TAU PDT

Advice: If compiler instrumentation causes high overhead, use one of the above
Instrumentation and execution with PDT:

```
% vtcc -vt:cc mpicc -vt:inst tauinst
    -vt:tau '-f tau_filter.txt' source.c -o exe
% mpirun -np 4 ./exe
```

Activates PDT instrumentation (Optional VampirTrace feature)

Sets a PDT filter (PDT specific format)

Instrumentation and execution with Dyninst:

```
% mpicc source.c -o exe -g
% mpirun -np 4 vtrun -mpi --dyninst ./exe
```

No modifications for compile&link necessary

“vtrun” attaches VampirTrace and invokes dyninst instrumentation
Tracing – vtsetup

A tool that aids in setting up VampirTrace runs

Infobox: PAPI Performance Counter:
Specify the counter that shall be recorded during the performance measurement.
Default: -
Variable: VT_METRICS
Content

- Overview
- Technology
- Example
- Tracing
- Visualization
- Meeting the Challenges
- Score-P
- Roadmap
Visualization – Vampir Modes (1)

Directly on a local machine:

```bash
% vampir
```
On local machine with VampirServer:

\%
start -n 12 -p 30088

\%
vampir
Connecting to VampirServer:

Open Remote …
VampirServer on HPC system; GUI on desktop:

- Start VampirServer:
  ```bash
  % vampirserver start -n 12 -p 30088
  ```

- Forward local port to HPC system:
  ```bash
  % ssh -L 30000:hpc-system:30088 login@hpc-system
  ```

- Access GUI on desktop:
  ```bash
  % vampir
  ```

E.g. if X forwarding is too slow:
  ```bash
  % ssh -L 30000:hpc-system:30088 login@hpc-system
  ```
Content

- Overview
- Technology
- Example
- Tracing
- Visualization
- Meeting the Challenges
- Score-P
- Roadmap
Meeting the Challenges

- HPC systems evolve; tools need adapt:
  - Scalability
  - Paradigms
  - Heterogeneity
  - Fault tolerance
  - Usability
- Several extensions address these challenges
Meeting the Challenges – Performance Radar
Meeting the Challenges – Performance Radar

![Performance Radar Diagram]

Movable!

- Timeline
- Values of Metric "PAPI_FP_OPS" over Time

Processes: 0 to 10

0 s to 50 s

WRF_TERMIO_DUP

Values range from 0 G to 4 G
Meeting the Challenges – Overlays
Meeting the Challenges – Overlays

[Image of a trace view interface with a heatmap and timeline showing processes and metrics like PAPI_FP_OPS.]
Meeting the Challenges – Overlays
Meeting the Challenges – Overlays
Meeting the Challenges – Derived Counters
Meeting the Challenges – Compare View
Meeting the Challenges – Compare View

The image shows a Compare View interface with timelines and processes. The timeline is divided into segments, each representing different processes and sub-processes such as WRF, MODULE_INTEGRATE::INTEGRATE, SOLVE_INTERFACE, and SOLVE_EM. The timeline is marked with time intervals and labels, indicating the duration and sequence of these processes. The interface appears to be used for analyzing and comparing the performance or output of these processes.
Meeting the Challenges – Compare View

The diagram shows a timeline and function summary for two processes, Process 0 and Process 1. The timeline visualizes the execution times and activities of the processes, while the function summary provides a breakdown of the accumulated exclusive time per function and category. The diagram highlights the differences in execution times and the distribution of time spent on various tasks between the two processes.
Tracing long running applications is hard

Case:
- Long running application (~50min)
- Non-reproducible bad performance in iterations
- Coarse-grained traces gave no insight
- Fine-grained traces would not fit into memory

Solution:
- VampirTrace feature “Rewind”
- Allows to discard parts of the trace buffer
- Controlled via VampirTrace instrumentation API
First and last iterations + 4 “slow” iterations
Meeting the Challenges – Long Tracing Runs (3)

Zoom into a slow iteration
Meeting the Challenges – Long Tracing Runs (4)

All except rank 20 need to wait

Can also be spotted in Radar

Process 20 total CPU cycles suddenly drop for > 6 seconds
Case:
- Particle in cell code on NVIDIA GPGPUs
- Multi-level parallelism: MPI + CUDA + pthreads
- Performance is a black box without tools

Solution:
- VampirTrace support for CUDA
- Close collaboration with NVIDIA
Meeting the Challenges – CUDA, PicOnGPU
Meeting the Challenges – CUDA, PicOnGPU
Meeting the Challenges – CUDA, PicOnGPU
Meeting the Challenges – CUDA, PicOnGPU
Goal:
- Support hybrid system and software architecture at 10 Petascale
- Show MPI and GPGPU programming
- Do full system performance profiling and tracing

Facts:
- Jaguar + Successor
- >220,000 cores
- 200,448 monitored MPI processes
- >20 Tera-bytes of performance data
- 21,515 VampirServer processes
Meeting the Challenges – Collaboration with ORNL

Timeline
Meeting the Challenges – Vampir I/O Analysis

- recording of POSIX I/O and MPI I/O operations with VampirTrace
- embedding of node local data like InfiniBand statistics
- on demand inclusion of external performance data from:
  - I/O network
  - storage controller
  - file servers
- within Vampir:
  - counter timelines for the host based and the external data
  - specialized I/O display to show:
    - details for single I/O events
    - grouping of events for the current portion of the timeline based on the filename, the type of the I/O record (read, write, …) and some more
    - I/O request size statistics
Meeting the Challenges – Vampir I/O Analysis

- Metadata rate
- Metadata server load
- Metadata server load
- Detailed per file and summary statistics
- Detailed per event data
Meeting the Challenges – Vampir I/O Analysis

- **application + VampirTrace**
- **trace**
- **performance data**
- **database**
- **secondary counters**

**Network Components:**
- **node**
- **MDS**
- **OSS**
- **RAID**
- **SAN**

**Data Flow:**
- Application + VampirTrace → Trace → Performance Data → Database → Secondary Counters
Starting Point: 6 from 576 IOR processes
IOR: write phase

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0-100</td>
<td>write</td>
<td>write</td>
<td>write</td>
<td>write</td>
<td>write</td>
<td>write</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>200-600</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Barrier</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MPI_Barrier</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MPI_Barrier</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MPI_Barrier</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MPI_Barrier</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MPI_Barrier</td>
</tr>
</tbody>
</table>
IOR: Write Phase

- 650 vs. 800 MB/s per DDN port

*ExtCounter 0:1, Values of Counter "ddn01 - port 1: write bandwidth" over Time*

*ExtCounter 0:1, Values of Counter "ddn01 - port 4: write bandwidth" over Time*
IOR: Read Phase

Break down

Bad Disk Sector
Content

- Overview
- Technology
- Example
- Tracing
- Visualization
- Meeting the Challenges
- Score-P
- Roadmap
Score-P – Motivation

From Sameer Shende, used during VI-HPS tuning workshop 8
Score-P – Overview

Tool ecosystem:

- Vampir
- Scalasca
- TAU
- Periscope

VampirTrace
OTF
EPILOG / CUBE
TAU native formats
Online measurement

Consequence:
- Each tool requires its own measurement system
- Tool combinations limited
- Higher tool development effort
Score-P – Overview

Tool ecosystem:

- Vampir
- Scalasca
- TAU
- Periscope

Score-P

Goals:
- Single measurement system
- Better learning curve
- More tool integration
- Reduced tool development costs
Score-P – Partners

- Forschungszentrum Jülich, Germany
- German Research School for Simulation Sciences, Aachen, Germany
- Gesellschaft für numerische Simulation mbH Braunschweig, Germany
- RWTH Aachen, Germany
- Technische Universität Dresden, Germany
- Technische Universität München, Germany
- University of Oregon, Eugene, USA
Score-P – State

- Preview versions since 2010
- First release December 2011
- Score-P will replace VampirTrace
- Feature migrations are ongoing:
  - CUDA support on its way
  - TAU PDT support available
  - OPARI2 (in Score-P) supersedes OPARI in VT
  - …
- Vampir Tutorials will use Score-P starting mid 2012
Score-P – Usage

Prefix commands provide instrumentation:

- CC=icc
- CXX=icpc
- F90=ifort
- MPICC=mpicc

As with VampirTrace, environmentals control tool:
- Profile/Tracing mode
- Buffer sizes
- Optional features
Content

- Overview
- Technology
- Example
- Tracing
- Visualization
- Meeting the Challenges
- Score-P
- Roadmap
Roadmap

June 2012

- Thinned out trace files (removal of similar/redundant tasks)
- Temporal pattern highlighting during tracing
- OpenMP 3.0 task support
- VampirServer: improved remote connections
- ScoreP becomes official measurement system for Vampir
- CUDA 4.1 support with native CUPTI based tracing

November 2012

- Further MPI performance metrics in performance radar
- Support for HMPP Instrumentation API
- OTF2 extensions
- Temporal pattern based trace compression
- Score-P: Scalability, rewind, selective tracing

Longer term

- In memory performance event data compression
- Alignment based trace comparison
- Improved correlation of event types.
- Scalability improvements
- Score-P: measurement of system background counters (I/O, energy, …)
- Vampir: Background counter view
- PGAS/SHMEM support
Part III: Runtime Error Detection with MUST

Dr. Matthias S. Müller
Tobias Hilbrich
Email: {matthias.mueller, tobias.hilbrich}@tu-dresden.de
Content

- Motivation
- Overview
- MPI Usage Errors
- Technology
- Example
- Usage
- Meeting the Challenges
Content

- Motivation
- Overview
- MPI Usage Errors
- Technology
- Example
- Usage
- Meeting the Challenges
Motivation

MPI programming is error prone

Results of an application run:
- Crash
- Application hanging
- Result

Questions:
- Why crash/hang?
- Is my result correct?
- Will my code also give correct results on another system?
Content

- Motivation
- Overview
- MPI Usage Errors
- Technology
- Example
- Usage
- Meeting the Challenges
Overview

Example:

```c
... MPI_Type_contiguous (2, MPI_INTEGER, &newtype);
MPI_Send (buf, count, newtype, target, tag, MPI_COMM_WORLD);
...```

MUST:

- Detects MPI usage errors at runtime
- A library that is attached to application
- Strengths: Deadlock detection, collective verification, type matching
- Goal: scalable correctness checks
- Cooperation with LLNL and LANL
Complications
- Calls with many arguments
- Fortran: many arguments have type INTEGER
- Several argument restrictions

Example:

```c
MPI_Send(
    buf,
    count,
    MPI_INTEGER,
    target,
    tag,
    MPI_COMM_WORLD);
```
Complications
- Many types of resources
- Leaks
- MPI internal limits

Example:

```c
MPI_Isend (... , &request);
MPI_Finalize ();
```

Applications should complete the outstanding communication associated with `request`
Complications
- Memory regions passed to MPI must not overlap (except send-send)
- Derived datatypes can span non-contiguous regions
- Collectives can both send and receive

Example:

```c
MPI_Isend (&(buf[0])/*buf*/ , 5/*count*/ , MPI_INT ,...);
MPI_Irecv (&(buf[4])/*buf*/ , 5/*count*/ , MPI_INT ,...);
```

Overlaps element buf[4] from the MPI_Isend call!
Complications

– Complex derived types
– Types match if the signature matches, not their constructors
– Partial receives

Example 1:

Task 0

MPI_Send (buf, 1, MPI_INT)

Task 1

MPI_Recv (buf, 1, MPI_INT)

No Error, types match
Example 2:
- Consider type T1 = {MPI_INT, MPI_INT}

Task 0
MPI_Send (buf, 1, T1)

Task 1
MPI_Recv (buf, 2, MPI_INT)

No Error, types match

Example 3:
- T1 = {MPI_INT, MPI_FLOAT}
- T2 = {MPI_INT, MPI_INT}

Task 0
MPI_Send (buf, 1, T1)

Task 1
MPI_Recv (buf, 1, T2)

Mismatch: MPI_FLOAT != MPI_INT
Complications:
- Non-blocking communication
- Complex completions (Wait{all, any, some})
- Non-determinism (e.g. MPI_ANY_SOURCE)
- Choices of implementation in MPI standard

Example:

Deadlock: tasks wait for each other
How to visualise/understand deadlocks?
- Common approach waiting-for graphs (WFGs)
- One node for each rank
- Rank X waits for rank Y => node X has arc to node Y

Consider situation from Example 1:

Visualization:

Deadlock criterion: cycle (For simple cases)
Technology

1. CPU → Application → MUST
2. CPU → Application → MUST
3. CPU → Application → MUST
...
N. CPU → Application → MUST

Non-Local Correctness Checking

Correctness Report

Matthias Müller and Tobias Hilbrich
Content

- Motivation
- Overview
- MPI Usage Errors
- Technology
- Example
- Usage
- Meeting the Challenges
Example – Synthetic Code

(1)  MPI_Init (&argc,&argv);
(2)  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
(3)  MPI_Comm_size (MPI_COMM_WORLD, &size);
(4)

(5)  //1) Create a datatype
(6)  MPI_Type_contiguous (2, MPI_INT, &newType);
(7)  MPI_Type_commit (&newType);
(8)

(9)  //2) Use MPI_Sendrecv to perform a ring communication
(10)  MPI_Sendrecv (  
(11)      sBuf, 1, newType, (rank+1)%size, 123,
(12)      rBuf, sizeof(int)*2, MPI_BYTE, (rank-1+size) % size, 123,
(13)      MPI_COMM_WORLD, &status);
(14)

(15)  //3) Use MPI_Send and MPI_Recv to perform a ring communication
(16)  MPI_Send ( sBuf, 1, newType, (rank+1)%size, 456,
(17)        MPI_COMM_WORLD);
(18)  MPI_Recv ( rBuf, sizeof(int)*2, MPI_BYTE, (rank-1+size) % size, 456,
(19)        MPI_COMM_WORLD, &status);
(20)

(21)  MPI_Finalize ();
### Example – Type mismatch

<table>
<thead>
<tr>
<th>Rank</th>
<th>Thread</th>
<th>Type</th>
<th>Message</th>
<th>From</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>Error</td>
<td>A send and a receive operation use datatypes that do not match! Missmatch occurs at (CONTIGUOUS)[0] (MPI_INT) in the send type and at (MPI_BYTE) in the receive type (consult the MUST manual for a detailed description of datatype positions). The send operation was started at reference 1, the receive operation was started at reference 2. (Information on communicator: MPI_COMM_WORLD) (Information on send of count 1 with type:Datatype created at reference 3 is for C, commited at reference 4, based on the following type(s): { MPI_INT}Typemap = { (MPI_INT, 0), (MPI_INT, 4)}) (Information on receive of count 8 with type:MPI_BYTE)</td>
<td>MPI_Sendrecv from: #0 <a href="mailto:main@test.c">main@test.c</a>:54 #1 <a href="mailto:start_main@libc.so">start_main@libc.so</a></td>
<td>reference 1 rank 0: MPI_Sendrecv from: #0 <a href="mailto:main@test.c">main@test.c</a>:54 #1 <a href="mailto:start_main@libc.so">start_main@libc.so</a></td>
</tr>
</tbody>
</table>

Center for Information Services & High Performance Computing

Matthias Müller and Tobias Hilbrich
<table>
<thead>
<tr>
<th>Rank</th>
<th>Thread</th>
<th>Type</th>
<th>Message</th>
<th>From</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>There are 1 datatypes that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these datatypes:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Error</td>
<td>- Datatype 1: Datatype created at reference 1 is for C, committed at reference 2, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4)}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>There are 1 datatypes that are not freed when MPI_Finalize was issued, a quality application should free all MPI resources before calling MPI_Finalize. Listing information for these datatypes:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Error</td>
<td>- Datatype 1: Datatype created at reference 1 is for C, committed at reference 2, based on the following type(s): { MPI_INT}Typemap = {(MPI_INT, 0), (MPI_INT, 4)}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example – Deadlock (potential)

MUST Deadlock Details, date: Thu Mar 22 15:24:15 2012.

Back to MUST error report

Message

The application issued a set of MPI calls that can cause a deadlock! The graphs below show details on this situation. This includes a wait-for graph that shows active waiting dependencies in the deadlock situation, a legend for this graph and a call stack view. The application still runs, if the deadlock manifested (e.g., caused a hang on this MPI implementation) you can attach to the involved ranks with a debugger or abort the application (if necessary).

Wait-for Graph

```
MPI_Send@0
     tag=456

MPI_Send@1
     tag=456

MPI_Send@2
```

Legend

```
Active MPI Call

Sub Operation

A waits for B and C

A waits for B or C
```

Call Stack

```
libe-213.so:(0x1eef)

Ranks: 0, 1, 2

main@vihps8_2011.c:60

Ranks: 0, 1, 2

MPI_Send
```
Content

- Motivation
- Overview
- MPI Usage Errors
- Technology
- Example
- Usage
- Meeting the Challenges
**Usage**

- MUST uses an mpirun wrapper:

  ```
  % mpicc source.c -o exe
  % mpirun -np 4 ./exe
  ```

  ```
  % mpicc source.c -o exe
  % mustrun -np 4 ./exe
  ```

- After run: inspect “MUST_Output.html”
- “mustrun” uses an extra process:
  - I.e.: “mustrun –np 4 …” will use 5 tasks
  - Allocate the extra resource in batch jobs
Usage – Operation Modes

- MUST causes overhead at runtime
- MUST expects a crash at any time
- Blocking communication ensures error detection

⇒ Can cause high overhead

If application does not crash:
  - Add “--must:nocrash” to the mustrun command
  - Uses aggregated non-blocking communication
  - Provides substantial speed up

More options: “mustrun --help”
Content

- Motivation
- Overview
- MPI Usage Errors
- Technology
- Example
- Usage
- Meeting the Challenges
Meeting the Challenges

Currently: core concern scalability

Distributed non-Local Correctness Checking

Correctness Report
Meeting the Challenges – Scalability

Local checks (Scalable):
- Integer validation
- Integrity checks (pointers valid, etc.)
- Operation, Request, Communicator, Datatype, Group usage
- Resource leak detection
- Memory overlap checks

Non-local checks (Scalability in progress):
- Collective verification
- Lost message detection
- Type matching (For P2P and collectives)
- Deadlock detection (with root cause visualization)
Part 4 VI: Conclusions & Wrap Up

Dr. Matthias S. Müller
Tobias Hilbrich

Email: {matthias.mueller, tobias.hilbrich}@tu-dresden.de
Tool Availability

- **VampirTrace**: Open source
  ⇒ [http://tinyurl.com/vampirtrace](http://tinyurl.com/vampirtrace)

- **Vampir**: Commercial; LLNL/LANL have licenses

- **MUST**: Open source
  ⇒ [http://tu-dresden.de/zh/must/](http://tu-dresden.de/zh/must/)

- **VM with full installation available, contact us**

- **Contact:**
  - {tobias.hilbrich, matthias.mueller}@tu-dresden.de
  - Bugs/wishes/ideas:
    vampirsupport@zih.tu-dresden.de
Conclusions

Vampir suite:
- Tool for performance optimization
- Many recent extensions
- Scalable to 200,000 cores
- New features include:
  - CUDA support, Counter overlays, Clustering, PDT/Dyninst instrumentation, Search functions, Performance radar, derived counters, Trace Rewind

MUST:
- Tool to detect MPI usage errors at runtime
- Scalable to about 1000 cores (more with less checks)
- 10,000 core scalability in development