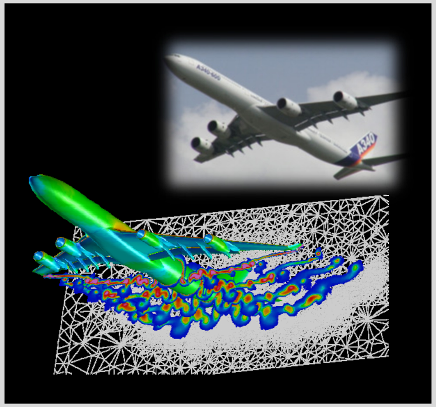


High Performance Computing

ADVANCED SCIENTIFIC COMPUTING

Prof. Dr. – Ing. Morris Riedel

Adjunct Associated Professor

School of Engineering and Natural Sciences, University of Iceland, Reykjavik, Iceland

Research Group Leader, Juelich Supercomputing Centre, Forschungszentrum Juelich, Germany

@Morris Riedel @MorrisRiedel @MorrisRiedel

LECTURE 9

Debugging & Profiling & Performance Toolsets

November 07, 2019

Room V02-258

UNIVERSITY OF ICELAND

SCHOOL OF ENGINEERING AND NATURAL SCIENCES

FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

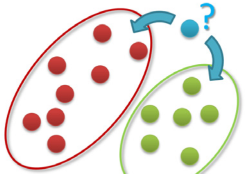
JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

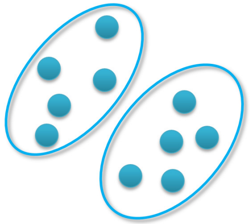
DEEP Projects

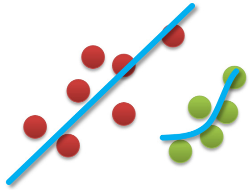
HELMHOLTZ RESEARCH FOR GRAND CHALLENGES

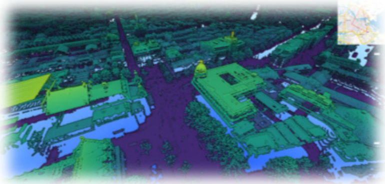
HAICU HELMHOLTZ ARTIFICIAL INTELLIGENCE COOPERATION UNIT

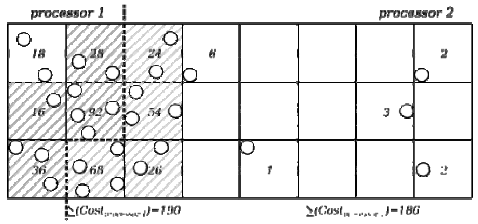
Review of Lecture 8 – Parallel & Scalable Machine & Deep Learning

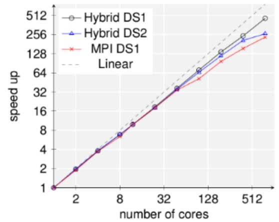


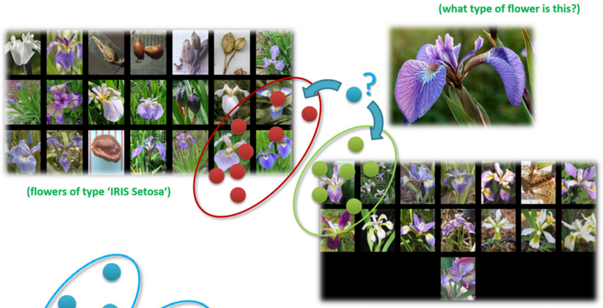


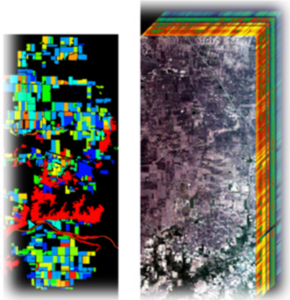












Scenario 'pre-processed data', 10xCV **serial**: accuracy (min)

γ/C	1	10	100	1000	10 000
2	48.90 (18.81)	65.01 (19.57)	73.21 (20.11)	75.55 (22.53)	74.42 (21.21)
4	57.53 (16.82)	70.74 (13.94)	75.94 (13.53)	76.04 (14.04)	74.06 (15.55)
8	64.18 (18.30)	74.45 (15.04)	**77.00** (14.41)	75.78 (14.65)	74.58 (14.92)
16	68.37 (23.21)	76.20 (21.88)	76.51 (20.69)	75.32 (19.60)	74.72 (19.66)
32	70.17 (34.45)	75.48 (34.76)	74.88 (34.05)	74.08 (34.03)	73.84 (38.78)

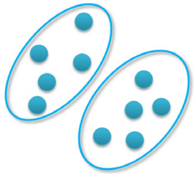
First Result:
best parameter set from 14.41 min to 1.02 min

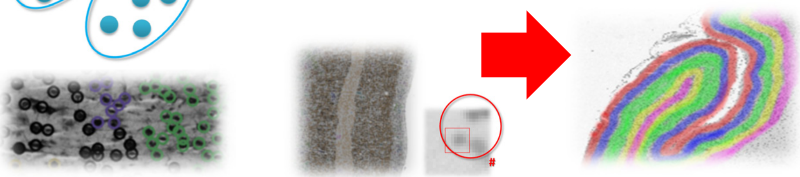
Second Result:
all parameter sets from ~9 hours to ~35 min

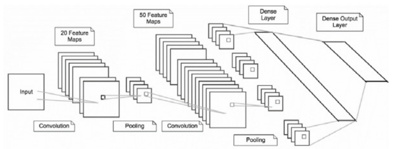
Scenario 'pre-processed data', 10xCV **parallel**: accuracy (min)

γ/C	1	10	100	1000	10 000
2	75.26 (1.02)	65.12 (1.03)	73.18 (1.33)	75.76 (2.35)	74.53 (4.40)
4	57.60 (1.03)	70.88 (1.02)	75.87 (1.03)	76.01 (1.33)	74.06 (2.35)
8	64.17 (1.02)	74.52 (1.03)	**77.02 (1.02)**	75.79 (1.04)	74.42 (1.34)
16	68.57 (1.33)	76.07 (1.33)	76.40 (1.34)	75.26 (1.05)	74.53 (1.34)
32	70.21 (1.33)	75.38 (1.34)	74.69 (1.34)	73.91 (1.47)	73.73 (1.33)

[14] G. Cavallaro & M. Riedel & J.A. Benediktsson et al., 'On Understanding Big Data Impacts in Remotely Sensed Image Classification Using Support Vector Machine Methods', Journal of Applied Earth Observations and Remote Sensing







TensorFlow Keras jupyter

[15] A. Gulli et al.

[12] Image sources: Species Iris Group of North America Database, www.signa.org

[13] M. Goetz and M. Riedel et al, Proceedings IEEE Supercomputing Conference, 2015

Outline of the Course

1. High Performance Computing
2. Parallel Programming with MPI
3. Parallelization Fundamentals
4. Advanced MPI Techniques
5. Parallel Algorithms & Data Structures
6. Parallel Programming with OpenMP
7. Graphical Processing Units (GPUs)
8. Parallel & Scalable Machine & Deep Learning
9. Debugging & Profiling & Performance Toolsets
10. Hybrid Programming & Patterns
11. Scientific Visualization & Scalable Infrastructures
12. Terrestrial Systems & Climate
13. Systems Biology & Bioinformatics
14. Molecular Systems & Libraries
15. Computational Fluid Dynamics & Finite Elements
16. Epilogue

+ additional practical lectures & Webinars for our hands-on assignments in context

- Practical Topics
- Theoretical / Conceptual Topics

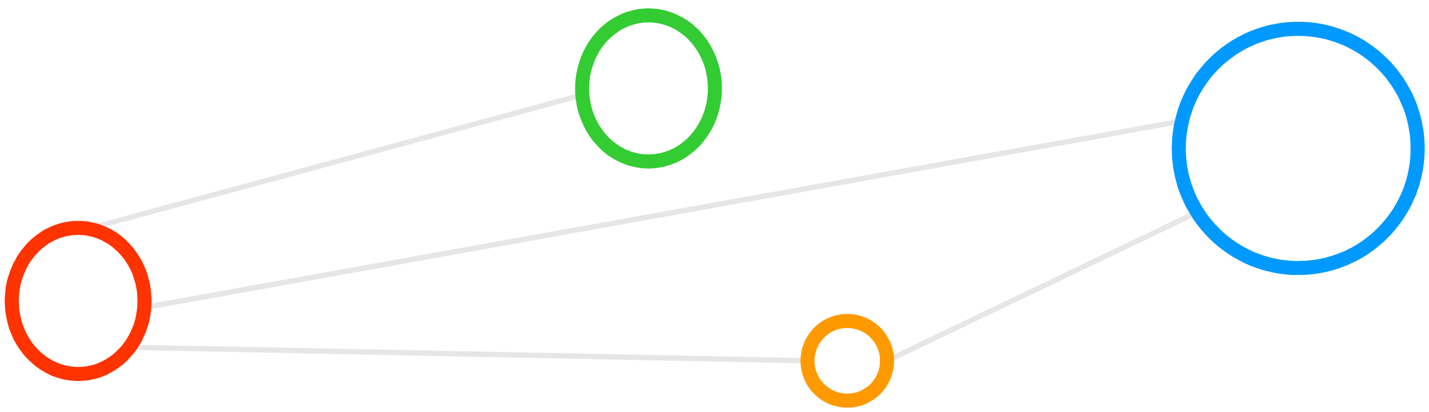
Outline

- Debugging & Profiling Techniques
  - Origin, Terminologies & Bug Prevention Approaches
  - Review Printf Debugging & Advanced Debugging Techniques & Tools
  - Terminologies, Performance Terms & Understanding Wall-clock time
  - Simple MPI Timing Approaches & MPI Profiling Interface
  - Selected Profiling Techniques & Tools using Profiling

- Performance Optimization Methods & Toolsets
  - Performance Measurements Metrics for MPI & OpenMP
  - Tracing Technique & Open Tracing Format
  - Simple Loops Constructs & Improving MPI Function Calls
  - Using the right MPI Collectives for better Performance
  - MPI & OpenMP Problem Patterns & I/O Hardware Dependencies

- **Promises from previous lecture(s):**
- ***Practical Lecture 0.2 & Lecture 1:*** **Lecture 9 will offer more insights into performance analysis systems with debugging, profiling, and HPC performance toolsets**
- ***Lecture 3 & 5:*** **Lecture 9 will give details on how to measure performance in parallel programms & and related tools using various applications**
- ***Lecture 4:*** **Lecture 9 on debugging, profiling & performance toolsets offers insights into performance analysis tools to understand MPI code better**
- ***Practical Lecture 5.1:*** **Lecture 9 will offer more examples where MPI non-blocking communication can influence the performance of parallel applications**
- ***Lecture 6:*** **Lecture 9 will provide a set of tools that can be used for monitoring, debugging, and performance analysis of MPI and OpenMP**

Debugging & Profiling Techniques



Origin & Terminologies

- Origin of term 'Debugging'
  - Mark II 'Supercomputer' @ Harvard University (~5 flop/s)
  - Incident: 'moth found trapped inside the computer, we are debugging…'
  - Coined the term: 'First actural case of bug being found'
- Examples
  - Memory problems: buffer overflow, wrong pointers, out of array bounds
  - Code complexity: one way of the program we haven't thought of…
  - Argument and (data) type mismatches
  - Unitialized variables
- Broad topic in parallel programming with many tools
  - E.g. HPC centers run intensive debugging training days…
  - Large collection of trainings in PRACE training repository

- **Debugging is a methodical process of finding and fixing flaws in software**

[1] Debuggers and Parallel Debugging, HPC Best practices

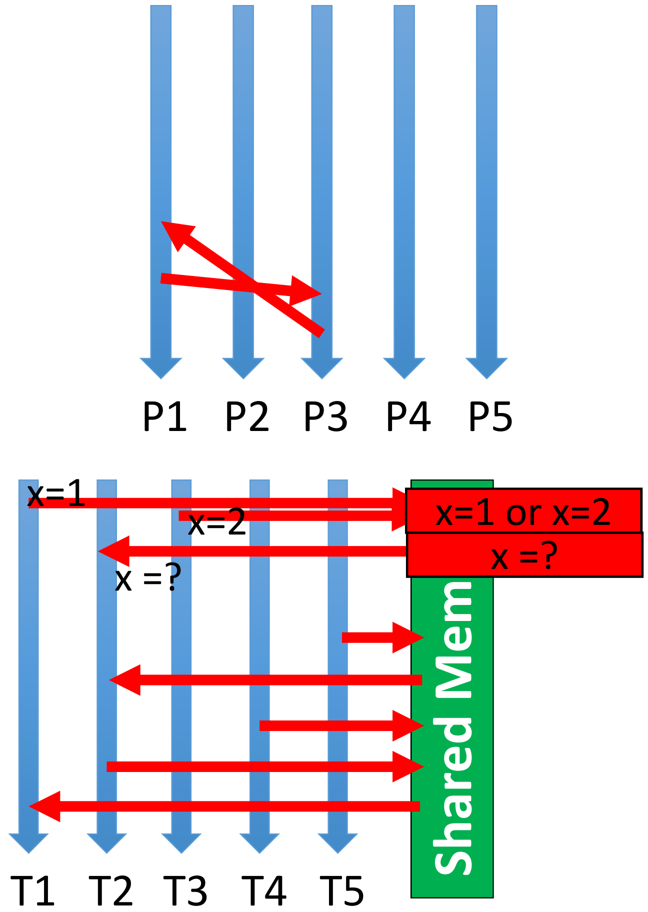
[2] PRACE Training

Terminologies – Deadlock & Race Condition

- Programming parallel algorithms
  - Challenges often rely in the complexity of ‘concurrency & computation‘

> ▪ **A deadlock is a situation wherein two ore more competing actions are each waiting for the other to finish, and thus neither ever is able to finish**

> ▪ **A race condition can be a flaw in a process whereby the output and/or result of the process is unexpectedly and critically dependent on the sequence or timing of other events**



Terminologies – Debugging, Profiling & Optimization

- Terminologies are related
  - Fine granular differentiation, but techniques (partly) overlap
- Debugging
  - Finding an error in the code and fixing it for correct program execution
  - E.g. correcting the usage of arrays in case of out of bounds problems
- Profiling (aka ‘aggregate statistics‘)
  - Understanding the program in terms of required execution time segments
  - E.g. which of the different functions in the program takes the most time?
- (Performance) Optimization
  - Should start when the ‘major flaws/bugs‘ in the software are solved
  - Tuning the program to enable a better performance (e.g. better speed-up)
  - E.g. finding ‘slow executions of codes patterns‘ with dedicated tools

Bug Prevention Approaches – Software Engineering

- Lessons learned from serial programming
  - Use same techniques as for parallel programming (e.g. MPI, OpenMP)
  - Apply software engineering principles (e.g. robustness, check error codes)
- Good parallel code readability
  - Meaningful variable and function names
  - Meaning and units of variables
  - Purpose and inputs/outputs descriptions of functions
- Version control
  - Take advantage of version control systems (e.g. cvs, svn, git, etc.)
- Well-defined code structures
  - Program towards different modules, enable re-usability of code elements

> - **Many parallel codes & libraries used in scientific computing don't implement the approaches**
> - **Bug prevention by applying software engineering concepts and having good code readability**

Bug Prevention Approaches – HPC Complexity

- Complex HPC environments
  - Fast 'code-change-compile-run' trials (from serial programming) infeasible
  - Scheduler is executing a script with a program, is it the right program?
  - Specifying a program as absolute path to executable can help
  - Using not the absolute path executes the first in $PATH variable
  - E.g. use `'which programname'` to check if it is really the right program
- Implement step-wise approach
  - Write 'serial code' that runs perfect, then use small number of processes
  - Next steps: fix communication/synchronization before going to large-scale
- Complex parallel programming with libraries
  - Large-scale parallel codes might depend on many existing libraries
  - Library version, handling, and implementation might vary over time

- **Bug prevention also means to check the HPC environments in which programs are executing**

Review ‘printf‘ Debugging Technique

- Use of simple yet effective ‘printf‘ statements while programming
  - Easy ‘instrumentation‘ of the code, no extra library, etc.
  - Take advantage of rank information what process is doing what work
  - Provides easy adjustable output, but order/process of outputs can vary

- Disadvantages
  - ‘Constant cycle programming‘ is time-consuming, error-prone, etc.
  - Not sure if bug found: add printf, compile, run, analyze output → again…
  - Extra printf code not helps for application logic → often remove after fix
  - Added extra code often helps only to ‘going after one bug‘, repeat per bug
  - Outputs maybe vary, e.g. in the timing when outputs are printed/process (cf. Lecture 3 and Lecture 6 printf statements in example code runs)

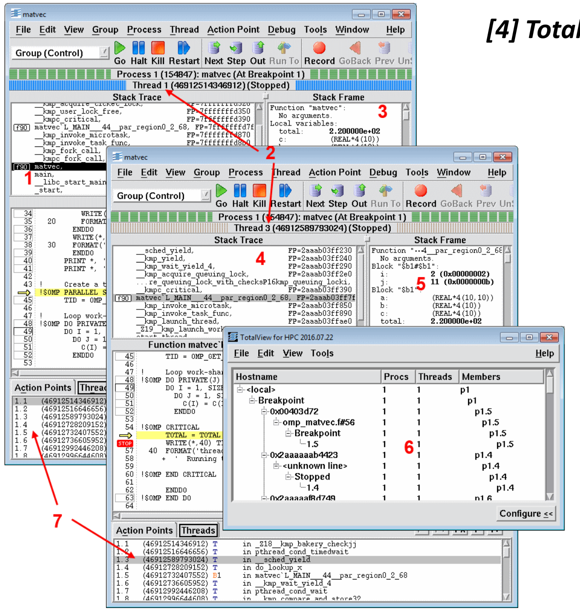
- **Printf debugging is not appropriate for the challenges of complex parallel program analysis**

Advanced Debugging Techniques

- Offer many advantages
  - Crash inspection
  - Function call stack overviews
  - Understanding logic via step-wise through code
  - Automated interruption and setting breakpoints
  - Insights into used variables (e.g. state, values, etc.)
- Added value: use of graphics
  - A wide variety of tools exists that visually support the debugging process
  - GUIs on local laptops is convenient, but limited for large-scale programs
  - HPC environments tend to be remote environments: GUIs might be slow (e.g. using SSH – X for X11 forwarding might be slow sometimes)
  - Good (fast) tools on command-line also exists, e.g. GDB
- Tools: Look all very similiar and often provide same advantages

Selected Debugging Tools

- Open Source Domain
  - GNU Debugger (GDB) – basic debugging together with DDD/Kdbg GUI
  - Marmot – MPI checker for parameters, standard conformance, deadlocks
  - Eclipse – Parallel Tools Platform integrated development environment
  - …
- Commercial tools
  - RogueWave TotalView –
    Graphical debugging tool supporting OpenMP/MPI
  - Alinea Distributed Debugging Tool (DDT) –
    Enabled highly scalable debugs
- Profiling tools or features supporting GPGPUs
  - Mostly very vendor-specific, e.g., NVIDIA toolsets



[4] TotalView Tool

- **The 'market of debugging tools' is dominated by strong commercial and expensive software**

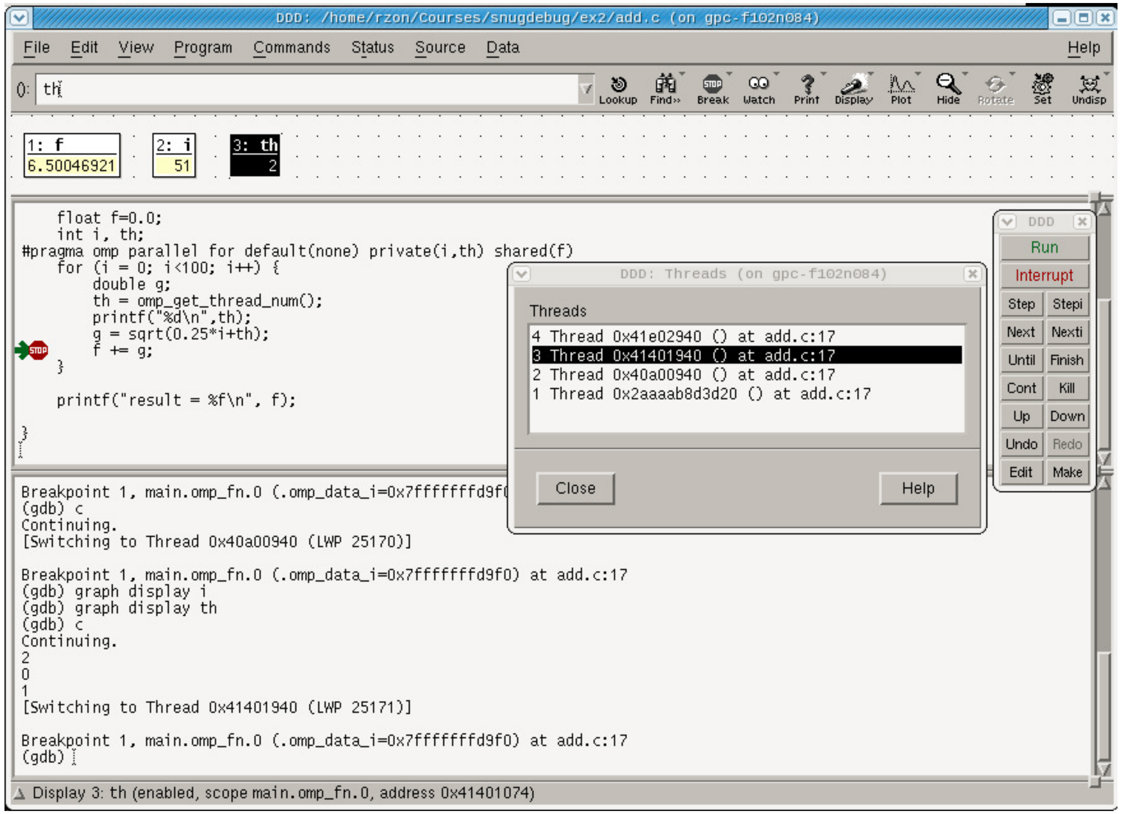
GDB Debugging Open Source Tool – Serial & Parallel

- GDB is essentially a tool for debugging serial programs
  - Serial debuggers can be used to debug parallel programs (‘basic features‘)
  - Works for low number of cores, no choice if using high number of cores
- Approach: Attach debugger to individual running MPI proccesses
  - Run `mpirun`, go to node,
    attach debugger to corresponding pids
- Approach: use mpirun to launch xterms with serial debuggers
  - Separate window for each MPI process,
    each running a serial debugger

```
/* source to obtain hostname and pid of correspondend process */
int i = 0;
Char hostname[256];
gethostname(hostname, sizeof(hostname));
printf("PID %d on %s ready for attach\n", getpid(), hostname);
fflush(stdout);
while (0==1) sleep(5);
...
/*commandline: attach gdb to a corresponding pid, here 4711 */
/home/user/gdb executable 4711
```

[3] OpenMPI Debugging

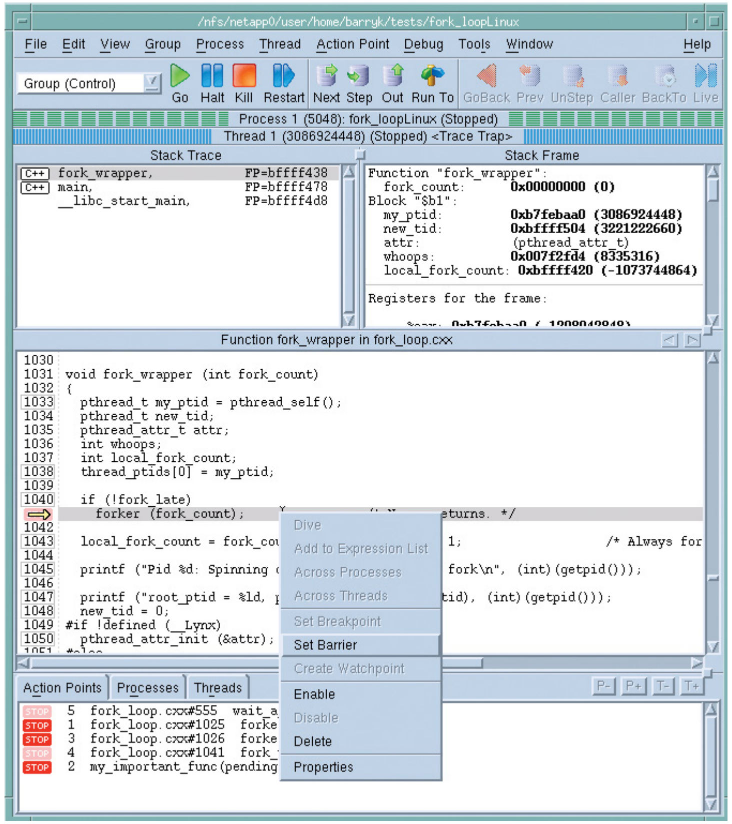
DDD Debugging Open Source GUI Tool – OpenMP Example



[1] Debuggers and Parallel Debugging

TotalView Commercial Debugging Tool – Capabilities

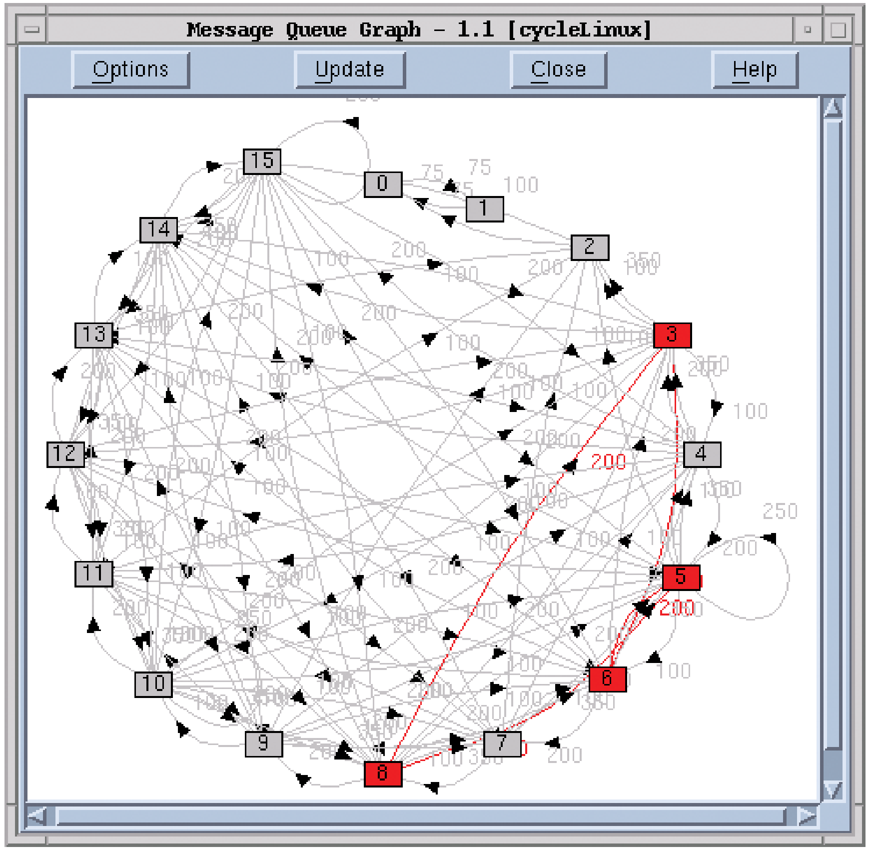
- Commercial Tool
  - Created and maintained by RogueWave Software
- Capabilities
  - Supports programming languages: C, C++, Fortran77, Fortran90
  - Offers a GUI for source code debugging and defect analysis
  - Enables deep views into program states and their variables
  - Provides control over processes and thread execution
- Parallel Debugging Support
  - Multi-threaded debugging
  - Distributed debugging



[4] TotalView Tool

TotalView Commercial Debugging Tool – Graphs

- E.g. identfiying cycles that may prevent the program to finish

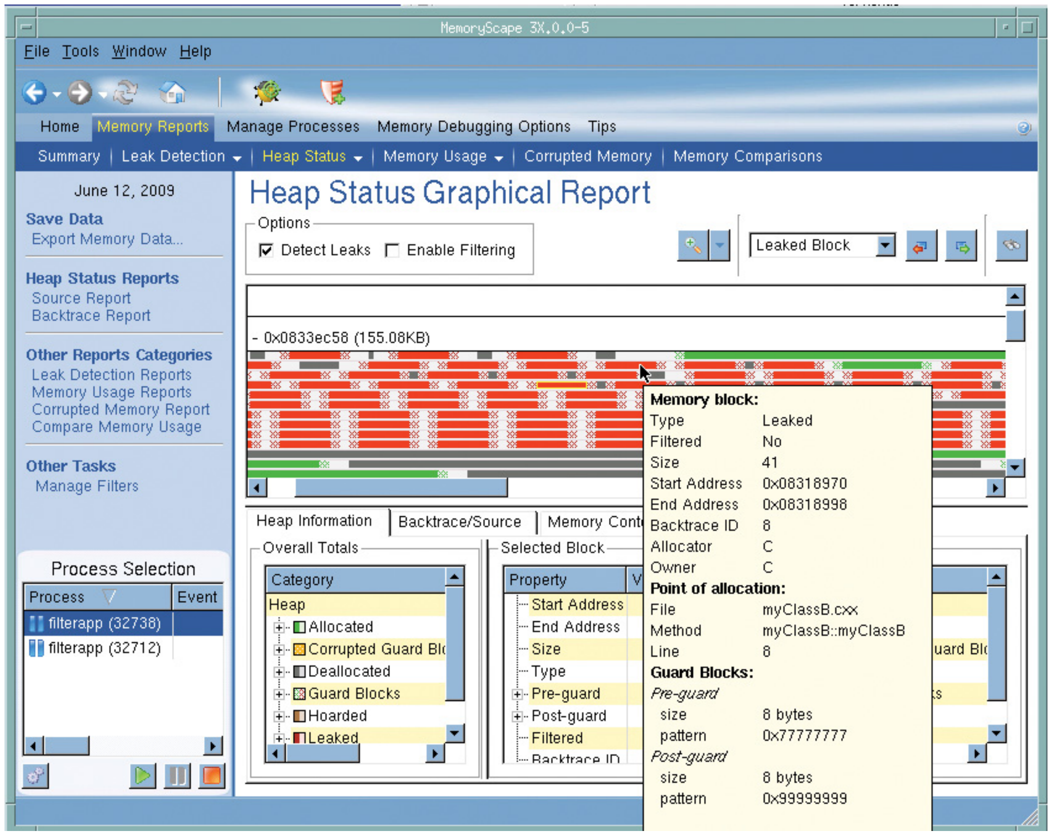


[4] TotalView Tool

TotalView Commercial Debugging Tool – MemoryScape

- E.g. understanding memory problems, segmentation faults, etc.

[4] TotalView Tool



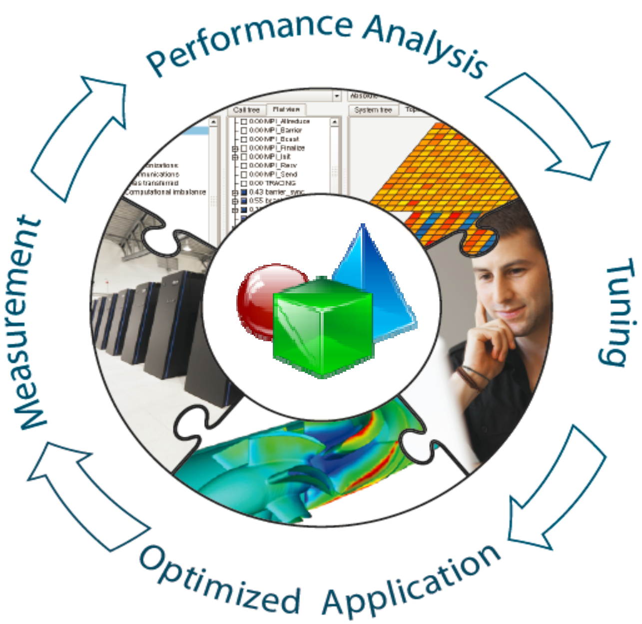
Terminologies – Debugging, Profiling & Optimization

- Terminologies are related
  - Fine granular differentiation, but techniques (partly) overlap
- Debugging
  - Finding an error in the code and fixing it for correct program execution
  - E.g. correcting the usage of arrays in case of out of bounds problems
- Profiling (aka 'aggregate statistics')
  - Understanding the program in terms of required execution time segments
  - E.g. which of the different functions in the program takes the most time?
- (Performance) Optimization
  - Should start when the 'major flaws/bugs' in the software are solved
  - Tuning the program to enable a better performance (e.g. better speed-up)
  - E.g. finding 'slow executions of codes patterns' with dedicated tools

Understanding your Program

- Performance Analysis & Tuning Tools
  - Enable optimized applications (after iterations)
  - Require concrete measure metrics
- Measurement metrics
  - Generic metrics
  - MPI / OpenMP specific metrics
- Scalability metrics
  - Strong / weak scaling
- HPC centers perform scalability workshops
  - Getting code scalable together with experts

modified from [7] Scalasca Flyer



- A scalable parallel code is a code that keeps a good performance ratio / core by increasing cores
- Getting a parallel code scalable is a 'process cycle' that include performance analysis & tuning
- Large-scale parallel code needs not only good optimization techniques (also fault tolerance, etc.)

Time-To-Solution – Parallelization with Serial Elements

- S = Algorithmic limitations
  - E.g. elements need to be simply executed one after another
- S = Bottlenecks with shared resources
  - E.g. shared paths to memory in multicore chips or I/O devices
- S = Startup overhead
  - E.g. starting a parallel program takes time (often initialization phases)
  - Note: if parallel application is short-running, startup has strong impact
- S = Communication
  - E.g. not always fully concurrent communication between different parts of a parallel system

- **Amount of work/overall problem size:**

$$s + p = 1$$

- **s = serial (nonparallelizable part)**
- **p = parallelizable part**

Performance Definitions & Time Measurements – Revisited (cf. Lecture 3)

- Performance here means 'work (s+p) over time (T_f^S)'
- P_f^S = serial performance for fixed problem with $T_f^S = s + p$

$$P_f^S = \frac{s+p}{T_f^S} = 1$$

- P_f^P = parallel performance for fixed problem with $T_f^P = s + p/N$

$$P_f^P = \frac{s+p}{T_f^P(N)} = \frac{1}{s+\frac{1-s}{N}}$$

Terminologies – Wall-Clock Time (aka Walltime)

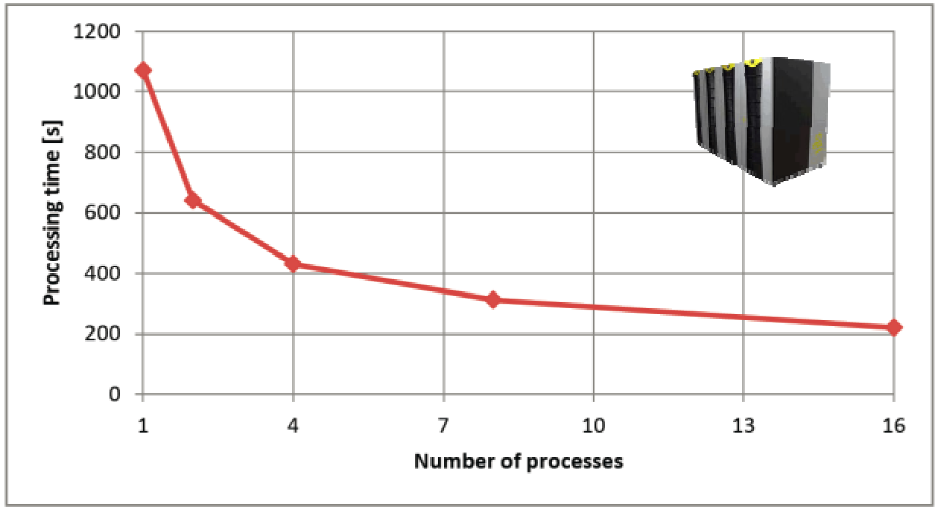
- ‘Benchmarking‘ a parallel program requires a dedicated term
  - Most sensible time measure is called wall-clock time (i.e. elapsed time)
  - Using ‘only CPU time‘ is prone to misinterpretation for many reasons…
  - E.g. program runtimes with ‘contributions‘ from I/O, other processes, etc.
- Relationship to ‘cost models‘ – why wall-clock time is important
  - Goal: discourage the use of too many workers (with less performance)
  - HPC centers ‘charge‘ for compute time in units of CPU wall-clock hours
  - Real money is rarely used - scientists get a ‘grant for N wall-clock hours‘
  - An N-CPU job running for a time T_w will be charged proportional to N T_w
  - Approach: minimizing walltime (i.e. ‘time-to-solution‘) saves ‘costs‘

- **Wall-clock time is the actual time taken to complete a program and the sum of three different terms: CPU time, I/O time and the communication channel delay (e.g. message passing)**

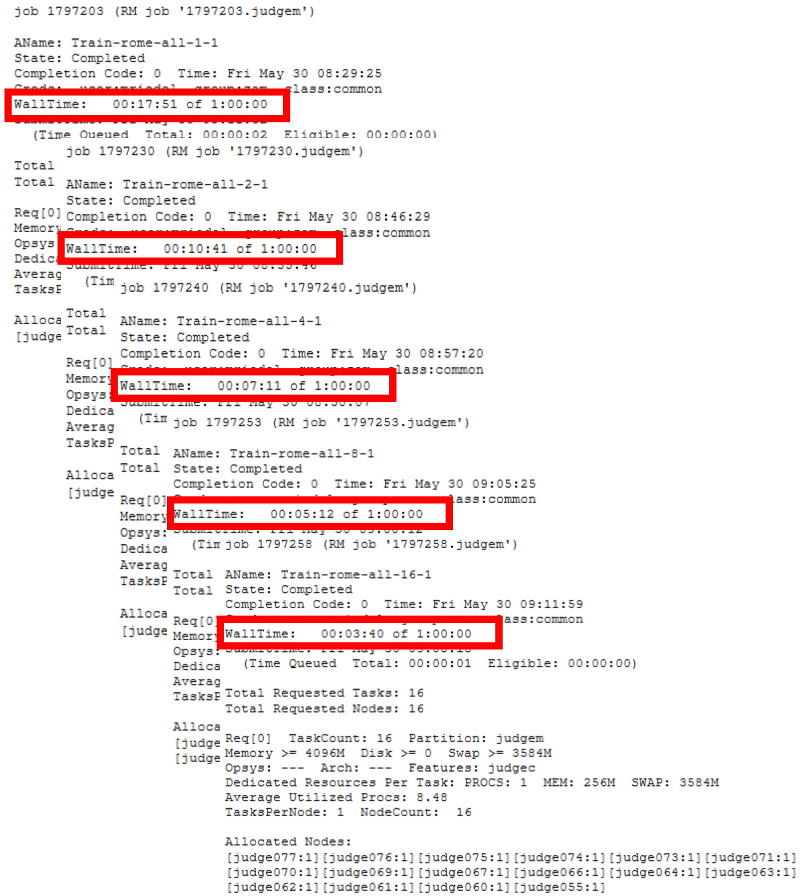
modified from [8] Wikipedia on ‘wall-clock time’ [6] Introduction to High Performance Computing for Scientists and Engineers

Simple Example: Use of Wall-clock time to show Speed-up

- E.g. measure walltimes for a whole parallel data analytics MPI application
  - Increasing number of cores leads to lower 'time-to-solution'



[9] B2SHARE, piSVM Analytics runtimes


job 1797203 (RM job '1797203.judgem')

AName: Train-rome-all-1-1
State: Completed
Completion Code: 0 Time: Fri May 30 08:29:25
WallTime: 00:17:51 of 1:00:00

job 1797230 (RM job '1797230.judgem')

AName: Train-rome-all-2-1
State: Completed
Completion Code: 0 Time: Fri May 30 08:46:29
WallTime: 00:10:41 of 1:00:00

job 1797240 (RM job '1797240.judgem')

AName: Train-rome-all-4-1
State: Completed
Completion Code: 0 Time: Fri May 30 08:57:20
WallTime: 00:07:11 of 1:00:00

job 1797253 (RM job '1797253.judgem')

AName: Train-rome-all-8-1
State: Completed
Completion Code: 0 Time: Fri May 30 09:05:25
WallTime: 00:05:12 of 1:00:00

job 1797258 (RM job '1797258.judgem')

AName: Train-rome-all-16-1
State: Completed
Completion Code: 0 Time: Fri May 30 09:11:59
WallTime: 00:03:40 of 1:00:00
(Time Queued Total: 00:00:01 Eligible: 00:00:00)

Total Requested Tasks: 16
Total Requested Nodes: 16

Req[0] TaskCount: 16 Partition: judgem
Memory >= 4096M Disk >= 0 Swap >= 3584M
Opsys: --- Arch: --- Features: judgec
Dedicated Resources Per Task: PROCS: 1 MEM: 256M SWAP: 3584M
Average Utilized Procs: 8.48
TasksPerNode: 1 NodeCount: 16

Allocated Nodes:
[judge077:1][judge076:1][judge075:1][judge074:1][judge073:1][judge071:1]
[judge070:1][judge069:1][judge067:1][judge066:1][judge064:1][judge063:1]
[judge062:1][judge061:1][judge060:1][judge055:1]

Simple MPI Timing Approaches

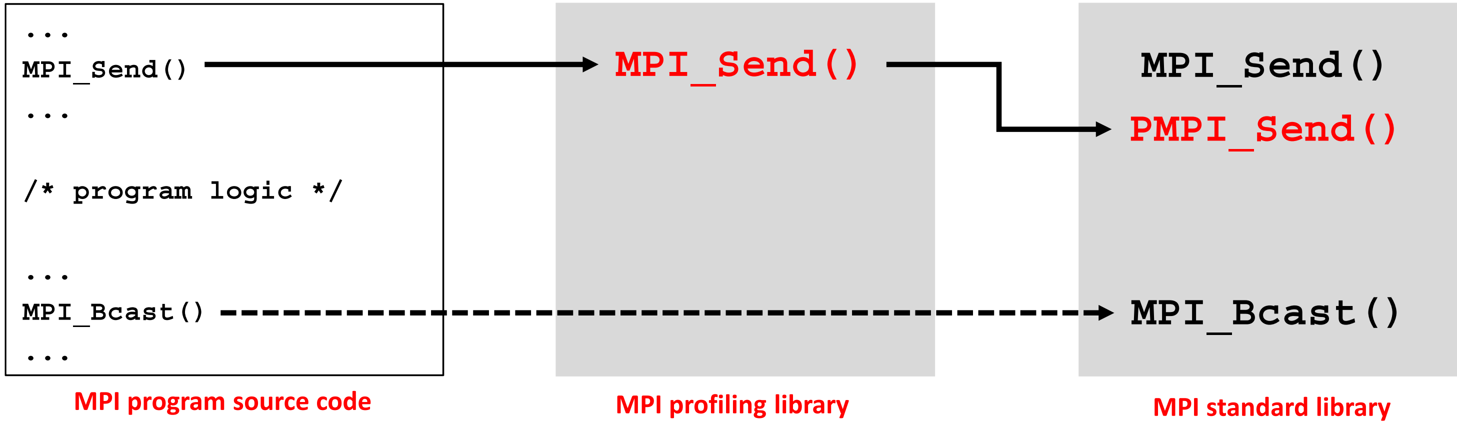
- Manual 'instrumentation' of a MPI program code segment
  - E.g. elapsed wall-clock time between two selected points in a program
  - Elapsed time can be computed with `MPI_Wtime()`
  - Useful in conjunction with 'printf' statements and calling it more than once
  - Simple, but manual work is time-consuming and later often removed

```
...
double time1, time2;
time1 = MPI_Wtime();
...
/* MPI program segment important to understand in terms of elapsed time */
...
time2 = MPI_Wtime();
Printf("elapsed time of program segment is %d\n", time2 - time1);
...
```

- **The function MPI_Wtime() provides the elapsed wall-clock time of a parallel MPI program**

MPI Profiling Interface

- Usage
  - Perform manual replacement of MPI routines at link time with PMPI
  - Augment wrapper routine with statements, e.g. 'function call counters'
  - e.g. performance analysis tools take advantage of PMPI interface



- **The MPI profiling interface PMPI enables flexible writing of MPI functions wrapper routines**
- **Wrappers named as standard MPI_xyz routines internally call MPI standard routines via PMPI**
- **MPI offers an alias PMPI_xyz for each standard MPI routine, e.g. PMPI_Send() & MPI_Send()**

MPI Profiling Interface – Simple Usage Example

- Usage
  - E.g. understanding how often a specific MPI function was called
  - Link the profiling library, e.g. `cc – o prog.exe prog.c -lpmpi -lmpi`

```
...
static int numberofsend = 0;
...

/* implement own wrapper function for MPI call and edit as needed */
int MPI_Send( void *start, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm) {

   /* another call - increase counter! */
   numberofsend++;
   return PMPI_send( start, count, datatype, dest, tag, comm);
}
...

/* MPI program segment with real program logic */
int program {
  ...
  MPI_Send( ... );
  ...
}
```

'simplified demo code'

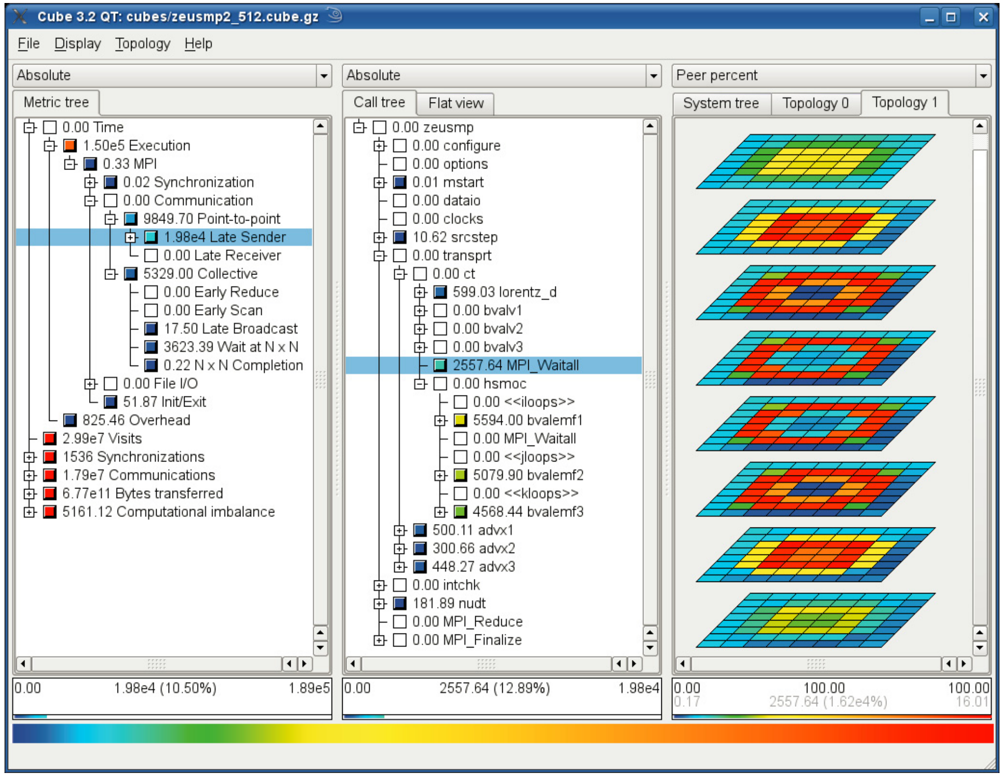
Selected Profiling Tools

- Open Source Domain
  - Valgrind – instrumentation framework with tools to profile memory usage
  - Vampir – Trace-based profiling offering a good ‘timeline view‘ of programs
  - Scalasca – Trace-based profiling and performance analysis (with patterns)
  - …
- Commercial tools
  - Intel® VTune™ Amplifier XE – Graphical profiler tool for parallel programs
  - …
- Profiling tools or features supporting GPGPUs
  - Mostly very vendor-specific, e.g., NVIDIA toolsets

- **There is an overlap between tools used in parallel debguggin, profiling & performance analysis**
- **Parallel performance analysis tools partly take advantage of profiling techniques & interfaces**
- **Tracing collects information about the program for post analysis – profiling aggregates statistics**

Scalasca Toolset Example – Analysis Report Examiner CUBE

[7] SCALASCA Performance Tool



- **A powerful analysis report examiner enables to determine (a) which performance problem is faced, (b) where in the program, and (c) which processes of the HPC machine are affected**

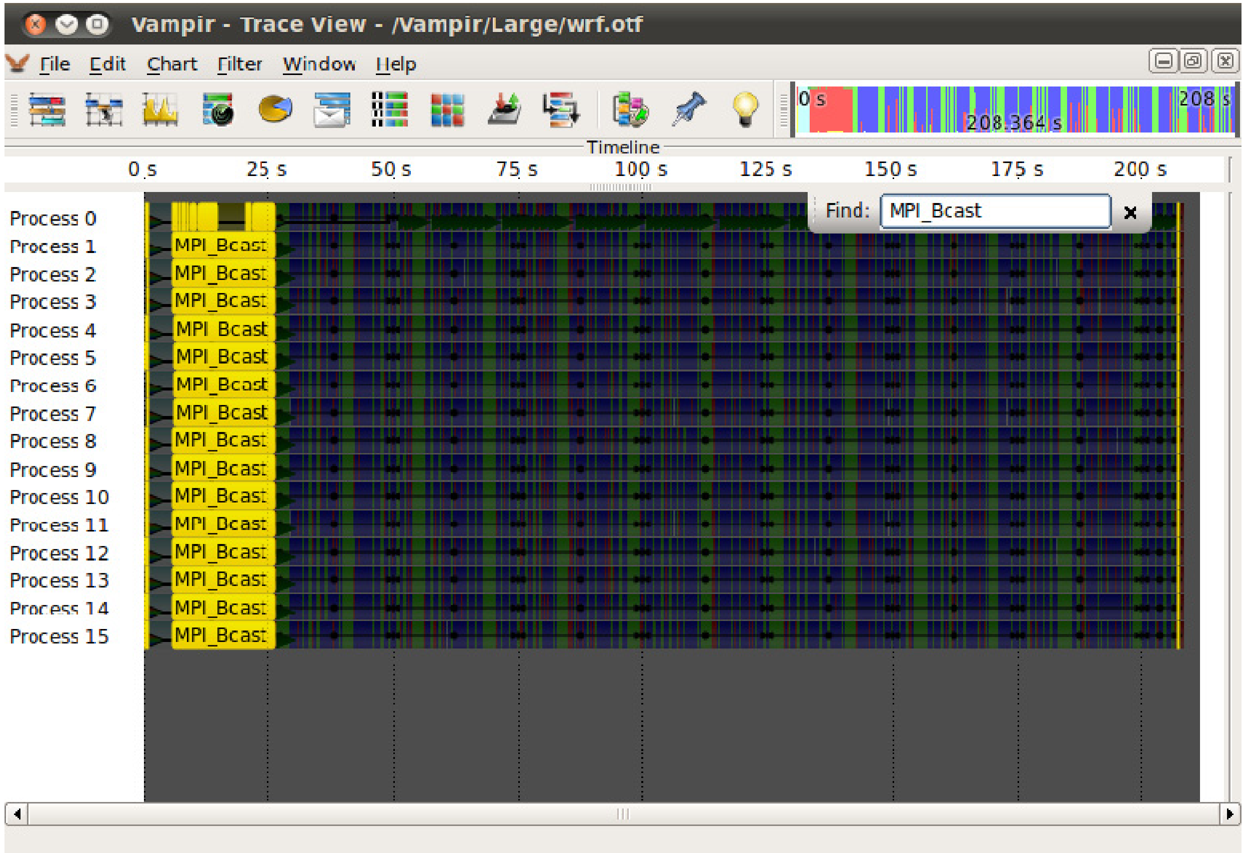
Valgrind Open Source Profiling Tool – Capabilities

- Valgrind is a whole open source 'instrumentation framework'
  - Enables building of dynamic analysis tools
  - Flexible system for profiling Linux executables (including MPI)
- Selected toolset
  - Memcheck/Addrcheck: Detection of memory-management problems
  - Cachegrind: Cache profiler - detailed simulation of the I1, D1 and L2 caches is provided to pinpoint the sources of cache misses
  - Callgrind: adds call graph tracing to cachegrind - used to get call counts and inclusive cost for each call happening in a program
  - Massif: Memory consumption profiling
  - Helgrind: Identify race conditions in multithreaded programs

```
export LD_PRELOAD = $VALGRIND_MPILIB
...
mpiexec -n <numbertasks> valgrind <valgrind switches> program.exe
```

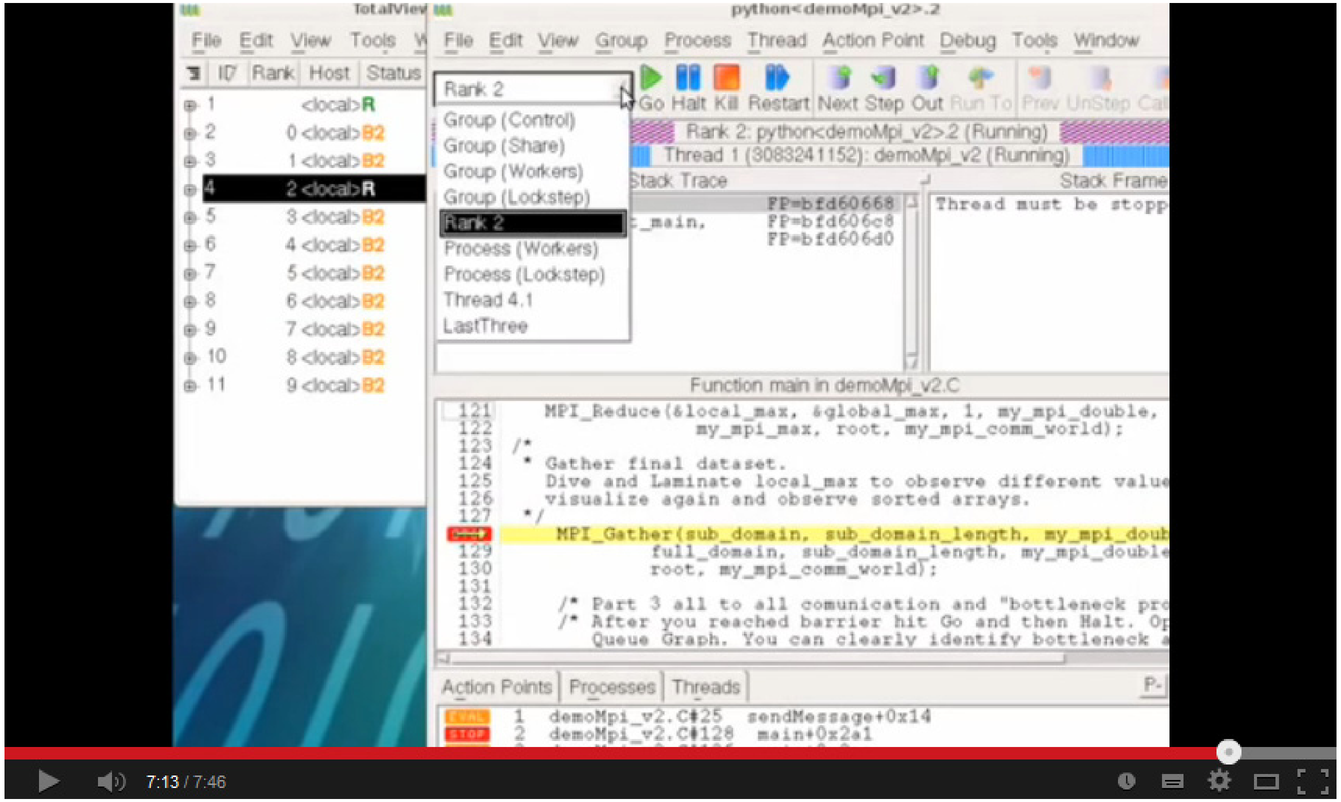
[10] Valgrind Webpage

VAMPIR Open Source Profiling Tool - Example



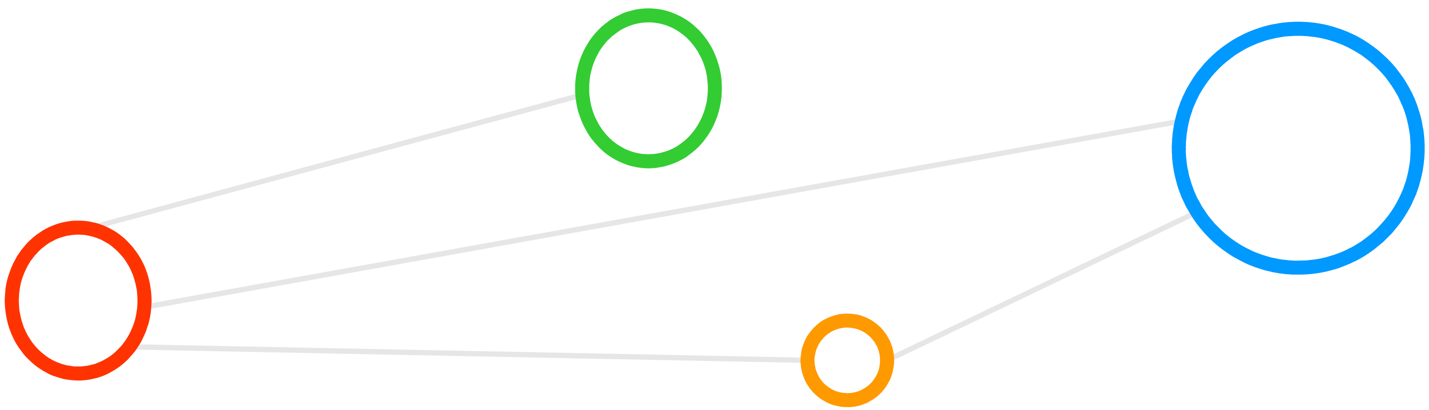
[11] VAMPIR Performance Tool

[Video] Debugging a MPI Program



[5] YouTube Video, MPI Debugging with the TotalView debugger

Performance Optimization Methods & Toolsets



Terminologies – Debugging, Profiling & Optimization

- Terminologies are related
  - Fine granular differentiation, but techniques (partly) overlap
- Debugging
  - Finding an error in the code and fixing it for correct program execution
  - E.g. correcting the usage of arrays in case of out of bounds problems
- Profiling (aka ‘aggregate statistics‘)
  - Understanding the program in terms of required execution time segments
  - E.g. which of the different functions in the program takes the most time?
- (Performance) Optimization
  - Should start when the ‘major flaws/bugs‘ in the software are solved
  - Tuning the program to enable a better performance (e.g. better speed-up)
  - E.g. finding ‘slow executions of codes patterns‘ with dedicated tools

HPC System Software Environment – Revisited (cf. Practical Lecture 0.2)

- Operating System
  - Former times often 'proprietary OS', nowadays often (reduced) 'Linux'
- Scheduling Systems
  - Manage concurrent access of users on Supercomputers
  - Different scheduling algorithms can be used with different 'batch queues'
  - Example: SLURM @ JÖTUNN Cluster, LoadLeveler @ JUQUEEN, etc.
- Monitoring Systems
  - Monitor and test status of the system ('system health checks/heartbeat')
  - Enables view of usage of system per node/rack ('system load')
  - Examples: LLView, INCA, Ganglia @ JOTUNN Cluster, etc.
- Performance Analysis Systems

  focus in this lecture

  - Measure performance of an application and recommend improvements (.e.g Scalasca, Vampir, etc.)

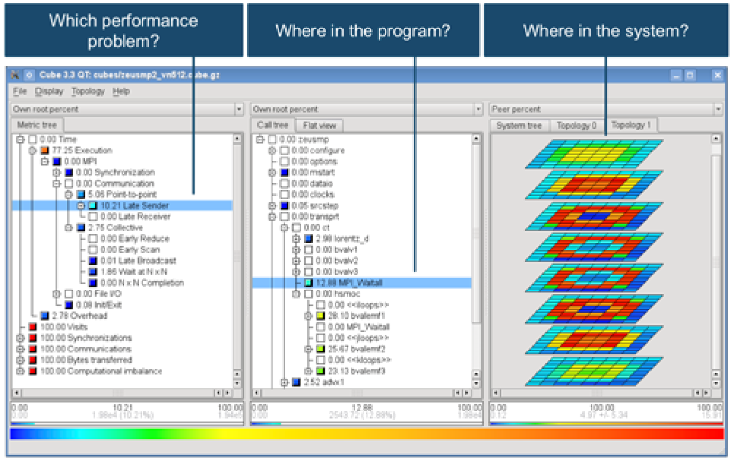
- **HPC systems and supercomputers typically provide a software environment that support the processing of parallel and scalable applications**

- **Monitoring systems offer a comprehensive view of the current status of a HPC system or supercomputer**

- **Scheduling systems enable a method by which user processes are given access to processors**

Performance Analysis is a Key Field in HPC – Revisited (cf. Lecture 3)

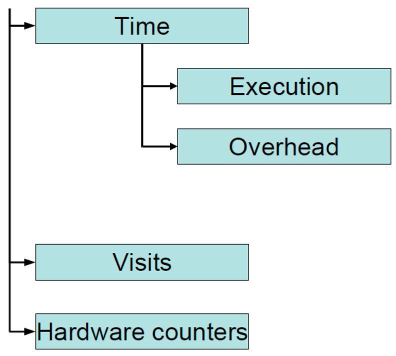
- Analysis is typically performed using (automated) software tools
  - Measure and analyze the runtime behaviour of parallel programs
  - Identifies potential performance bottlenecks
  - Offer performance optimization hints and views of the location in time
  - Guides exploring causes of bottlenecks in communication/synchronization



[7] SCALASCA Performance Tool

Generic Measurement Metrics

- Time – Total CPU allocation time
  - Execution time w/o overhead
  - Overhead time spent in tasks related to the measurement itself
- Visits
  - Number of times a function / region was executed
- Hardware counters
  - Aggregated counter values for each function / region

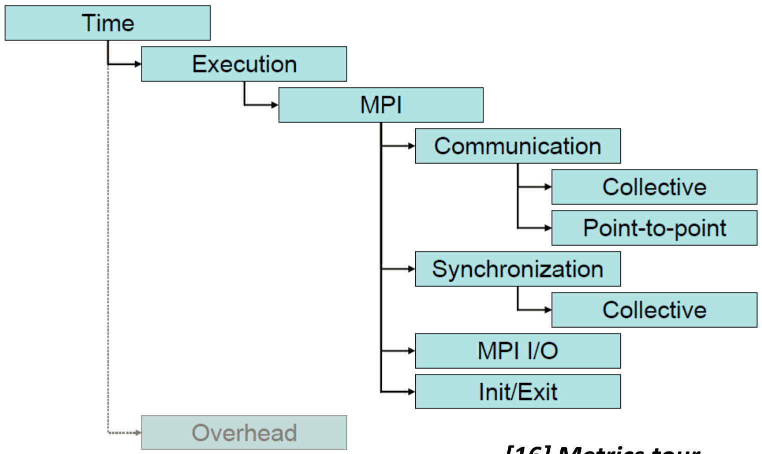


[16] Metrics tour

- **Metrics are required in order to have a clear understanding of what is measured in analysis steps**
- **Generic metrics are CPU allocation time (execution and overhead), visits, and hardware counters**

Metrics for Parallel Programs using MPI

- Time – Execution - MPI –
  Time spent in (instrumented) MPI functions
  - Communication - Time spent in MPI communication calls (collective and point-to-point)
  - Synchronization - Time spent in calls to `MPI_Barrier()`
  - MPI I/O - Time spent in MPI I/O functions
  - Init/Exit - Time spent in `MPI_Init()` and `MPI_Finalize()`

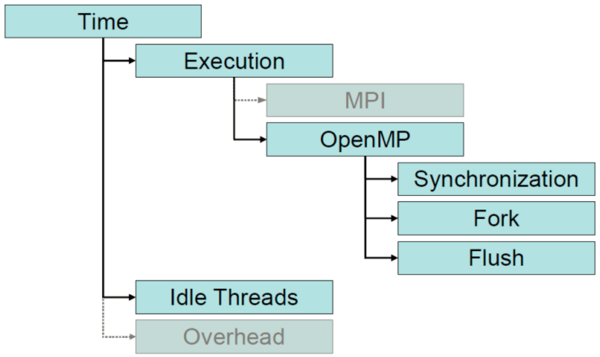


[16] Metrics tour

- **Metrics for parallel programs using MPI are based on time as part of the program execution time**
- **MPI metrics are Communication (collective/point-to-point), Synchronization, MPI I/O, and Init/Exit**

Metrics for Parallel Programs using OpenMP

- Time – Execution - OpenMP –
  Time spent for OpenMP-related tasks
  - Synchronization - Time spent for synchronizing OpenMP threads
  - Fork - Time spent by master thread to create thread teams
  - Flush - Time spent in OpenMP flush directives
- Idle Threads
  - Time spent idle on CPUs reserved for worker threads

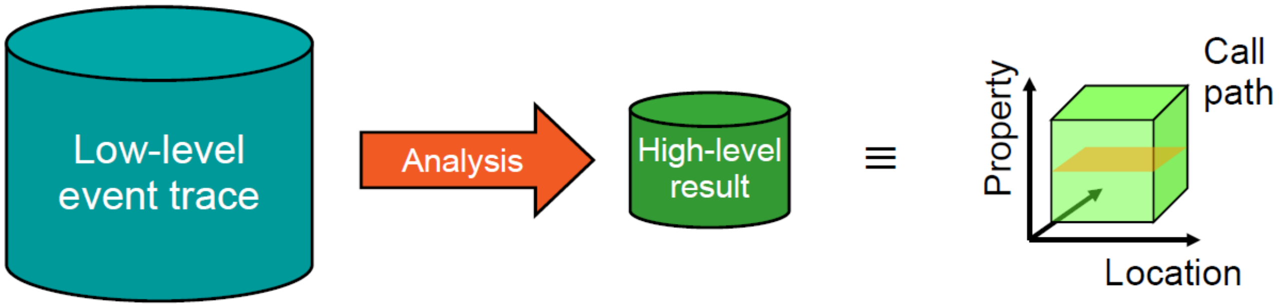


[16] Metrics tour

- **Metrics for parallel programs using MPI are based on time as part of the program execution time**
- **MPI metrics are Communication (collective/point-to-point), Synchronization, MPI I/O, and Init/Exit**

Tracing Technique – Need for Automation

- Manual/visual trace analysis for whole parallel codes is inefficient
  - Measuring metrics with e.g. `MPI_Wtime()` is error-prone & time consuming
- Automatic trace analysis process
  - Enables automatic search for patterns of inefficient behaviour
  - Quicker than manual/visual trace analysis and feasible (e.g. large-scale)
  - Guaranteed to cover the entire event trace
  - Classification of behaviour & quantification of significance

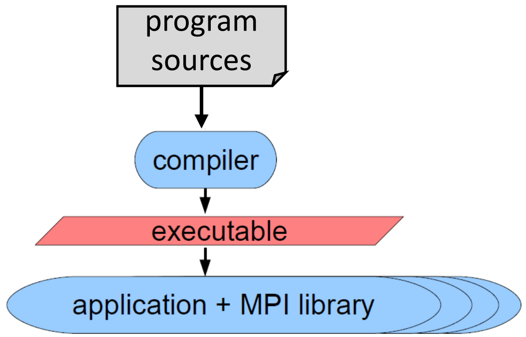


[7] SCALASCA Performance Tool

- **Tracing collects information about the program for post analysis – profiling aggregates statistics**

Tracing Technique – Functionality (1)

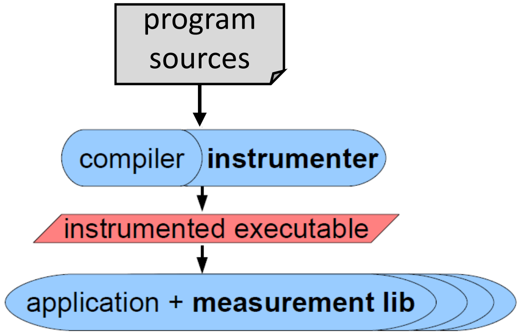
- Step 0 - Based on general MPI Program Build & Run process
  - Application code compiled & linked into executable (e.g. using `mpicc`)
  - Launched with script (using e.g. `mpiexec`)
  - Application processes interact via MPI library



[7] SCALASCA Performance Tool

Tracing Technique – Functionality (2)

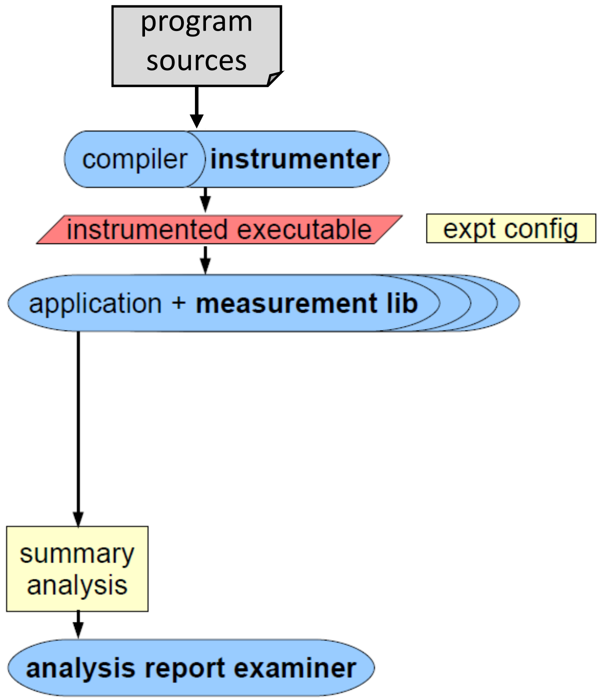
- Step 1 – Application Instrumentation
  - Run automatic code instrumenter (also manual elements possible)
  - Program sources are automatically processed to add instrumentation to the executable
  - Measurement library is added into application executable
  - Exploits MPI standard profiling interface (PMPI) to acquire MPI events



[7] SCALASCA Performance Tool

Tracing Technique – Functionality (3)

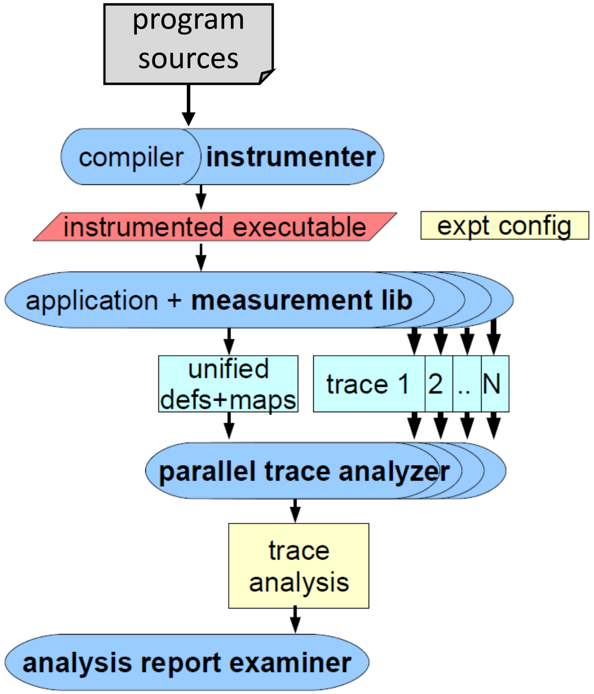
- Step 2 a) – Measurement
  runtime summarization & analysis
  - Measurement library
    manages threads & events
    (e.g. enter/exit a function)
    produced by instrumentation
  - Measurements summarized
    by thread & call-path during execution
  - Summary analysis report unified &
    collated at finalization
  - Investigation of summary analysis
    using a analysis report examiner tool



[7] SCALASCA Performance Tool

Tracing Technique – Functionality (4)

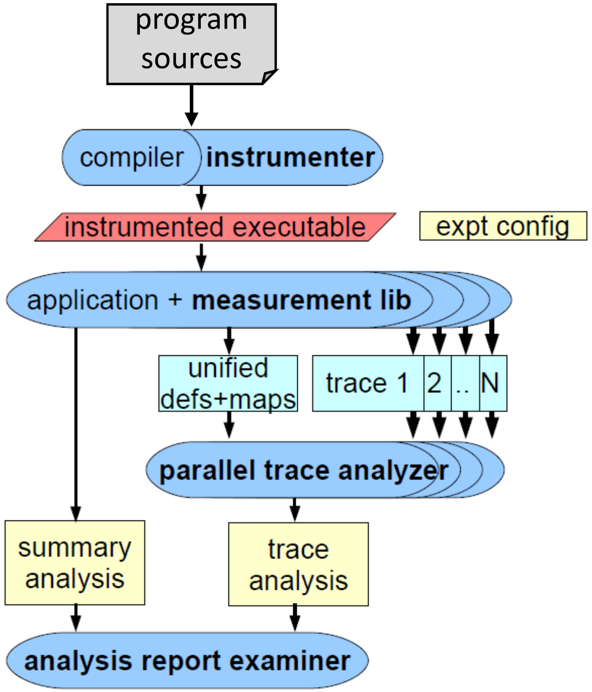
- Step 2 b) – Measurement event tracing & analysis
  - During measurement time-stamped events are buffered for each thread
  - Flushed to files along with unified definitions & maps at program finalization
  - Follow-up analysis replays events and produces extended analysis report
  - Investigation of trace analysis using a analysis report examiner tool



[7] SCALASCA Performance Tool

Tracing Technique – Summary

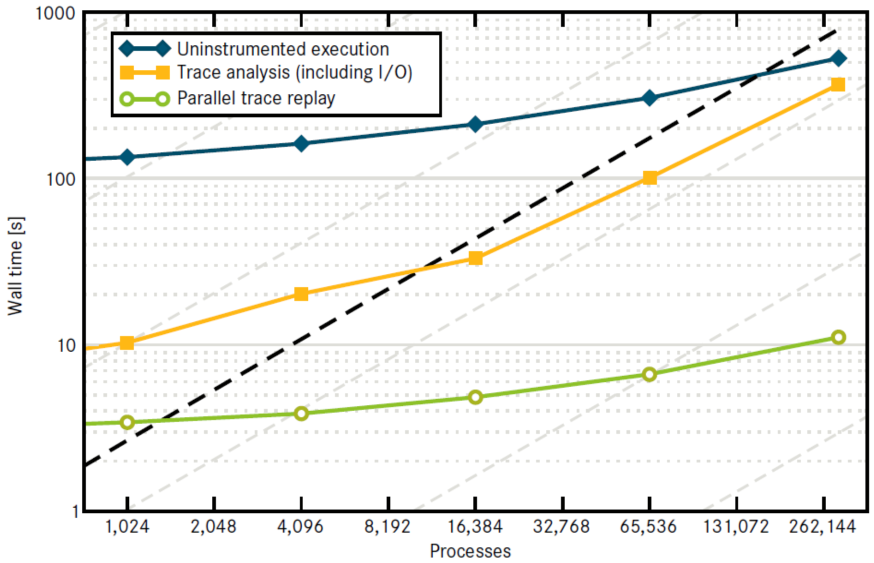
- Automatic/manual code instrumenter is used to enable runtime measurement and event tracing (use of MPI profiling interface)
- Tracing requires a specific measurement library for runtime summary & event tracing
(basic MPI techniques are limited)
- Trace architecture enables serial and parallel event trace analysis
- Use of analysis report examiner tools for interactive exploration of measured execution performance properties & metrics



[7] SCALASCA Performance Tool

Tracing Technique – Impacts on Scalability

- Weak Scaling Example
  - Parallel application Sweep3D benchmark code (fixed problem size/process)
  - Scalasca trace analysis completed with up to 294,912 processes
  - Parallel trace replay analysis exploits memory & processors for scalability



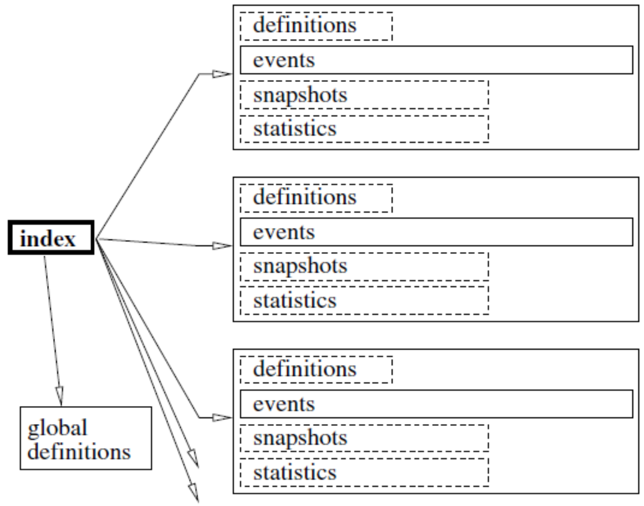
- **Using the tracing technique has an impact on the runtime and scalability of codes (e.g. I/O & # files)**
- **Replay and analysis of original parallel codes requires parallel tools & techniques to be scalable too**

[7] Scalasca Flyer

Open Trace Format (OFT)

- Performance Analysis & Optimization is active research field
  - Result is a wide variety of partly different tools with many different formats
- Inconvenience when using different performance analysis tools
  - Epilog (Kojak/Scalasca)
  - Paje format (Paje)
  - STF (Intel Trace Analyzer)
  - Tau trace format (Tau)
  - Slog2 (Jumpshot)
  - Paraver format (Paraver)
- Different OTF versions
  - OTF2 is successor format to OTF and Epilog formats
  - Major re-design and new implementation

[17] Open Trace Format



- **The open trace format is a standardized data structure and API specification for tracing data**

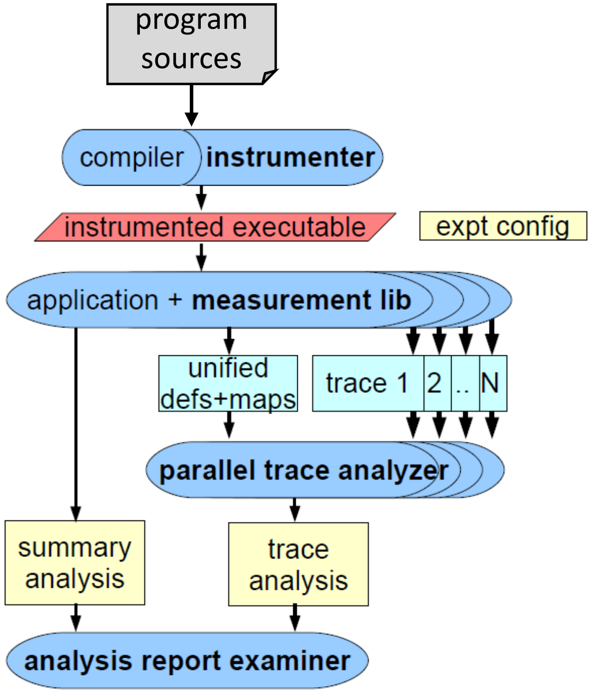
Selected Performance Analysis Tools

- Open Source Domain
  - Valgrind – instrumentation framework with tools to profile memory usage
  - Vampir – Trace-based profiling offering a good 'timeline view' of programs
  - Scalasca – Trace-based profiling and performance analysis (with patterns)
  - Periscope – Scalable automatic performance analysis tool (in development)
  - PAPI – Interfacing to hardware performance counters
  - TAU – Integrated parallel performance system
  - Score-P - Scalable performance measurement infrastructure
- Commercial tools
  - Intel® VTune™ Amplifier XE – Graphical profiler tool for parallel programs
  - Intel Tracing Tools (Trace Collector, Trace Analyzer, Message Checker, …)
  - SGI ProPack (suite of performance optimization libraries & tools)

- **There is an overlap between tools used in parallel debgugging, profiling & performance analysis**
- **Parallel performance analysis tools partly take advantage of profiling techniques & interfaces**

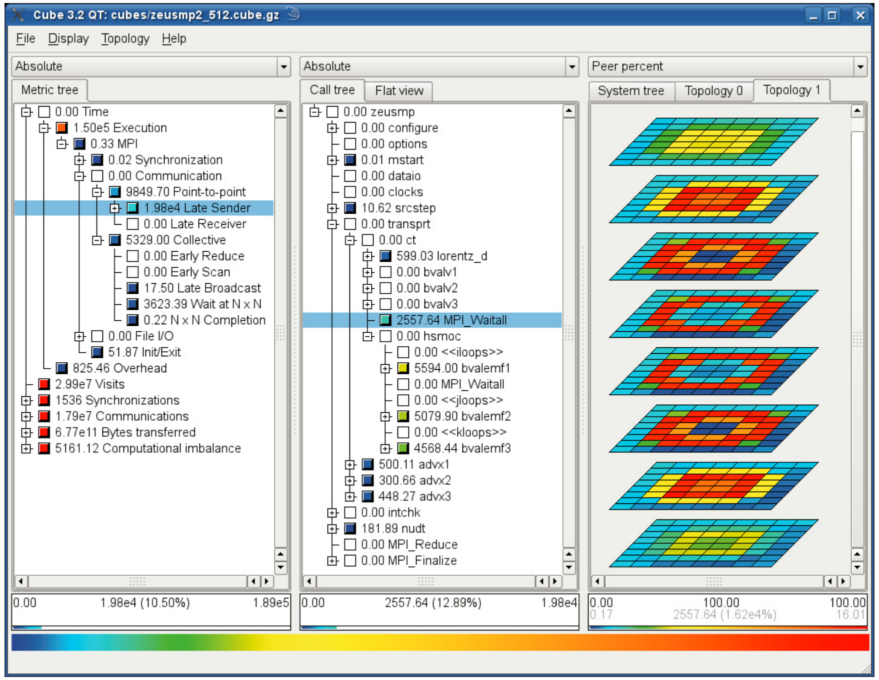
Scalasca Toolset Example

- Based on tracing technique
  - Three key tools for different elements in tracing & analysis steps
- Compiler instrumenter
  - Scalasca SKIN
- Measurement collector & analyzer
  - Scalasca SCANTBD
- Analysis report examiner
  - Scalasca CUBE

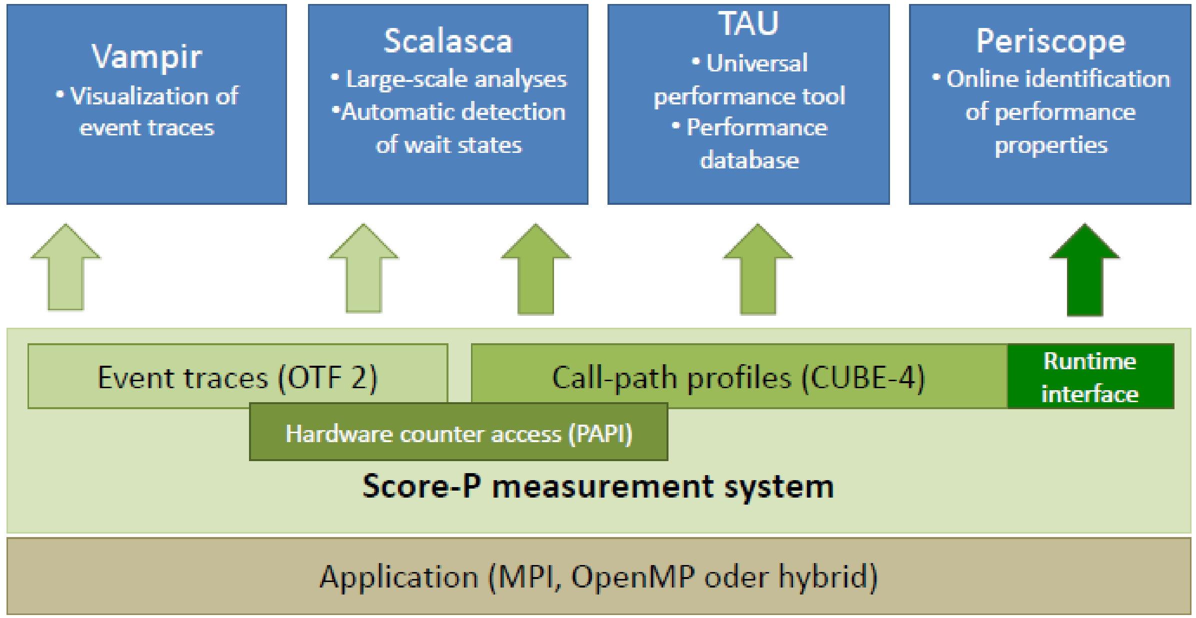


[7] SCALASCA Performance Tool

Examples: Scalasca Analysis Report Examiner CUBE & Score-P



[7] SCALASCA Performance Tool



[18] Future of OTF

- **A powerful analysis report examiner such as Scalasca CUBE enables to determine (a) which performance problem is faced, (b) where in the program, and (c) which processes of the HPC machine are affected**
- **Score-P Performance Measurement Infrastructure works with a variety of performance analysis tools such as Vampir, Scalasca, Tau, and Periscope**

Optimizing Simple Loop Constructs in MPI & OpenMP

- Values that depend on each other in loops
  - Example: choose R according to N so overall execution time stays constant

```
do j=1,R
  do i=1,N
    A(i) = B(i) + C(i) * D(i)   ! 3 loads, 1 store
  enddo
  if(A(2).lt.0) call dummy(A,B,C,D) ! prevent loop interchange
enddo
```

[6] Introduction to High Performance Computing for Scientists and Engineers

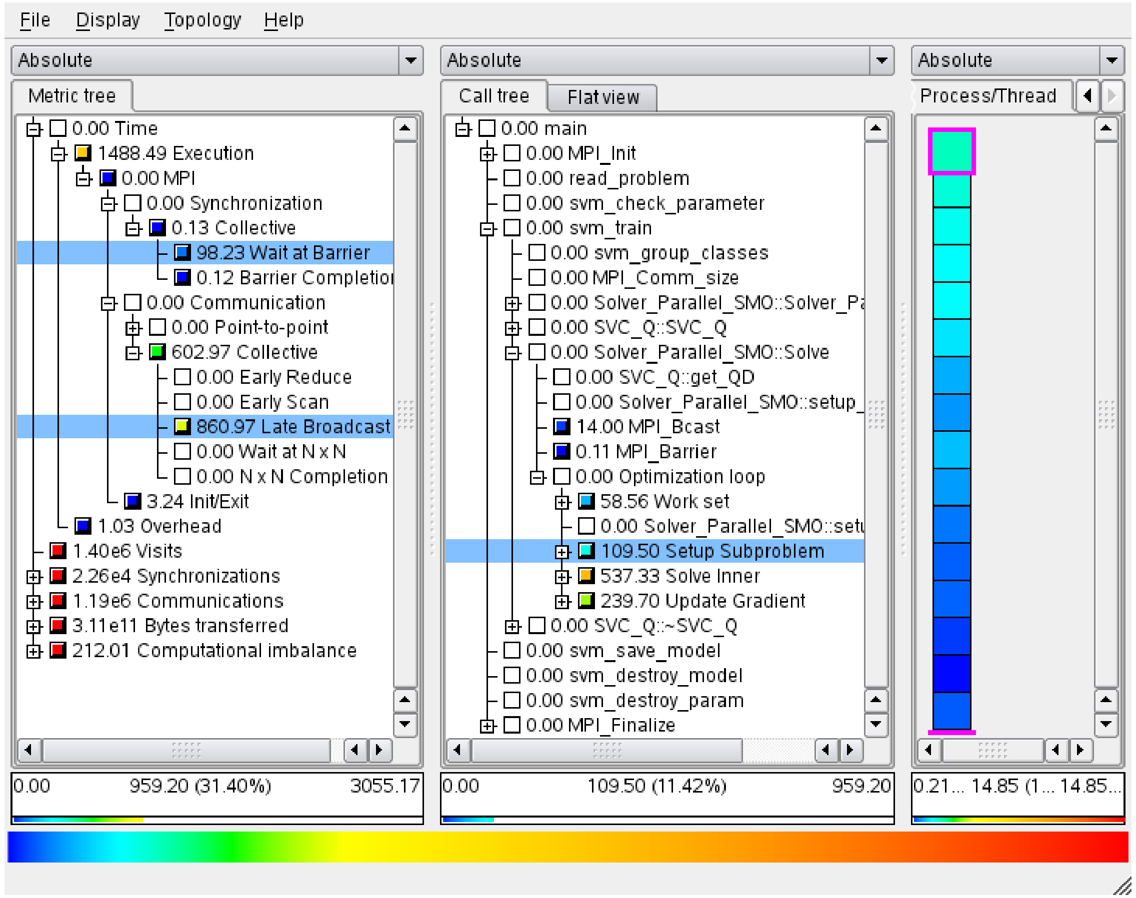
- Index of nested loops matter
  - Example: simple switch of indices makes a difference (e.g. memory access)

```
...
for (int j=0; j<dim2; j++) {

  for (int i=0; i<dim2; i++) {
    array[i][j] = testvalue();
  }
  ...
}
...
```

→

```
...
for (int j=0; j<dim2; j++) {

  for (int i=0; i<dim2; i++) {
    array[j][i] = testvalue();
  }
  ...
}
...
```

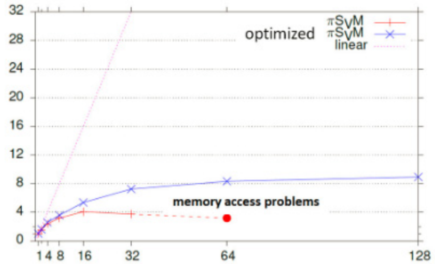
Understanding Communication with Scalasca for SVM Data Science Example

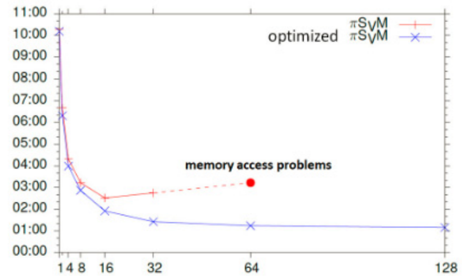



```
//Jeder Prozess hat l Werte an die Stelle Rang * l in p_cache_status geschrieben
for(int k = 0; k < p; ++k)
{
    //Jeder Prozess broadcastet sein Ergebnis zu allen anderen Prozessen
    MPI_Bcast(&p_cache_status[k * l], l, MPI_CHAR, k, comm);
}

for(int i = 0; i < p; ++i) {
    if(rank == i) { //Der sendende Prozess kopiert in den Sendebereich
        for(int j = 0; j < lmn; ++j)
            G_buf[j] = G_n[j];
    }
    //Alle anderen Prozesse erhalten die Daten
    MPI_Bcast(G_buf, lmn, MPI_DOUBLE, i, comm);
    //Und addieren sie auf
    for(int j = 0; j < lmn; ++j)
        G[not_work_set[j]] += G_buf[j];
}
```

Using MPI_Allgather() instead

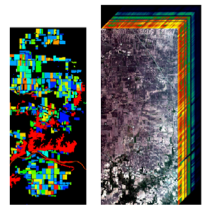
Using MPI_Allreduce() instead





[14] G. Cavallaro & M. Riedel & J.A. Benediktsson et al., 'On Understanding Big Data Impacts in Remotely Sensed Image Classification Using Support Vector Machine Methods', Journal of Applied Earth Observations and Remote Sensing

[7] SCALASCA Performance Tool



Optimizing by Improving MPI Function Calls (1)

- E.g. instead multiple `MPI_Bcast()` one `MPI_Allgather()`

```
...
for (int k = 0; k < p; ++k) {
  MPI_Bcast (&p_cache_status [k * 1], 1, MPI_CHAR , k, comm );
}
...
```

```
...
MPI_Allgather ( MPI_IN_PLACE , 1, MPI_CHAR , p_cache_status , 1, MPI_CHAR , 3 comm );
...
```

- **Good usage of MPI collective operations can significantly reduce the overall runtime (i.e. walltime)**
- **Overhead of each operation - it is better to call one MPI collective than multiple times another**

Optimizing by Improving MPI Function Calls (2)

- E.g. instead multiple `MPI_Bcast()` one `MPI_Allreduce()`

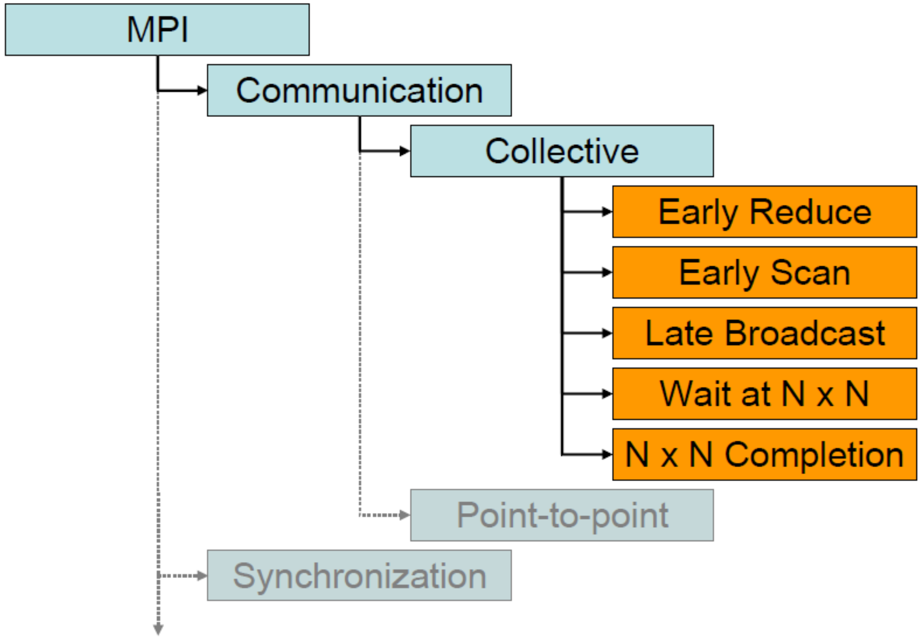
```
/* every process has lmn values written in data structure G_n *//
for ( int i = 0; i < p; ++i) {
  if( rank == i) {
    for ( int j = 0; j < lmn; ++j)
      G_buf [j] = G_n [j];
  }
  /* all other processors receive the data */
  MPI_Bcast (G_buf , lmn , MPI_DOUBLE , i, comm );
  /* values are added up */
  for ( int j = 0; j < lmn; ++j)
      G[ not_work_set [j]] += G_buf [j];
}
```

```
/* Adding up data from G_n in G_buf */
MPI_Allreduce (G_n , G_buf , lmn , MPI_DOUBLE , MPI_SUM , comm );
/* values are added up */
for ( int j = 0; j < lmn; ++j)
      G[ not_work_set [j]] += G_buf [j];
```

- **Bad usage of MPI collective operations are one cause for many ‘wrong usage patterns & problems‘**

Optimizing MPI Collective Communication

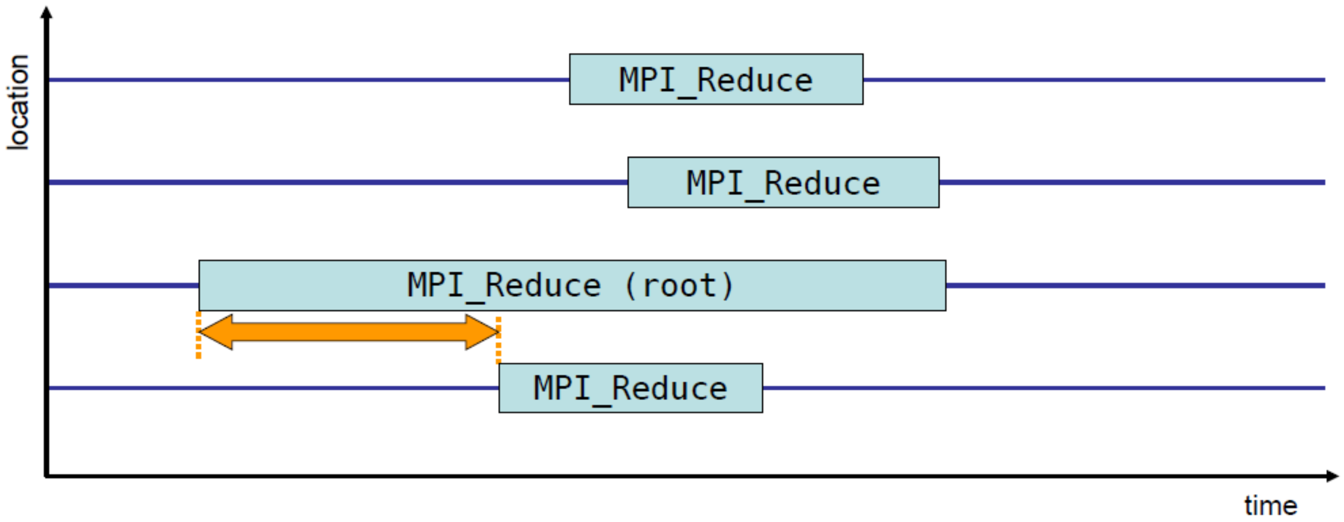
- Metrics: Communication - Time spent in MPI communication calls



[16] Metrics tour

Early Reduce Problem

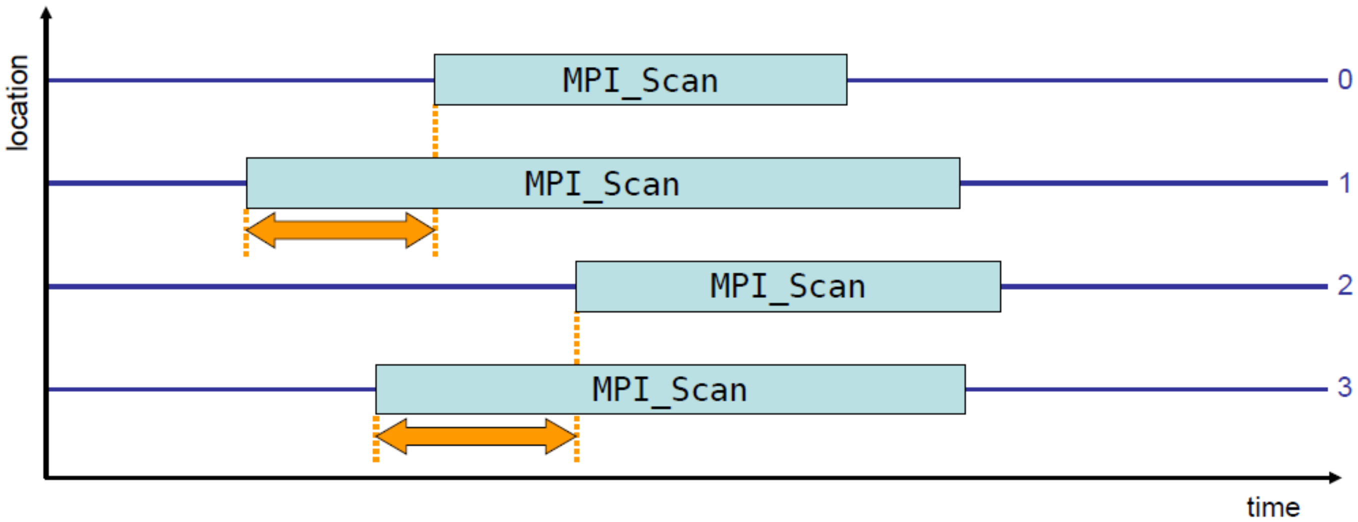
- Understanding the problem
  - Waiting time if the destination process (root) of a collective N-to-1 operation enters the operation earlier than its sending counterparts
  - Applies to: **`MPI_Reduce()`, `MPI_Gather()`, `MPI_Gatherv()`**



[16] Metrics tour

Early Scan Problem

- Understanding the problem
  - Waiting time if process *n* enters a prefix reduction operation earlier than its sending counterparts (i.e., ranks 0..*n*-1)
  - Applies to: **`MPI_Scan()`**

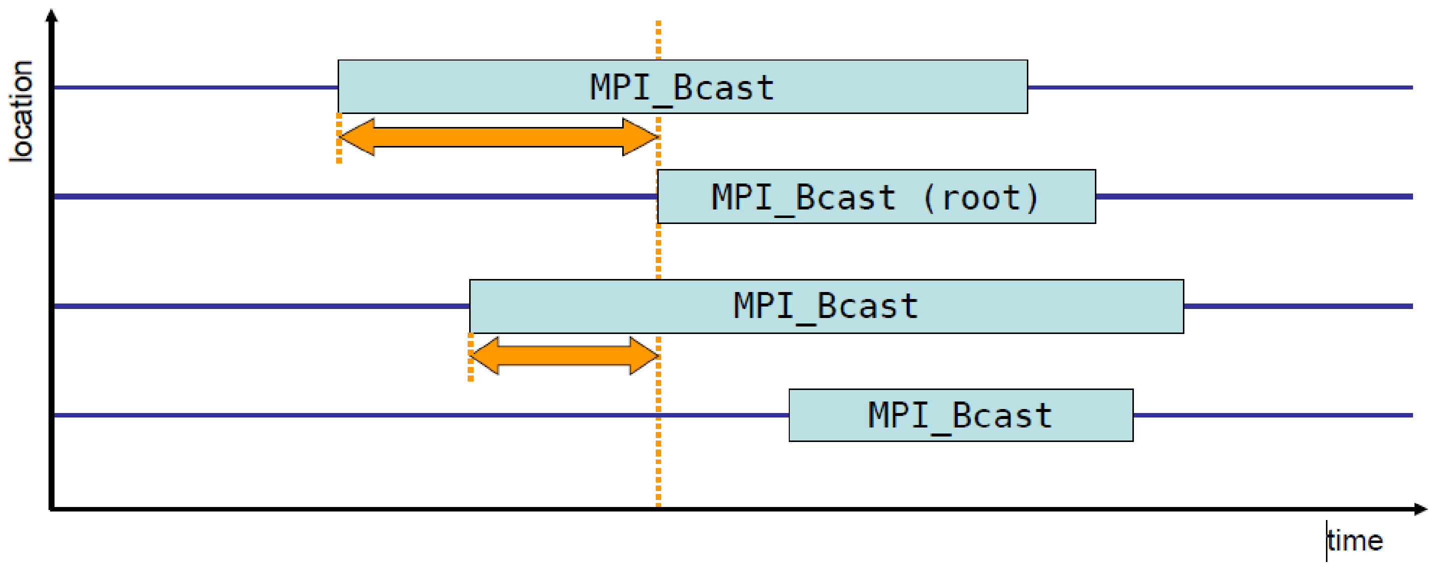


- **MPI_Scan() computes the scan (partial reductions) of data on a collection of processes - prefix reduction on process i includes the data from process i (here: 4 ranks)**

[16] Metrics tour

Late Broadcast Problem

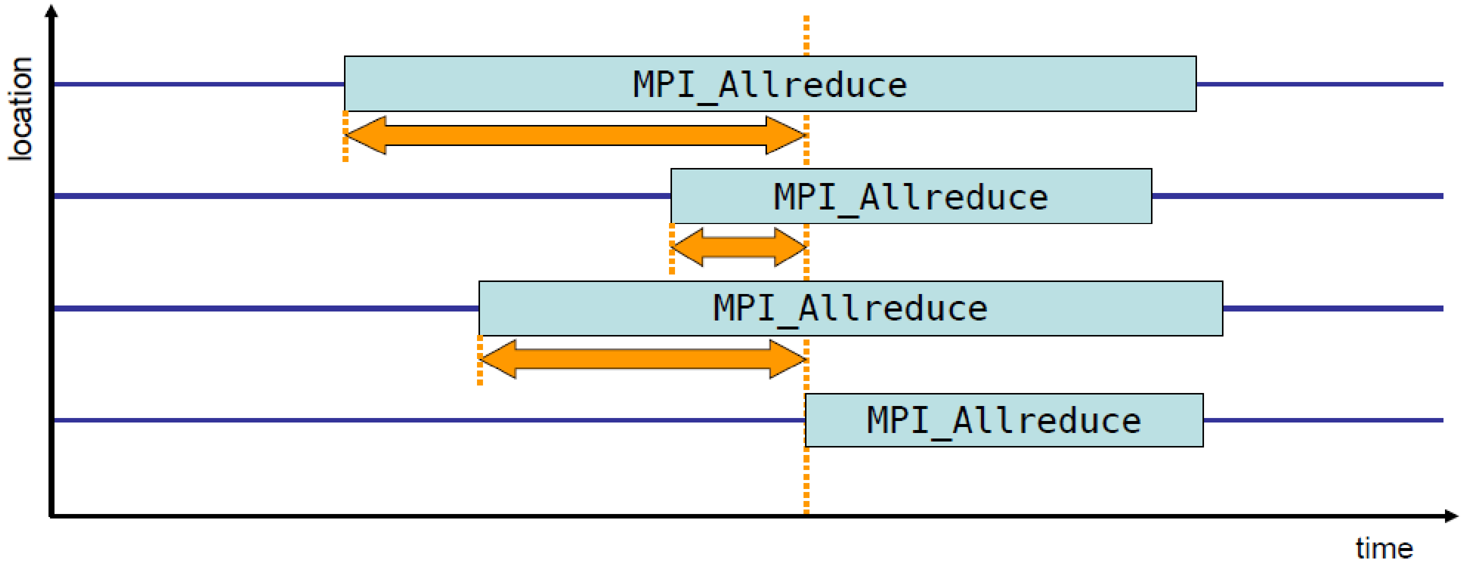
- Understanding the problem
  - Waiting times if the destination processes of a collective 1-to-N operation enter the operation earlier than the source process (root)
  - Applies to: **`MPI_Bcast()`**, **`MPI_Scatter()`**, **`MPI_Scatterv()`**



[16] Metrics tour

Wait at NxN Problem

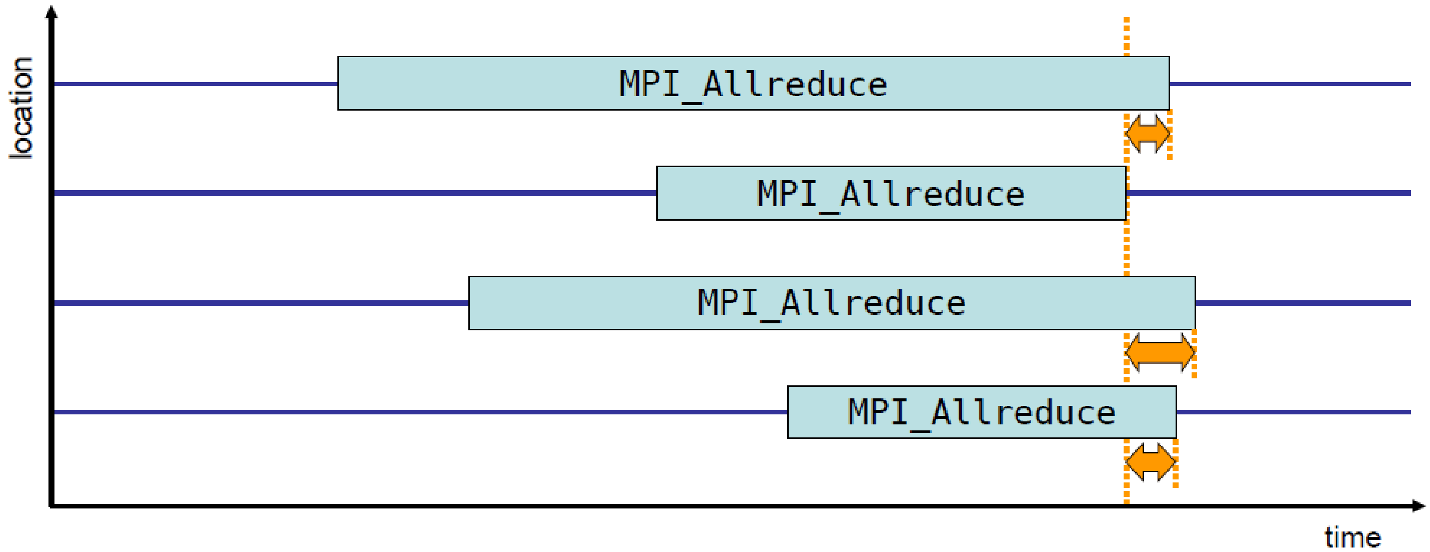
- Understanding the problem
  - Time spent waiting in front of a synchronizing collective operation call until the last process reaches the operation
  - Applies to: **`MPI_Allreduce()`**, **`MPI_Alltoall()`**, **`MPI_Alltoallv()`**, **`MPI_Allgather()`**, **`MPI_Allgatherv()`**, **`MPI_Reduce_scatter()`**



[16] Metrics tour

NxN Completion Problem

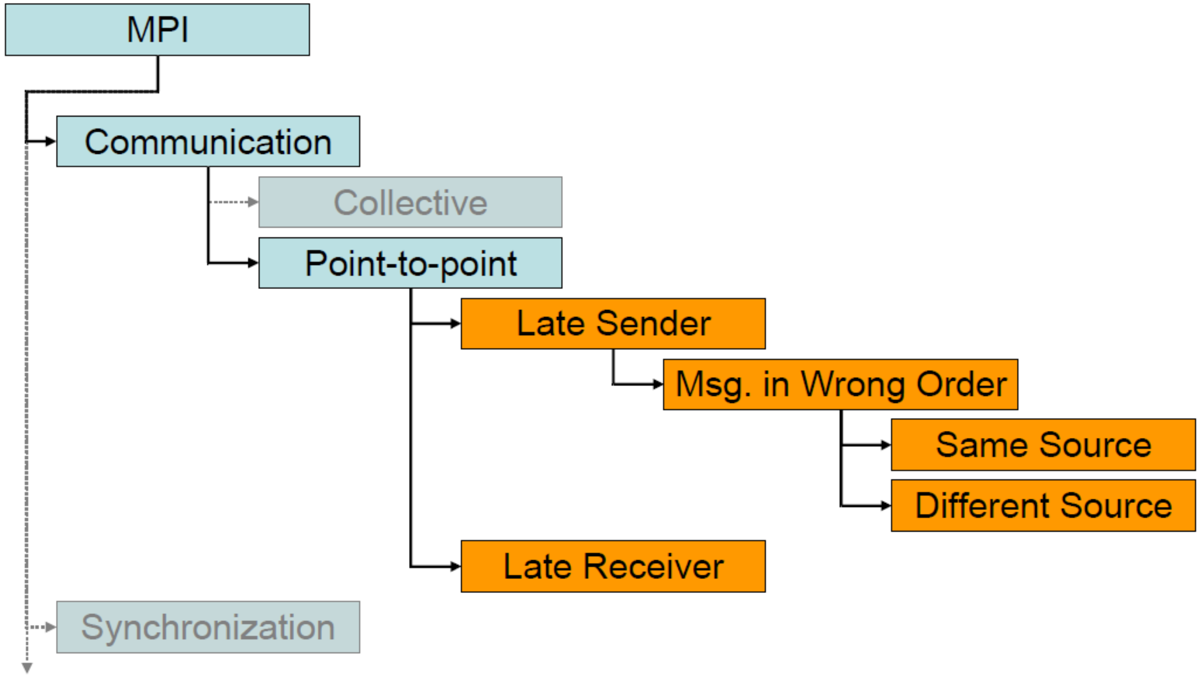
- Understanding the problem
  - Time spent in synchronizing collective operations after the first process has left the operation
  - Applies to: **`MPI_Allreduce(),MPI_Alltoall(),MPI_Alltoallv(), MPI_Allgather(),MPI_Allgatherv(),MPI_Reduce_scatter()`**



[16] Metrics tour

Optimizing MPI Point-to-Point Communication

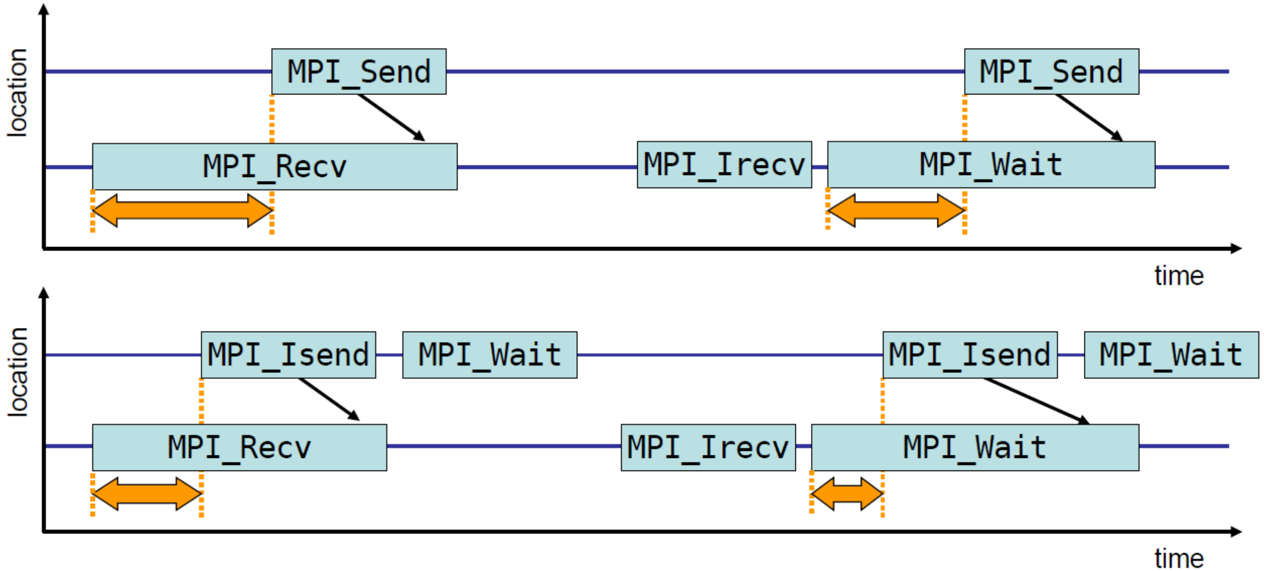
- Metrics: Communication - Time spent in MPI communication calls



[16] Metrics tour

Late Sender Problem (1)

- Understanding the problem
  - Waiting time caused by a blocking receive operation posted earlier than the corresponding send operation
  - Applies to blocking as well as non-blocking communication



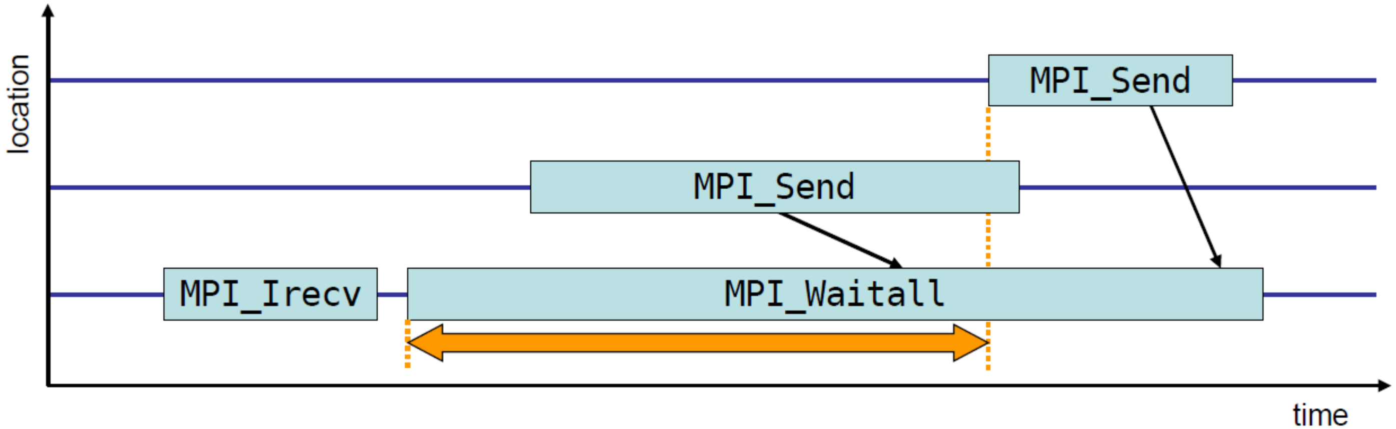
- Blocking vs. non-blocking: MPI_Send() blocks until data is received; MPI_Isend() continues

[16] Metrics tour

Late Sender Problem (2)

- Understanding the problem
  - While waiting for several messages, the maximum waiting time is accounted
  - Applies to **`MPI_Waitall()`**, **`MPI_Waitsome()`**

- **MPI_Waitall() does wait for all given MPI requests (e.g. waiting for message) to complete before continuing**

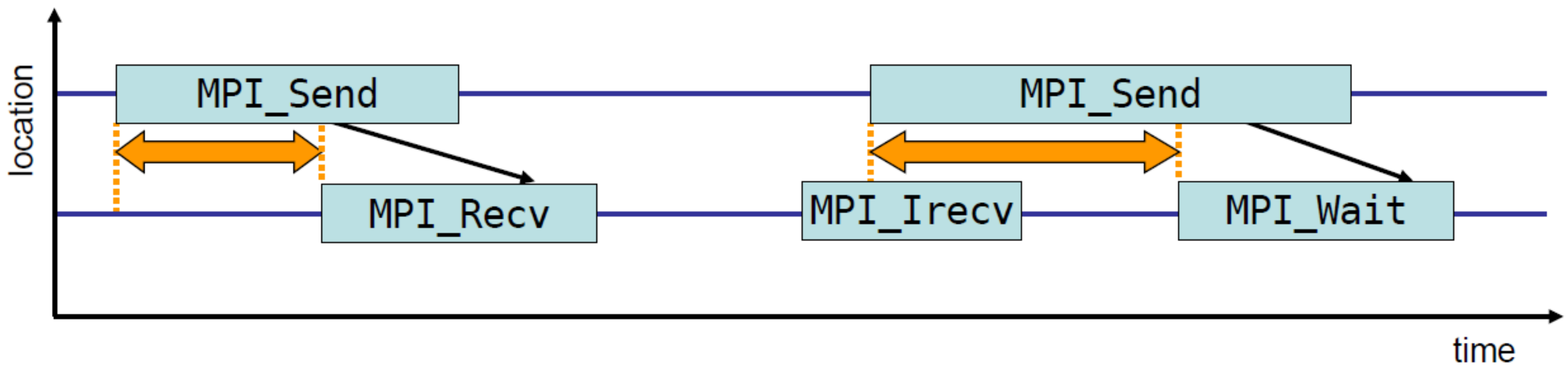


[16] Metrics tour

Late Sender Problem (3)

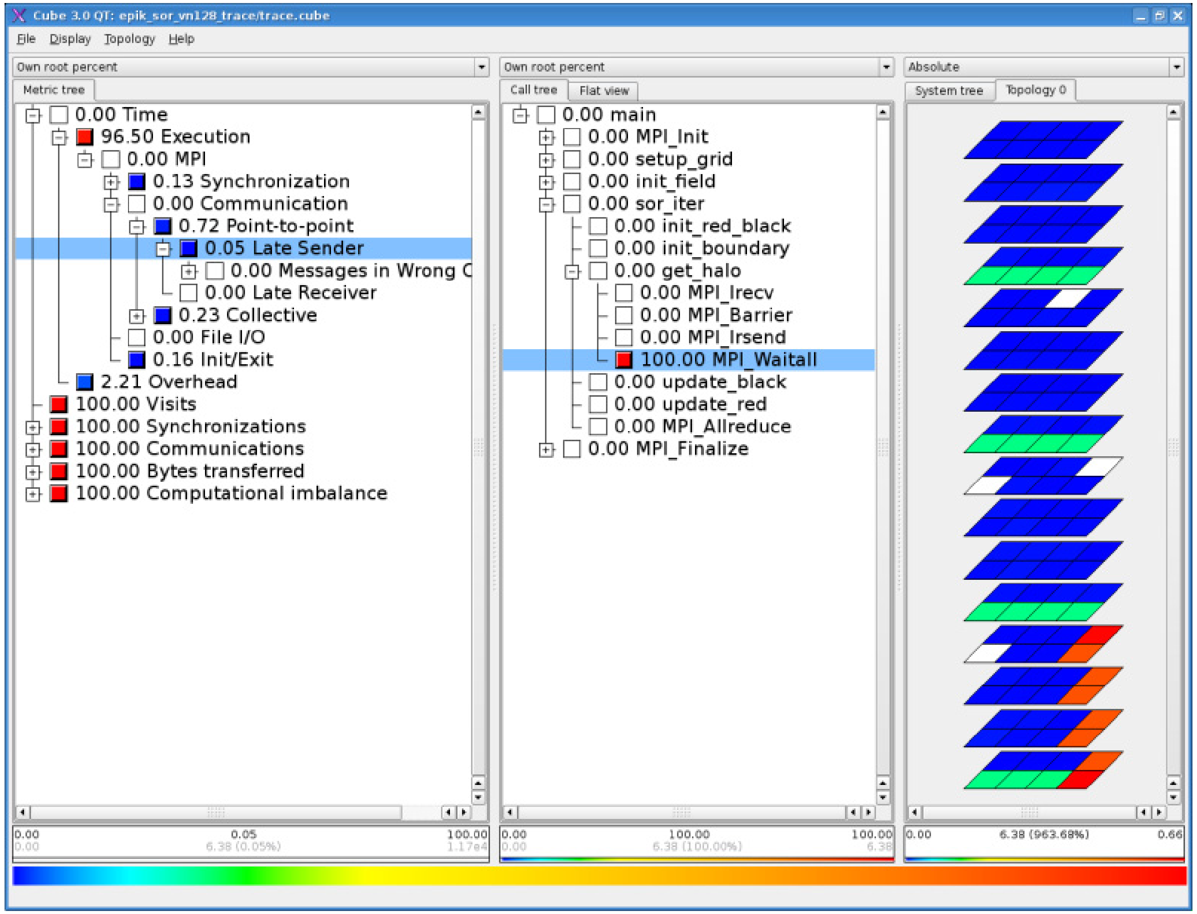
- Understanding the problem
  - Refers to Late Sender situations which are caused by messages received in wrong order
  - Two flavours: (a) Messages sent from same source location; (b) Messages sent from different source locations

- **MPI_Wait() does wait for a given MPI request to complete before continuing**



[16] Metrics tour

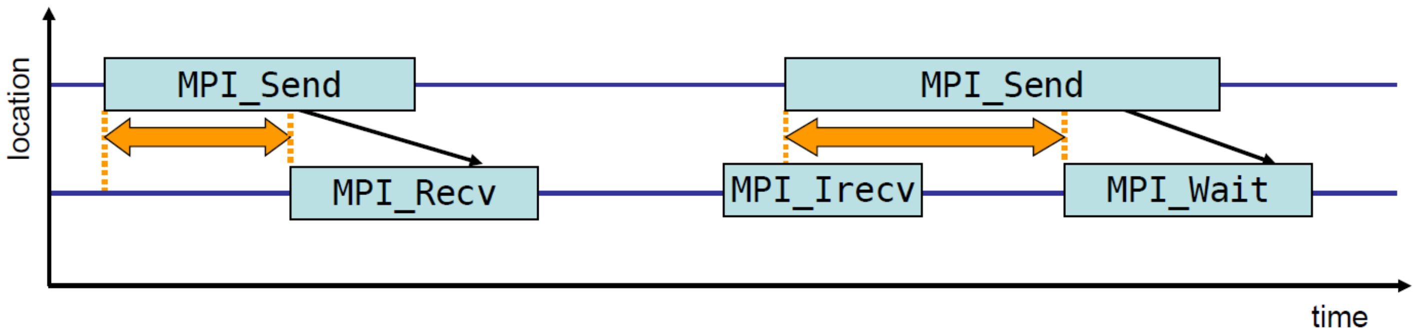
Late Sender Problem – Scalasca CUBE Analysis



[19] Scalasca User Guide

Late Receiver Problem

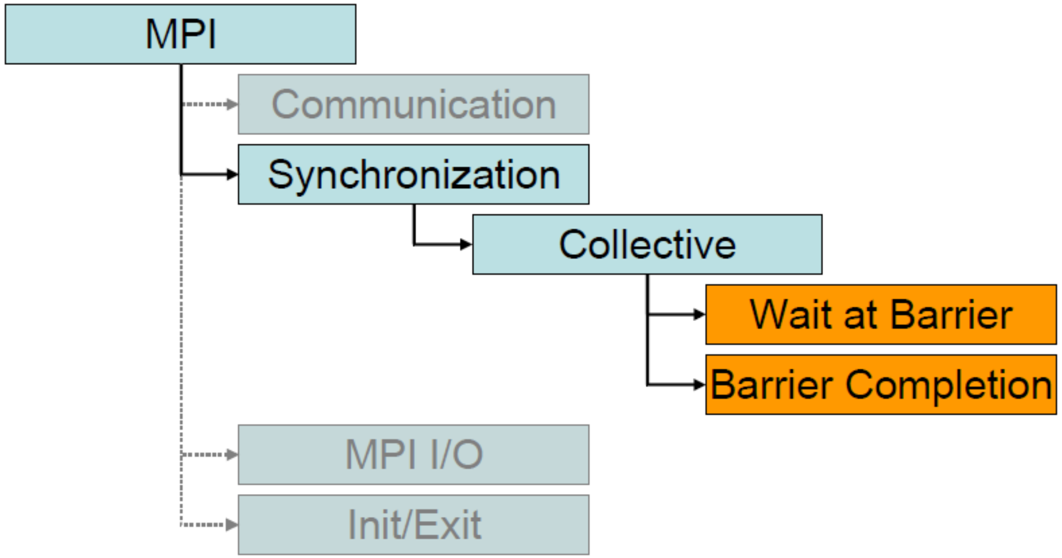
- Understanding the problem
  - Waiting time caused by a blocking send operation posted earlier than the corresponding receive operation
  - Calculated by receiver but waiting time attributed to sender
  - Applies not to non-blocking sends



[16] Metrics tour

Optimizing MPI Synchronization

- Metrics: Synchronization - Time spent in calls to `MPI_Barrier()`

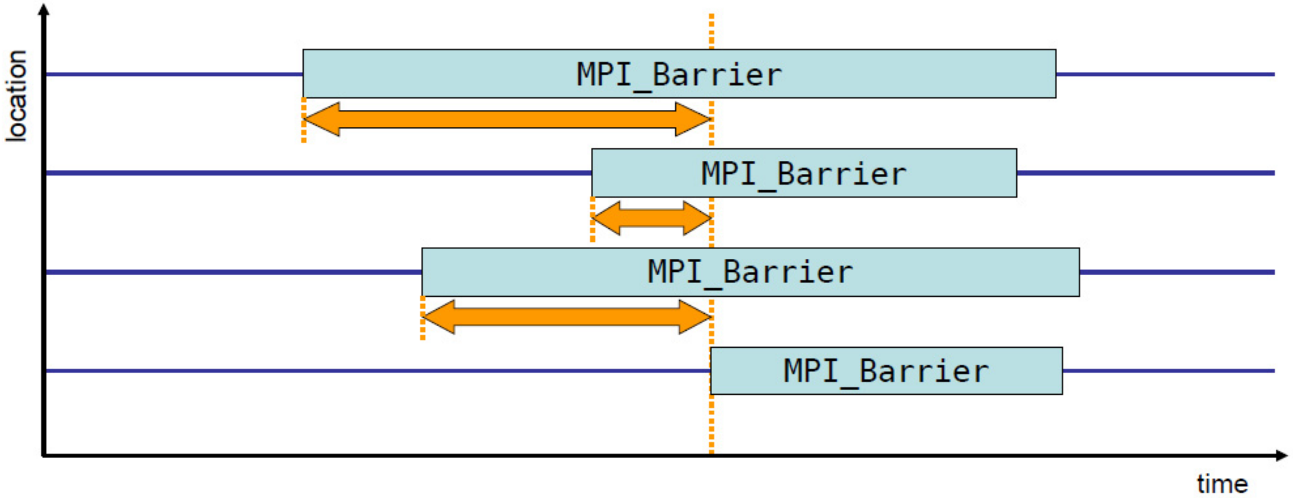


[16] Metrics tour

Wait at Barrier Problem

- Understanding the problem
  - Time spent waiting in front of a barrier call until the last process reaches the barrier operation
  - Applies to: **`MPI_Barrier()`**

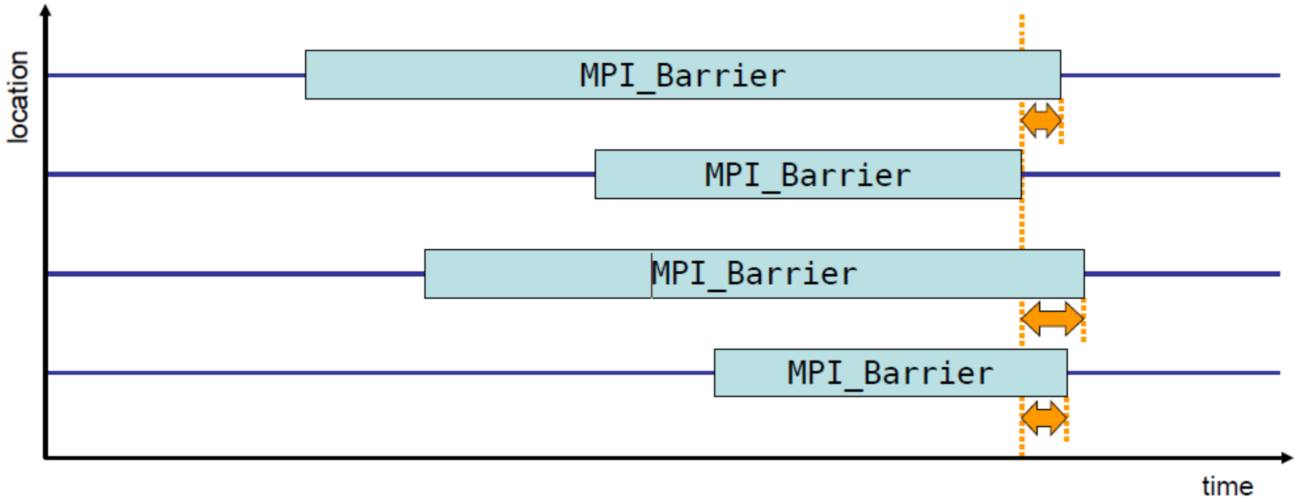
- **MPI_Barrier() blocks the caller until all processes in the communicator have called it for synchronisation**



[16] Metrics tour

Barrier Completion Problem

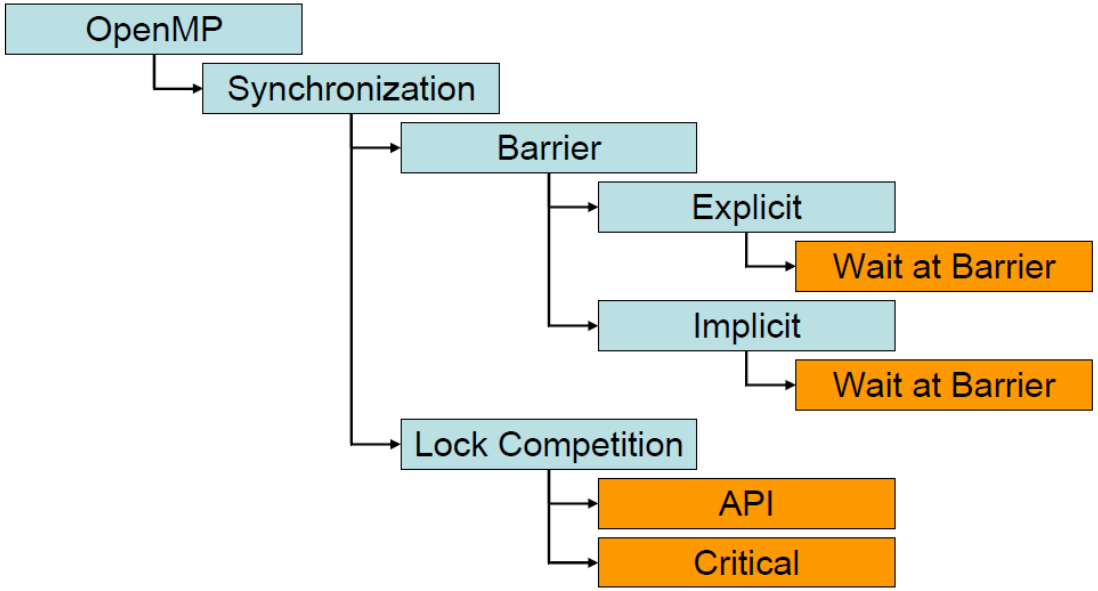
- Understanding the problem
  - Time spent in barrier after the first process has left the operation
  - Applies to: `MPI_Barrier()`



[16] Metrics tour

Optimizing OpenMP Synchronization

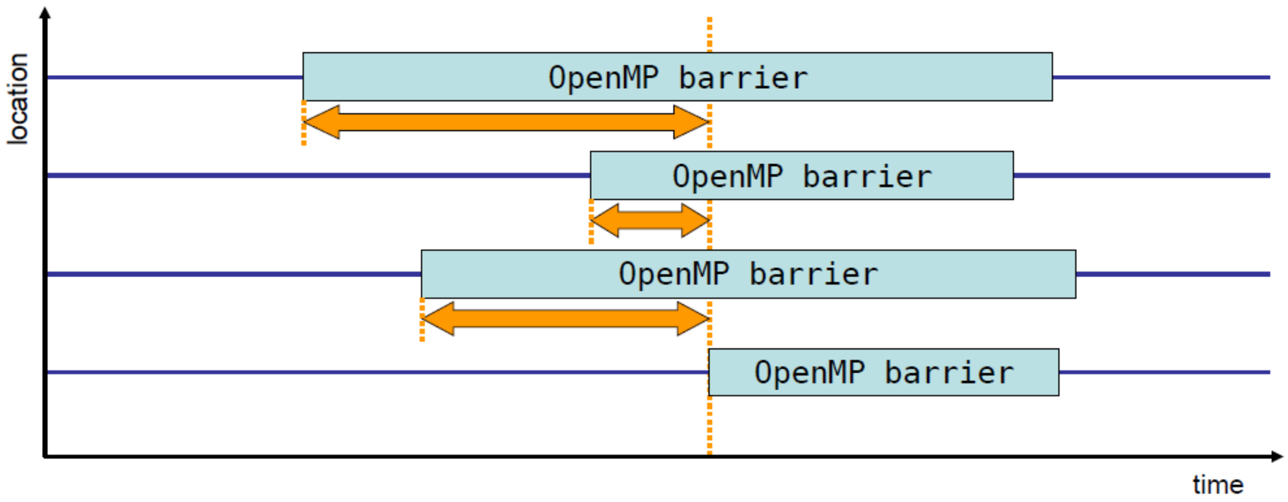
- Metrics: Synchronization
  - Time spent for synchronizing OpenMP threads



[16] Metrics tour

Wait at Barrier Problem

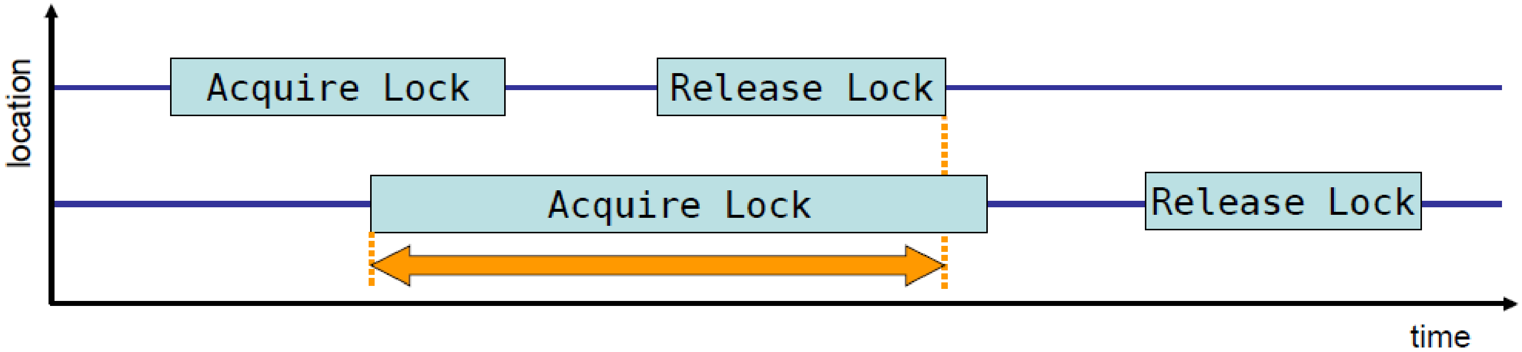
- Understanding the problem
  - Time spent waiting in front of a barrier call until the last process reaches the barrier operation
  - Applies to: Implicit/explicit barriers



[16] Metrics tour

Lock Competition (API & Critical Regions) Problem

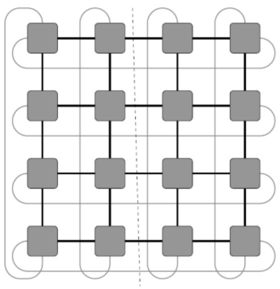
- Understanding the problem
  - Time spent waiting for a lock that has been previously acquired by another thread
  - Applies to: critical sections, OpenMP lock Application Programming Interface (API)

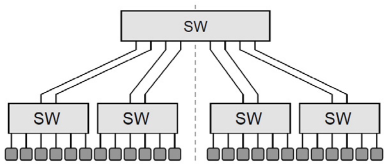


[16] Metrics tour

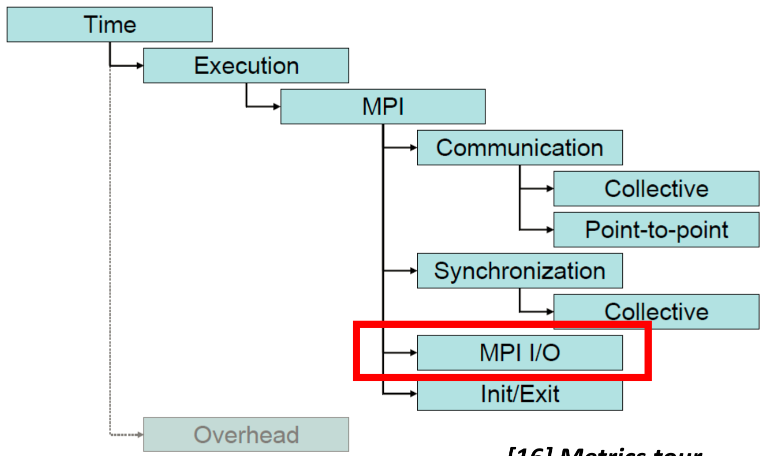
Optimization on Hardware & I/O – Revisited

- Optimizations in terms of software & hardware are important
  - Optimization can be interpreted as using 'dedicated' hardware features
  - E.g. network interconnections enable different used 'network topologies'
  - E.g. parallel codes are tuned applying parallel I/O with parallel filesystems



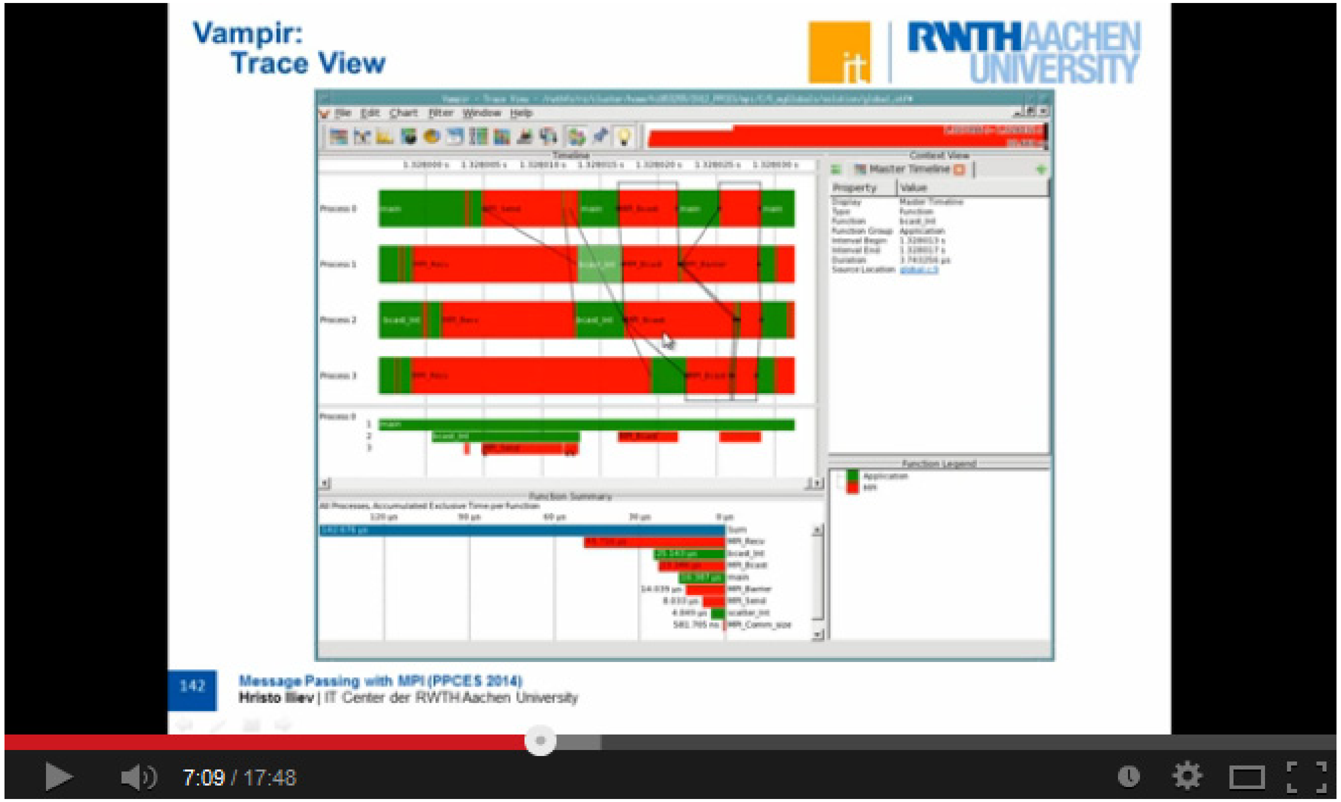


[6] Introduction to High Performance Computing for Scientists and Engineers



[16] Metrics tour

[Video] Vampir Toolset Example



[20] Vampir Trace Demo Video

Lecture Bibliography

Lecture Bibliography (1)

- [1] Debuggers & Parallel Debugging – HPC Best Practices, SciNet, Toronto, Online:
  http://wiki.scinethpc.ca/wiki/images/c/ce/Best-practice-debug.pdf
- [2] PRACE Training, Online:
  http://www.training.prace.ri/material/index.html
- [3] OpenMPI, 'FAQ: Debugging applications in parallel', Online:
  http://www.open-mpi.org/faq/?category=debugging#serial-debuggers
- [4] TotalView Debugger, Online:
  http://www.roguewave.com/products/totalview.aspx
- [5] YouTube Video, 'MPI Debugging with the TotalView debugger', Online:
  http://www.youtube.com/watch?v=ErfwWXdGJMo
- [6] Introduction to High Performance Computing for Scientists and Engineers, Georg Hager & Gerhard Wellein, Chapman & Hall/CRC Computational Science, ISBN 143981192X
- [7] Scalasca Flyer – Scalasca Performance Analysis Tool, Online:
  http://www.scalasca.org/
- [8] Wikipedia on 'Wall-clock time', Online:
  http://en.wikipedia.org/wiki/Wall-clock_time
- [9] B2SHARE, 'piSVM Analytics Runtimes JUDGE Cluster Rome Images 55 Features', Online:
  http://hdl.handle.net/11304/69430fd2-e7d6-11e3-b2d7-14feb57d12b9
- [10] Valgrind tool, Online:
  http://valgrind.org/
- [11] VAMPIR Performance Analysis Tool, Online:
  http://www.vampir.eu/

Lecture Bibliography (2)

- [12] Species Iris Group of North America Database, Online:
  http://www.signa.org
- [13] M. Goetz, C. Bodenstein, M. Riedel, 'HPDBSCAN – Highly Parallel DBSCAN', in proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC2015), Machine Learning in HPC Environments (MLHPC) Workshop, 2015, Online:
  https://www.researchgate.net/publication/301463871_HPDBSCAN_highly_parallel_DBSCAN
- [14] G. Cavallaro, M. Riedel, M. Richerzhagen, J. A. Benediktsson and A. Plaza, "On Understanding Big Data Impacts in Remotely Sensed Image Classification Using Support Vector Machine Methods," *in the IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 8, no. 10, pp. 4634-4646, Oct. 2015, Online:
  https://www.researchgate.net/publication/282524415_On_Understanding_Big_Data_Impacts_in_Remotely_Sensed_Image_Classification_Using_Support_Vector_Machine_Methods
- [15] A. Gulli and S. Pal, 'Deep Learning with Keras' Book, ISBN-13 9781787128422, 318 pages, Online:
  https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-keras
- [16] M. Geimer et al., 'SCALASCA performance properties "The metrics tour"'
- [17] Open Trave Format, Online:
  http://www.paratools.com/OTF
- [18] A. Knuepfer, F. Wolf, M. Gerndt, S. Shende, 'The Future of the Open Trace Format (OTF) and Open Event Trace Recording', Online:
  http://www.gauss-allianz.de/files/pdf/silc/silc_otf2_sc_bof_2010-11_slides.pdf
- [19] Scalasca User Guide, Online:
  http://apps.fz-juelich.de/scalasca/releases/scalasca/1.4/docs/UserGuide.pdf
- [20] YouTube Video, 'Introduction to VampirTrace and Vampir, by Hristo Iliev ', Online:
  http://www.youtube.com/watch?v=pZVSs__h76Q


Data
Research
Science
Big
science
big
simulations
processing
analysis
methods
computational
Computational
technologies
TB
computing
European
Supercomputing
Scientific
research
Climate
modelling
structures
HPC
Juelich
access
datasets
performance
management
data-intensive
distributed
large-scale
Service
Centre
climate
Computing
nodes
centers
storage
community
ENES
EUDAT
Environment
scientific
important
Infrastructure
Enable
Modelling
images
often
databases
brain
Project
E.g
e.g
High
PB
services
measurement
within
fusion
new
funding
concepts
device
Analysis
Computer
expertise
Approaches
construction
Fusion
different
files
Policy-based
forms
cross-disciplinary
resources
year
like
Test
Cross-Disciplinary
EU
etc
DKRZ
Use
Key
disciplines
results
basis
increasing
via
physical
IPCC
Common
Methods
Using
Work
model
international
SMAQ
engineering
References
formats
codes
MapReduce
create
GB
JSC
use
DLCL
Level
stored
Resources
analyze
Large
Summary
Matters
Concrete
external
levels
system
Landscape
acquisition
System
DLCLs
Understanding
Germany
Provide
NASA
Energy
Simulation
systems
AR5
Hardware
Technologies
Earth
directory
day
Processing
Health
cores
compute
project
Structure
General
current
hundreds
Nutshell
Services
network