

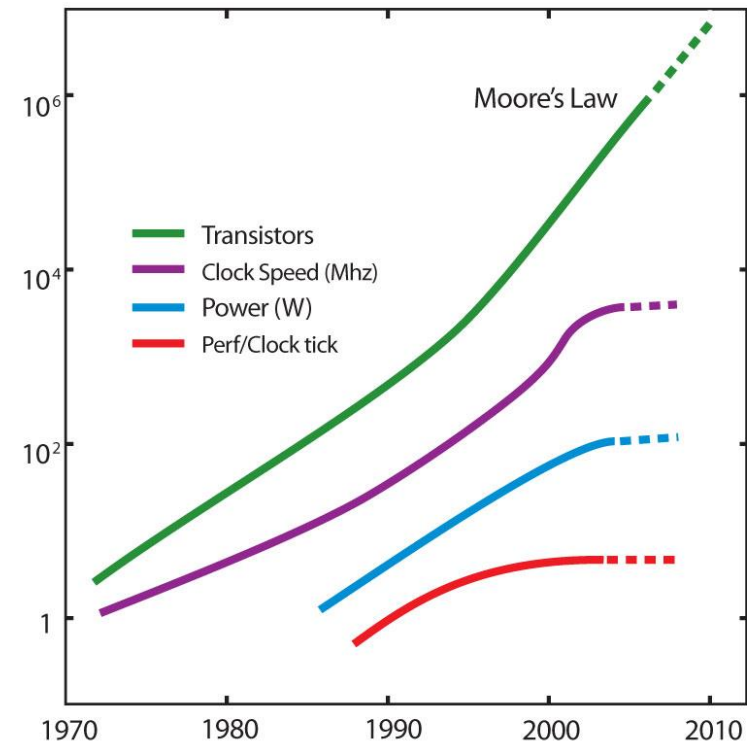
Challenges of the Exascale era

... and how tools can help

2013-04-09 | Markus Geimer
Jülich Supercomputing Centre
m.geimer@fz-juelich.de

The “free lunch” is over

- Moore's law is still in charge, but
 - Clock rates no longer increase since a couple of years
 - Performance gains only through increased parallelism
- Optimization of applications more difficult due to increasing machine complexity
 - Hierarchical networks / memory
 - More CPUs / multi-core / accelerators



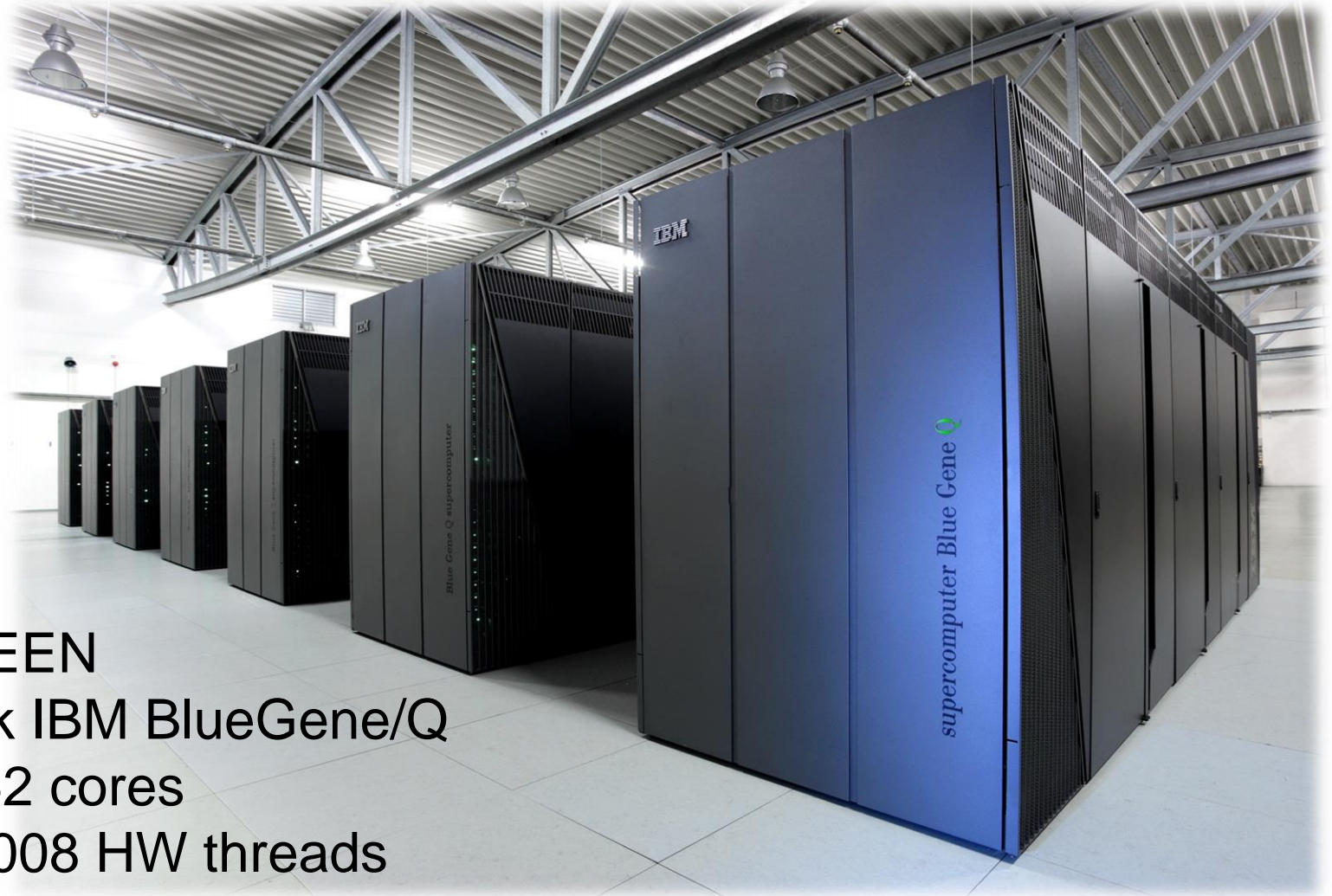
Personal motivation (I)



JUGENE
72 rack IBM BlueGene/P
294,912 cores

Most parallel system
in the world
06/2009 to 06/2011 !!!

Personal motivation (II)



JUQUEEN
28 rack IBM BlueGene/Q
458,752 cores
1,835,008 HW threads

Projection for an Exascale system*

System attributes	2010	"2015"		"2018"		Difference 2010 & 2018
System peak	2 Pflop/s	200 Pflop/s		1 Eflop/sec		O(1000)
Power	6 MW	15 MW		~20 MW		
System memory	0.3 PB	5 PB		32-64 PB		O(100)
Node performance	125 GF	0.5 TF	7 TF	1 TF	10 TF	O(10) – O(100)
Node memory bandwidth	25 GB/s	0.1 TB/sec	1 TB/sec	0.4 TB/sec	4 TB/sec	O(100)
Node concurrency	12	O(100)	O(1,000)	O(1,000)	O(10,000)	O(100) – O(1000)
Total Concurrency	225,000	O(10 ⁸)		O(10 ⁹)		O(10,000)
Total Node Interconnect BW	1.5 GB/s	20 GB/sec		200 GB/sec		O(100)
MTTI	days	O(1day)		O(1 day)		- O(10)

* From <http://www.exascale.org>

Exascale performance challenges

- Exascale systems will consist of
 - Complex configurations
 - With a huge number of components
 - Very likely heterogeneous
 - Deep software hierarchies of large, complex software components will be required to make use of such systems
- ⇒ Operating systems, runtime & I/O systems, programming models, compilers, frameworks & libraries, debuggers & performance analysis tools, ...

Cross-cutting considerations

- Performance-aware design, development and deployment of hardware and software necessary
- Integration with OS, compilers, middleware and runtime systems required
- Support for performance observability in HW and SW (runtime) needed
- Enable performance measurement and optimization in case of HW and SW changes due to faults or power adaptation

Technical challenges

- Heterogeneity
 - General purpose CPUs, specialized cores, accelerators, ...
- Extreme concurrency
 - On various levels:
 - SIMD
 - Multi-/Many-core
 - Interconnected nodes
- Perturbation and data volume
 - Resilience
 - Input / Output

The basics

(First make it right...)

The often forgotten “tool”: The compiler

- Know the available optimization flags
 - In particular: inter-procedural and architecture-specific optimizations
- Increase warning level
 - ... and take these warnings serious 😊
- Perform some runs with runtime checks turned on
 - With production workloads!
 - Especially array bounds checking
- GCC 4.8 and Clang 3.2 also come with integrated memory error and data race detectors
- But **you** can also assist the compiler
 - Function prototypes, modules & interfaces, const correctness, intent specifiers, compiler-specific attributes

Static code analysis

- Can detect errors a compiler doesn't detect, e.g.,
 - NULL pointer dereferences, bounds checking, memory leaks, variable scoping, unused functions, portability issues
- ... but sometimes suffer from false positives
- Not available for Fortran (to the best of my knowledge)
- Open-source examples:
 - Cppcheck (C++), Clang static analyzer (C/C++/ObjC), Splint (C)
- Commercial examples:
 - FlexeLint (C/C++), Coverity (C/C++/C#/Java), Insure++ (C/C++)

Memory error checking

- Can detect errors such as
 - Memory leaks
 - Memory corruption
 - Allocation/deallocation API mismatches
- Open-source example:
 - Valgrind's "memcheck" tool
- Commercial example:
 - Intel Inspector XE

Going parallel

(Still making it right...)

MPI correctness checking (I)

- Check for correct MPI usage at runtime
 - Conformance to MPI standard
 - Parameters passed to MPI
 - MPI resource usage
 - Detect deadlocks
- Open-source example:
 - MUST
 - Developed by TU Dresden, LLNL & LANL
 - <http://tu-dresden.de/zih/must/>

MPI correctness checking (II)

- Open MPI provides (limited) MPI parameter checking
- Also interfaces with Valgrind to detect MPI-specific memory errors at runtime, e.g.,
 - Accessing buffer under control of non-blocking communication
 - Wrong input parameters (e.g., wrongly sized send buffers)
 - Uninitialized input buffers

Threading error detection

- Getting multi-threaded codes right is non-trivial!
- Automatic thread error detectors can help identify
 - Data races
 - Potential deadlocks
 - Even if they don't show up within your environment and under your typical workload!
- Open-source example:
 - Valgrind's "hellgrind" and "DRD" (data race detector) tools
- Commercial/closed-source examples:
 - Intel Inspector XE
 - Oracle Thread Analyzer

Parallel debugging

- Using printf() for debugging is cumbersome with parallel applications 😊
 - ➔ Better use full-featured or specialized debuggers
- Open-source examples:
 - GDB integration in Eclipse
 - Scales up to a few hundred processes
 - STAT: scalable collection of stack traces
- Commercial examples:
 - RogueWave TotalView
 - Allinea DDT

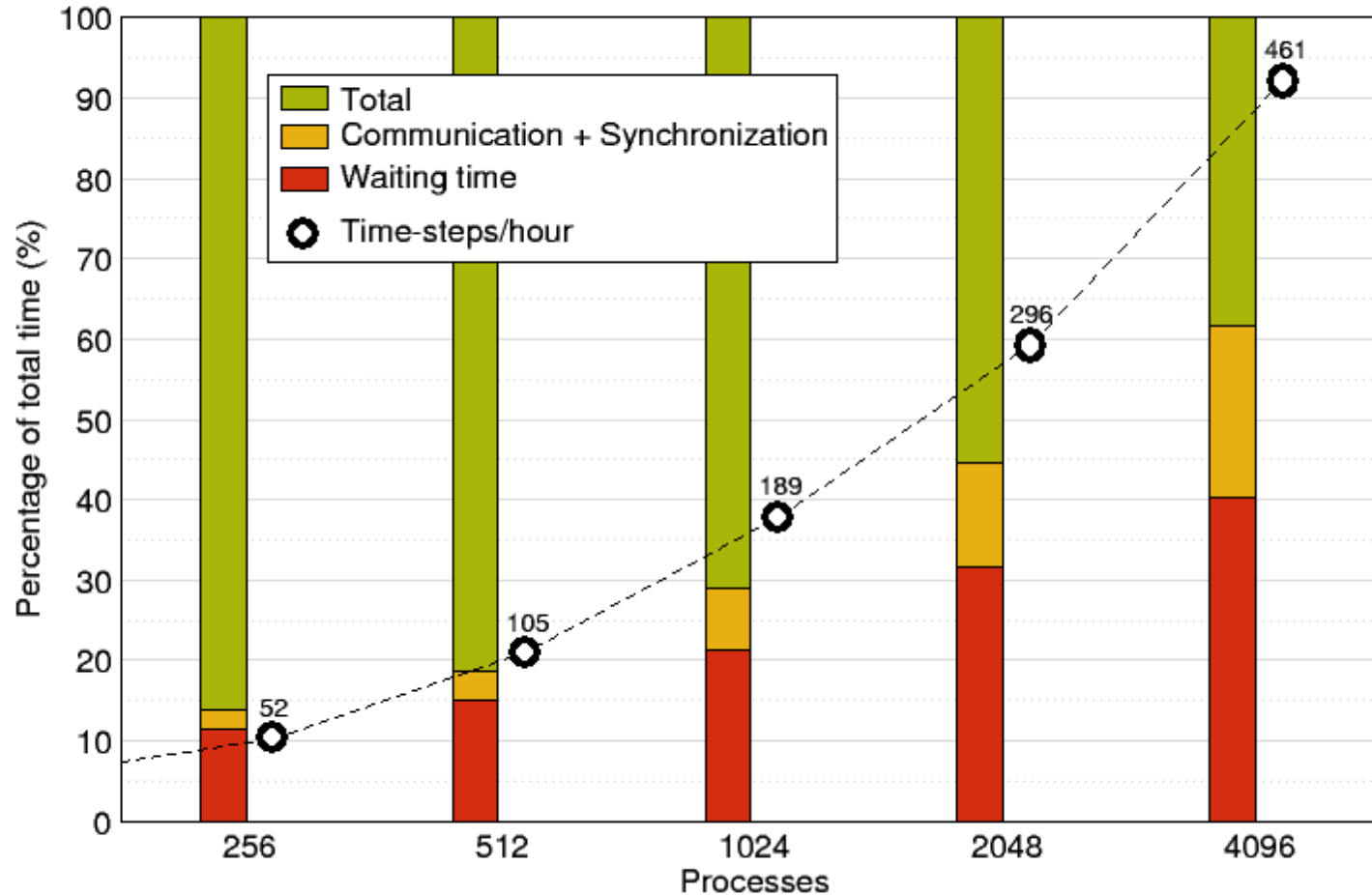
Performance analysis, optimization and scaling

(Then make it fast...)

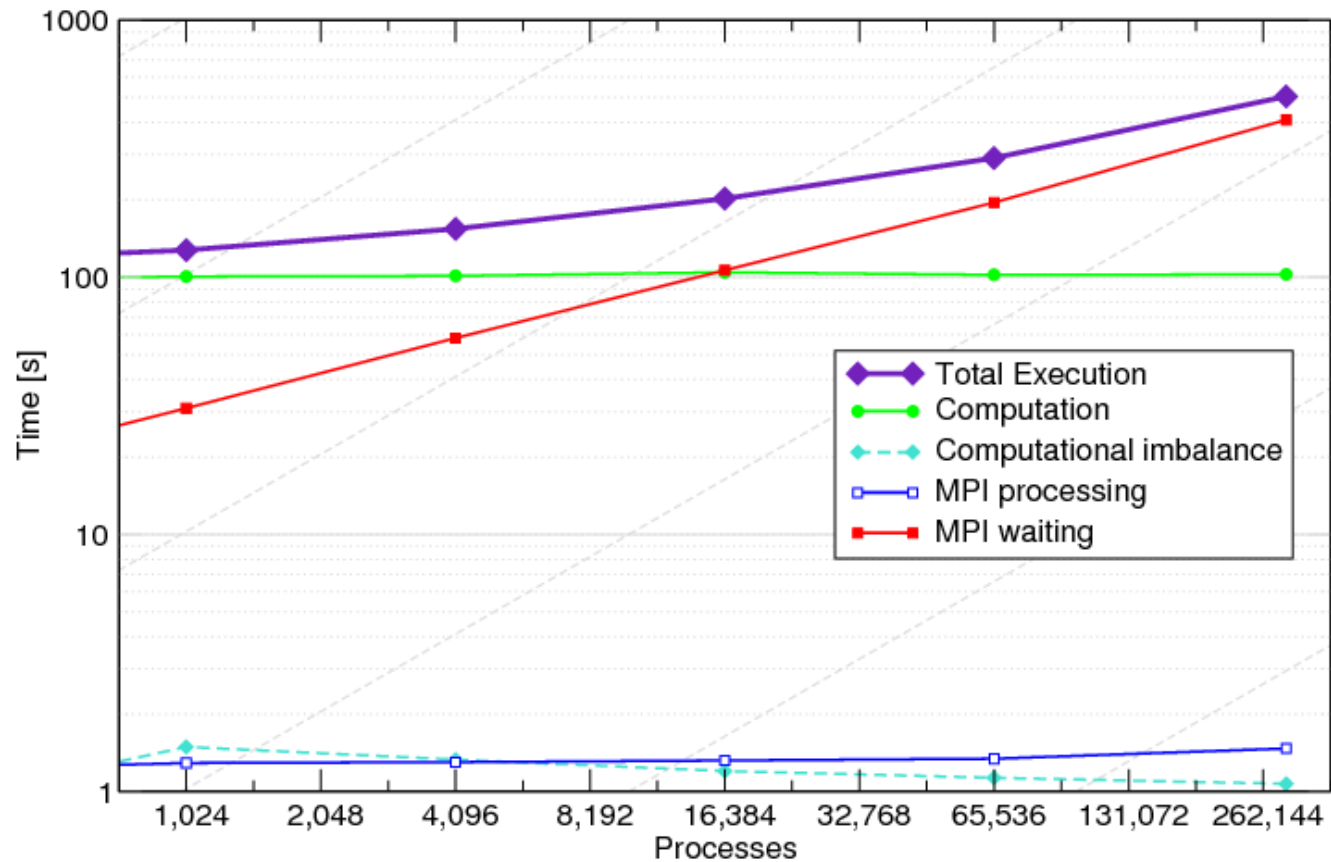
Tuning basics

- Successful performance engineering is a combination of
 - The right algorithms and libraries
 - Compiler flags and directives
 - Thinking !!!
- Measurement is better than guessing
 - To determine performance bottlenecks
 - To compare alternatives
 - To validate tuning decisions and optimizations
 - ☞ After each step!
 - ☞ Every doubling of scale reveals a new bottleneck!

Example: XNS wait-state analysis on BG/L (2007)



Example: Sweep3D wait-state analysis on BG/P (2010)



Performance factors of parallel applications (I)

- “Sequential” factors
 - Computation
 - ☞ Choose right algorithm, use optimizing compiler
 - Cache and memory
 - ☞ Tough! Only limited tool support, hope compiler gets it right
 - Input / output
 - ☞ Often not given enough attention, but extremely important at scale!

Performance factors of parallel applications (II)

- “Parallel” factors
 - Partitioning / decomposition
 - Communication (i.e., message passing)
 - Multithreading
 - Synchronization / locking
 - ➔ More or less understood, good tool support

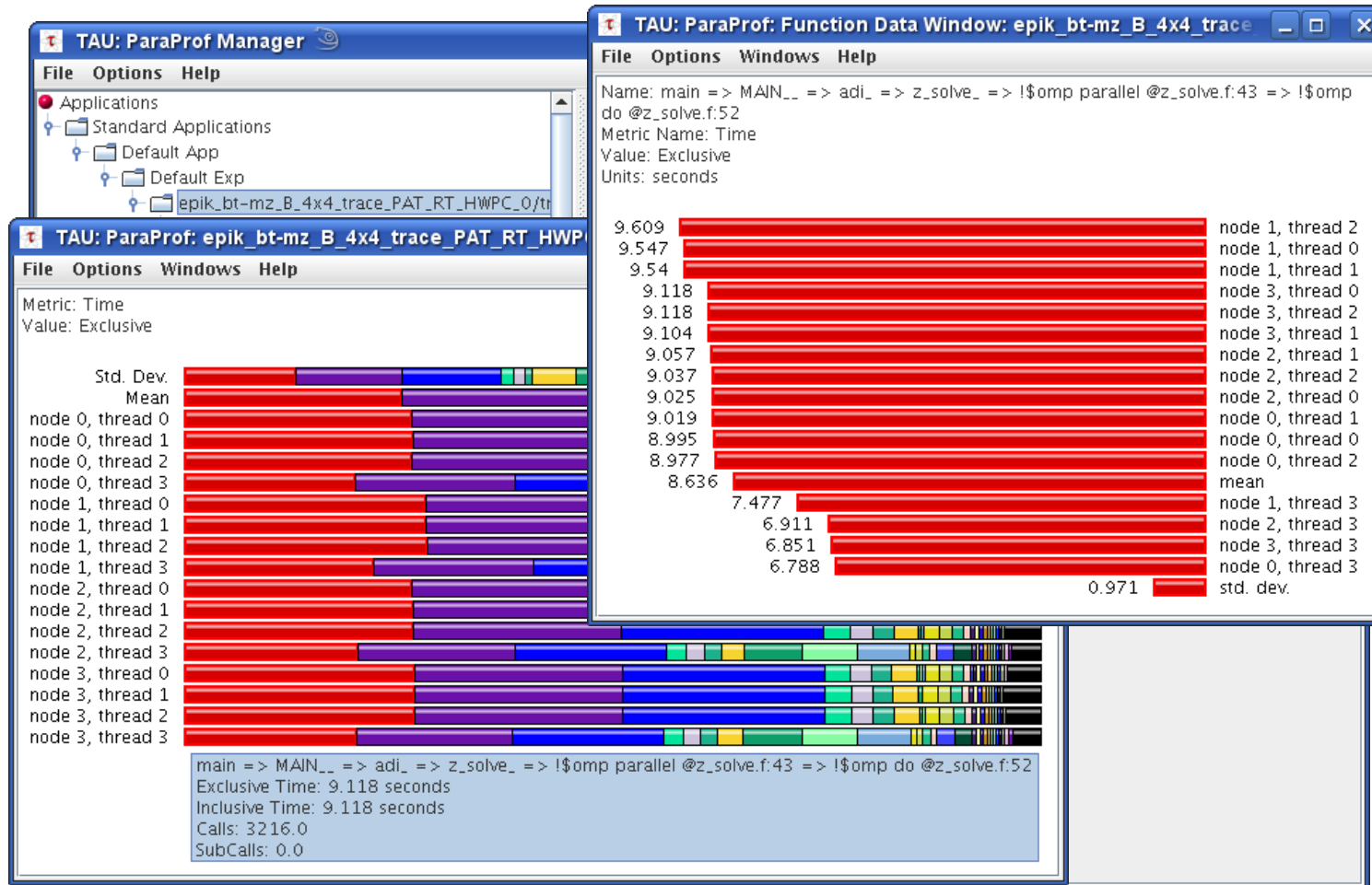
Memory / cache optimization

- Important to achieve good node-level performance
- Some tools can help identifying memory issues, such as
 - Cache utilization
 - False sharing
 - Suboptimal prefetching
 - NUMA effects
- Open-source examples:
 - Valgrind's "cachegrind" and "callgrind" tools
 - KCachegrind (GUI)
- Commercial examples:
 - RogueWave ThreadSpotter
 - Intel Amplifier XE

Scalable profiling

- Profile:
 - Records aggregated information
 - Data volume (more or less) independent of program runtime
 - Shows distribution of metrics (e.g., time or visits) across routines or call-paths
 - Sufficient to find computational hot spots
- Open-source examples:
 - HPCToolkit (Rice University)
 - TAU (U. of Oregon)
 - Score-P (community project, see <http://www.score-p.org>)

Example: TAU



Scalable tracing for in-depth analysis

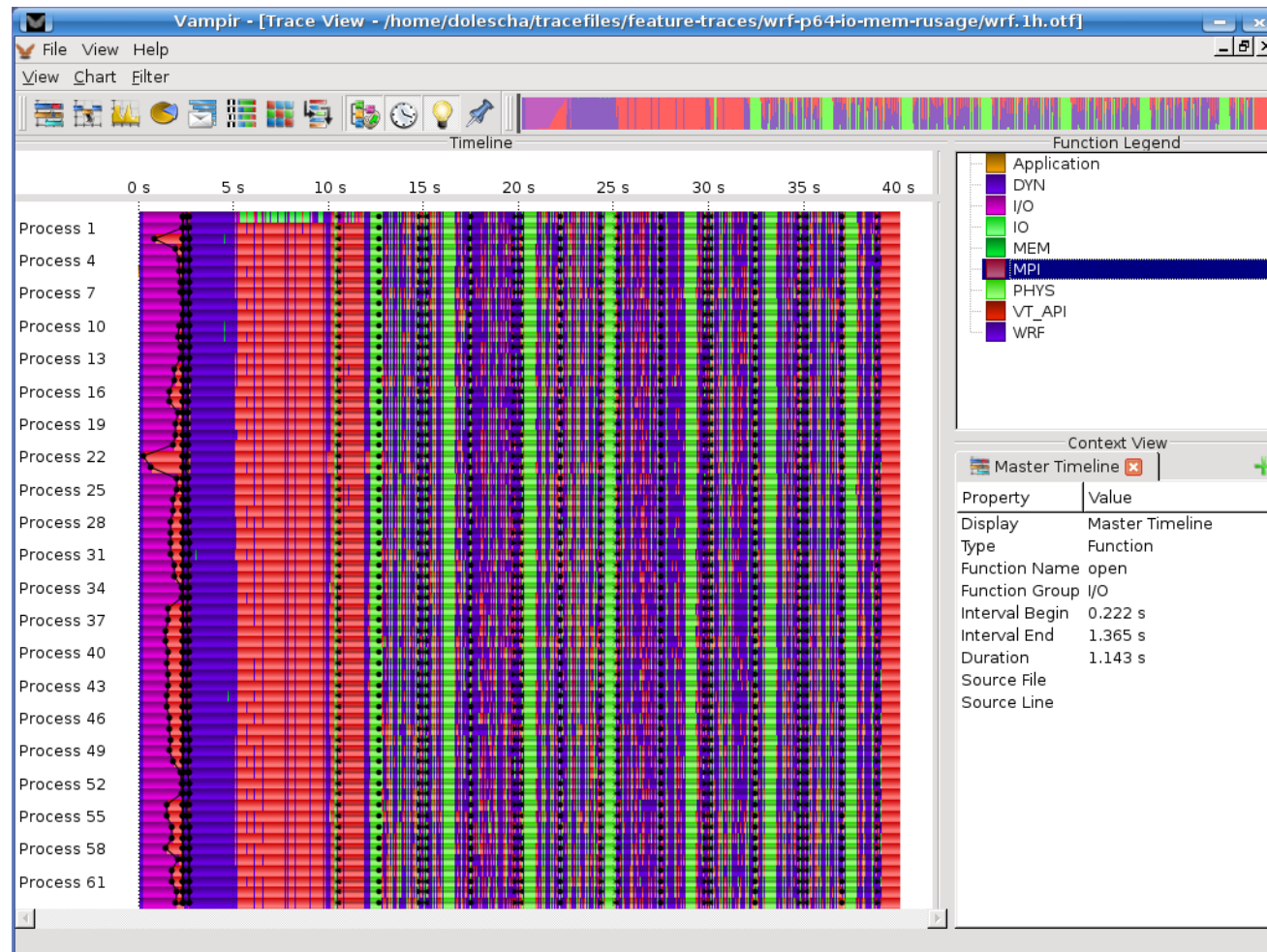
- Trace:
 - Records chronologically ordered sequence of events (e.g., enter/leave of a function, send/receive of a message)
 - Abstract execution model on level of defined events
 - Data volume dependent on runtime, scale, and level of detail
- Open-source examples:
 - Extrae / Paraver (BSC)
 - Score-P
- Commercial examples:
 - Intel Trace Collector & Analyzer

Tracing vs. profiling

- Tracing advantages
 - Event traces preserve the **temporal** and **spatial** relationships among individual events (👉 context)
 - Allows reconstruction of **dynamic** application behavior on any required level of abstraction
 - Most general measurement technique
 - Profile data can be reconstructed from event traces
- Disadvantages
 - Traces can very quickly become extremely large
 - Writing events to file at runtime causes perturbation
 - Writing tracing software is complicated
 - Event buffering, clock synchronization, ...

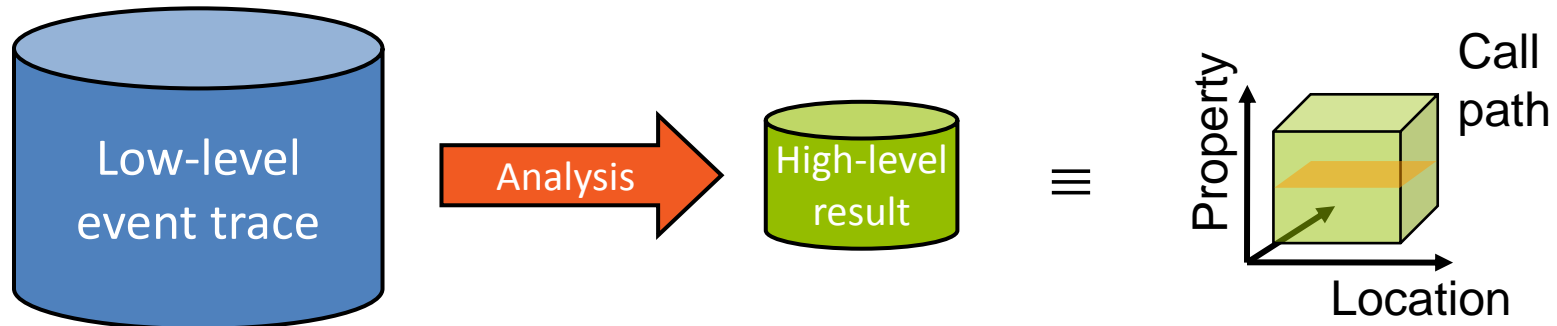
Visual trace analysis

- Allows for an in-depth analysis
- Here: Vampir (commercial)



Automatic trace analysis

- Idea
 - Automatic search for patterns of inefficient behavior
 - Classification of behavior & quantification of significance

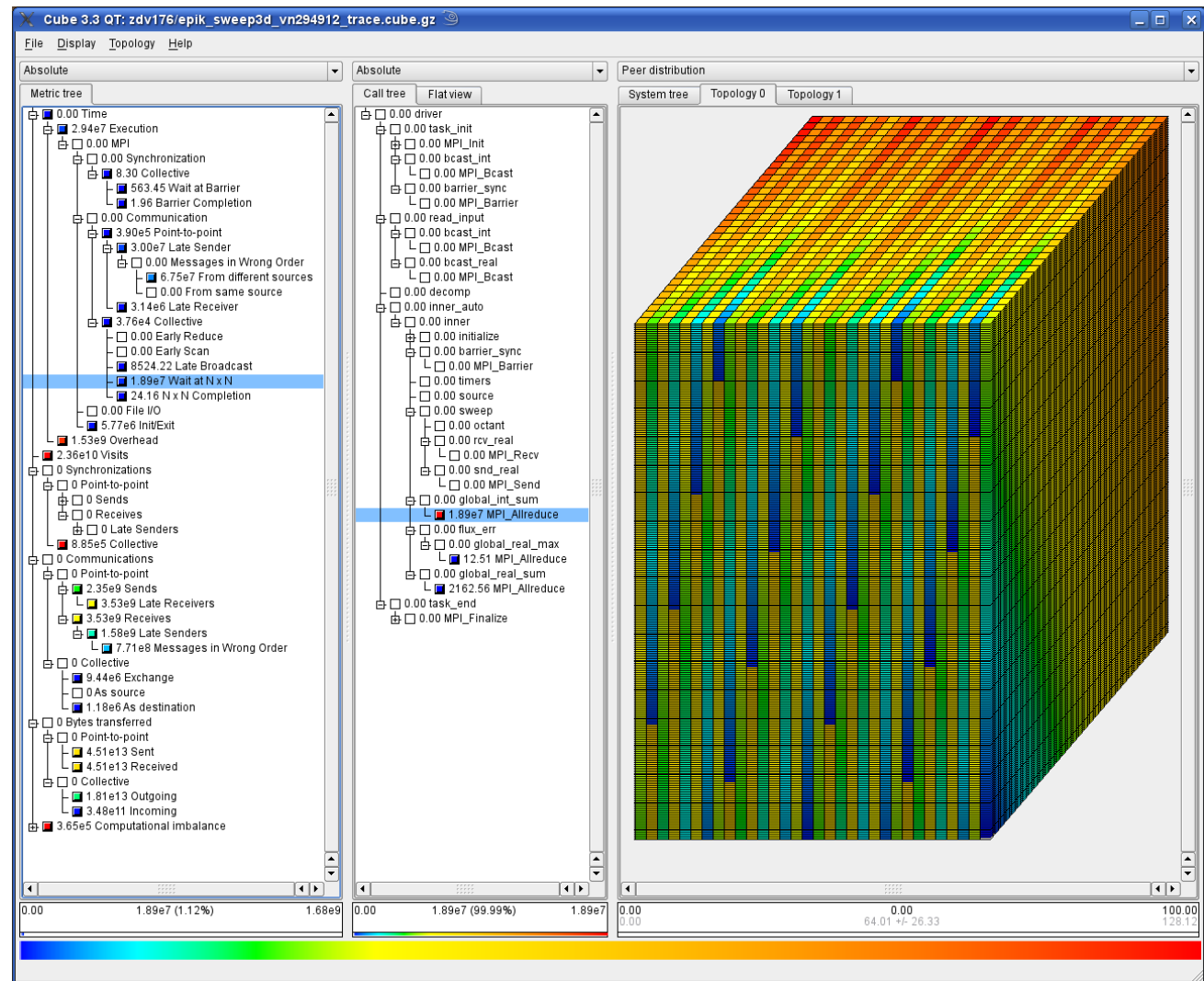


- Advantages
 - Guaranteed to cover the entire event trace
 - Quicker than manual/visual trace analysis
 - Helps to identify hot-spots for in-depth manual analysis
- Open-source example: Scalasca

Scalasca trace analysis sweep3D@294,912 BG/P

- 10 min sweep3D runtime
- 11 sec analysis
- 4 min trace data write/read (576 files)
- 7.6 TB buffered trace data
- 510 billion events

B. J. N. Wylie, M. Geimer,
 B. Mohr, D. Böhme,
 Z.Szebenyi, F. Wolf:
 Large-scale performance
 analysis of Sweep3D with
 the Scalasca toolset.
 Parallel Processing Letters,
 20(4):397-414, 2010.



Take away messages

Scaling is already important today

- TOP500 list November 2012:
 - Average system size: **29,772 cores**
 - Median system size: **15,390 cores**
- Machines will get even more parallel in the future
 - ☞ The most parallel systems of today will be medium-sized tomorrow

Use tools!

- Avoid do-it-yourself solutions
 - Tool developers collaborate with many people, e.g, compiler and platform vendors
 - They often know (and use) platform-specific tricks or undocumented features
 - E.g., low-overhead timers
- There are many powerful tools available
 - Often released as open-source
 - Often already installed on your HPC system (otherwise bug the center's support team 😊)

But: no single solution is sufficient...



☞ *A combination of different methods, tools and techniques is typically needed!*

- Tools focus on their specific strength to cope with technical challenges
- The tools community collaborates
 - Improved interoperability
 - Common data formats

Tool usage is not a one-off process

- Application codes evolve over time
- You need to verify whether code changes introduce new
 - Memory errors
 - Race conditions
 - Performance bugs
- Increasing the scale may reveal new bottlenecks
 - Don't guess based on tests at small scale
 - Perform analyses at production scale

Thank you!