D3.7 – Frameworks for Exascale Applications

WP3: Development Environment

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exa-scale computing, software and simulation

Due date:	M38
Submission date:	30/11/2014
Project start date:	01/10/2011
Project duration:	39 months
Deliverable lead organization	KTH
Version:	1.0
Status	Final
Author(s):	Xavier Aguilar, Stefano Markidis, Michael Schliephake (KTH), Alan Luis Cebamanos, Alan Gray, David Henty (EPCC), Alistair Hart, Harvey Richardson (Cray UK) Jens Doleschal, Tobias Hilbrich, Michael Wagner (TUD), David Lecomber (Allinea)
Reviewer(s)	Dmitry Khabi (HLRS), Lorna Smith (EPCC)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	05/08/2014	First skeleton version of the deliverable	Stefano Markidis (KTH)
0.2	15/08/2014	Added Alan's Contribution	Stefano Markidis (KTH)
0.3	17/09/2014	Added Luis' Contribution	Stefano Markidis (KTH)
0.4	19/09/2014	Added Michael and Xavier Contribution	Stefano Markidis (KTH)
0.5	26/09/2014	Added Tobias' contribution	Stefano Markidis (KTH)
0.6	30/09/2014	Added Jens and Michael Contribution	Stefano Markidis (KTH)
0.7	03/10/2014	Combining all contributions	Stefano Markidis (KTH)
0.8	04/11/2014	Changing Performance Monitoring part	Stefano Markidis (KTH)
0.9	25/11/2014	Responding to Reviewers	Stefano Markidis (KTH)
1.0	27/11/2014	Final version for submission	Lorna Smith (UEDIN)

Table of Contents

Index of Figures

Index of Tables

1 Executive Summary

This deliverable reports on the development, extensions and modifications to different frameworks that have been developed by CRESTA WP3 to enable efficient execution of parallel applications on exascale machines.

We describe first the development of a new framework, called “targetDP”, to express thread and instruction level parallelism for lattice-based codes. CRESTA participation in standardization committees, such as the MPI Forum and OpenACC and OpenMP committees, is briefly described.

We present a first mock-up implementation of the CRESTA DSL specification to enable automatic tuning of OpenACC codes.

The software architecture and the performance of two components (runtime administration and monitoring components) of the CRESTA run-time system are provided.

Extensions and modifications to the Score-P and Vampir performance monitoring and analysis tools are presented. To deal effectively with large amount of data from performance hardware counters, selective instrumentation and monitoring, hierarchical buffer management, runtime event reduction and message matching have been implemented. In addition, we report on how to handle file system limitations, to support performance monitoring for new programming systems and application using hybrid approaches, and how to monitor energy and network performance hardware counters.

Together with the extensions and modifications to Allinea DDT and Dresden Technical University MUST debuggers, we describe the integration of the MUST MPI correctness checker into Allinea DDT parallel debugger.

2 Introduction

The development and implementation of efficient computer codes for exascale supercomputers will require combined advancement of all development environment components: programming models compilers, automatic tuning frameworks, run-time systems, debuggers and performance monitoring and analysis tools. The exascale era poses unprecedented challenges [1]. Because the presence of accelerators is more and more common among the fastest supercomputer and will play a role in exascale computing, compilers will need to support hybrid computer architectures and generate efficient code hiding the complexity of programming accelerators [1],[2]. Hand optimization of the code will be very difficult on exascale machine and will be increasingly assisted by automatic tuners. Application tuning will be more focus on parallel aspects of the computation because of large amount of available parallelism. The application workload will be distributed over million of processes, and to implement ad-hoc strategies directly in the application will be probably unfeasible while an adaptive run-time system will provide automatic load balancing. Debuggers and performance monitoring tools will deal with million processes and with huge amount of data from application and hardware counters, but they will still be required to minimize the overhead and retain scalability.

In CRESTA WP3, we developed, extended and modified different frameworks to enable efficient execution of parallel applications on exascale machines. The CRESTA frameworks of the development environment are presented in the Figure below.

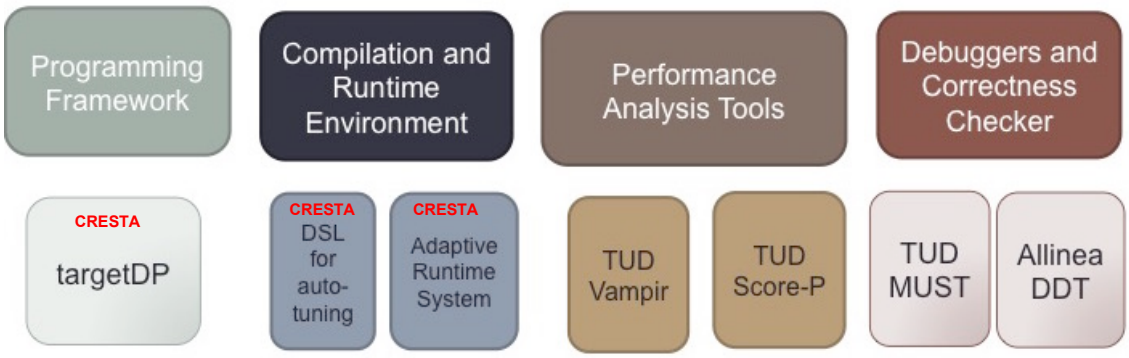


Figure 1: Different CRESTA frameworks for exascale applications

WP3 work was divided in four tasks:

- Programming models.
- Compilation and runtime environments.
- Performance analysis tools
- Debuggers.

The goal of programming framework task was to investigate programming models with exascale potential. An investigation of how to use effectively these programming models in applications is presented in CRESTA D3.11 (“Experiences with benchmarks and co-design applications”). In this deliverable, we present a new programming framework, called “targetDP”. CRESTA compilation and runtime environment task focused on the design of new Domain Specific Language for auto-tuning and of run-time system; the performance analysis tools task studied the extension and modification of performance monitoring and analysis tools of Technische Universitaet Dresden (TUD) (Score-P and Vampir). The debugger and correctness checker task focuses on the extensions of parallel debugger Allinea DDT and of MPI Correctness checker, TUD MUST.
In this deliverable, we report on the development, extensions and modifications to these frameworks that have been developed by CRESTA WP3 to enable efficient execution of parallel applications on exascale machines.The deliverable is organized as follows. The third section presents the work on programming models and describes

the standardization efforts in CRESTA. The fourth section presents the CRESTA DSL for auto-tuning and the CRESTA runtime system. The fifth section describes the modifications and extensions to the Score-P and Vampir performance monitoring and analysis tools. The sixth section describes the improvement in Allinea DDT and MUST MPI correctness checker. Finally, the seventh section concludes the deliverable summarizing the results.

2.1 Purpose

To goals of this deliverable are:

- To present the frameworks, developed and implemented in the CRESTA project, to support a productive and an effective development of exascale applications.
- To describe the targetDP framework for lattice-based codes
- To describe standardization activities by CRESTA membersin MPI Forum, OpenACC and OpenMP committees.
- To present the development of a mock-up version of the CRESTA DSL for auto-tuning of OpenACC codes
- To present the new CRESTA adaptive runtime system for exascale applications.
- To describe the enhancements and modifications to the performance monitoring framework, Score-P, and visualization and performance analysis tool, Vampir.
- To present modifications and extensions of the parallel debuggers and MPI correctness checker, Allinea DDT and MUST.
- To describe the integration of the MPI Correctness checker, MUST, into the Allinea DDT parallel debugger.

2.2 Glossary of Acronyms

AVX	Advanced Vector eXtension
D	Deliverable
DSL	Domain Specific Language
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Units
GTI	Generic Tool Infrastructure
IFS	Integrated Forecast System
ILP	Instruction Level Parallelism
IOSL	I/O Forwarding Scalability Layer
IPM	Integrated Performance Monitoring
MPI	Message Passing Interface
NUMA	Non Uniform Memory Access
OTF2	Open Trace Format 2
PGAS	Partitioned Global Address Space
PIA	Performance Introspection API
RMA	Remote Memory Access
SIMD	Single Instruction Multiple Data
TBON	Tree-Based Overlay Network
TLP	Thread Level Parallelism
UPC	Unified Parallel C
VVL	Virtual Vector Length
WP	Work Package

.

3 Programming Models

The trend towards the exascale is of increasing parallelism, partitioned into a hierarchy of levels within the hardware. At the most coarse-grained level, many nodes may be coupled via a high performance interconnect. Each node features one or more CPUs each with multiple compute cores. At the finest level, each core features a vector floating point unit, which can perform multiple operations per clock cycle. Furthermore, many systems now feature accelerators such as Graphics Processing Units (GPUs), on which computationally intensive kernels can be offloaded and executed with high efficiently on many low-power cores using high bandwidth graphics memory. Accelerators are used in conjunction with CPUs, and can result in additional complexity such as distinct physical memory spaces within a single application. The challenge for the programmer is to expose algorithmic parallelism in a way that maps on to the hierarchy of architectural parallelism. Ideally, this would be done in a way that optimises performance, but also allows intuitive expression of algorithmic content whilst promoting software maintainability across different systems such as those with and without accelerators.

3.1 TargetDP: thread and instruction level parallelism for CPU and GPU.

targetDP is a lightweight framework we have developed to target the data parallelism inherent in lattice-based applications to the hierarchy of hardware parallelism for either SIMD multi-core CPUs or NVIDIA GPUs. targetDP consists of a set of (C99) standard C preprocessor macros, and a small C library interface for set up and memory management. It therefore requires no new pseudo-language intermediate code, or compiler-like translation software layer.

The new abstraction promotes optimal mapping of code to hardware thread-level parallelism (TLP) and instruction-level parallelism (ILP), via the partitioning of lattice-based parallelism and translation to OpenMP or CUDA threads (for TLP) and perfectly SIMDizable parallel loops (for ILP). For large-scale parallel applications, targetDP may be used in conjunction with coarse-grained node-level parallelism, e.g. that provided by MPI. Thus, targetDP allows maintenance of a single source code base with portable performance on the majority of leading edge computational architectures. The programmer expresses the parallelism and memory management using targetDP functionality, and the relevant targetDP implementation can be chosen for a specific hardware platform at compile-time.

3.1.1 Memory management

Lattice based applications use ``lattice field'' data structures: arrays that have values (or sets of values) defined at every point on the lattice. The runtime of such applications is dominated by operations on lattice fields: these are data parallel in nature since they involve the same operation at all lattice sites. We use the terminology ``host'' to refer to the CPU that is hosting the execution of the application, and ``target'' to refer to the device targeted for execution of lattice-based operations. The target may be an accelerator such as a GPU or it may simply be the host CPU itself. It is an important aspect of our model that even in the case of the latter, we retain the distinction between host and target. We maintain both host and target copies of our lattice data, where the target copy is located in a memory space suitable for access on the target, and is treated as the master copy within those lattice-based computations. The host copy is located on the host memory, and is updated from the target copy as and when required to permit those (non computationally demanding) operations that should always be performed by the host. The targetDP library provides facilities to manage the host and target data structures. The basic functionality consists of memory allocation, de-allocation and copying. These operations map, in a straightforward manner, to the relevant CUDA operations in our GPU implementation, and to the equivalent C operations in our CPU implantation. For example, the `copyToTarget`

routine can map to either `cudaMemcpy` or `memcpy` at compile time, depending on the targetDP implementation selected. More sophisticated targetDP operations allow host/target value synchronization for only subsets of lattice data: this is important for minimisation of overheads in large-scale complex parallel applications. In these operations, compressed buffers are populated, transferred and uncompressed. Also included is functionality to utilize fast on-chip read-only memory (important for GPU performance).

3.1.2 Execution model

targetDP aims to expose the data parallelism inherent in the application in a way that can be mapped to the hardware efficiently. TLP will map to CUDA threads on a GPU or OpenMP threads on a CPU. When the target is an X86 CPU, ILP can be mapped to those vector instructions that extend the X86 set, such as 128-bit SSE, 256-bit AVX and 512-bit IMCI. ILP can similarly be mapped to equivalent vector instructions on other CPU architectures. On NVIDIA GPUs, exposure of ILP within a kernel can also be very beneficial, since it can facilitate latency hiding through use of fewer thread blocks, with more instructions per block.

Consider a simple example: the scaling of a 3-vector field by a constant, often used in Lattice-Boltzmann codes. This is, schematically:

```
//loop over lattice sites
for (idx = 0; idx < N; idx++) {

int iDim;
for (iDim = 0; iDim < 3; iDim++)
field[iDim*N+idx] = a*field[iDim*N+idx];

}
```

We can introduce targetDP by replacing the above code with the following function:

```
TARGET_ENTRY void scale(double* t_field) {

int baseIndex;

  TARGET_TLP(baseIndex, N) {

int iDim, vecIndex = 0;
for (iDim = 0; iDim < 3; iDim++) {
      TARGET_ILP(vecIndex) \
t_field[iDim*N + baseIndex + vecIndex] = \
t_a*t_field[iDim*N + baseIndex + vecIndex];
    }

  }
return;
}
```

For the C implementation, the `TARGET_ENTRY` macro holds no value, and the code will compile as a standard C function. For the CUDA implementation, it is defined as `__global__` to specify compilation for the GPU. We similarly provide a `TARGET` macro for use on subroutines called from `TARGET_ENTRY` functions. The `t_` prefix syntax is used to identify target data structures, where these can be managed using the library functionality described above. The function is launched in “host” code using additional targetDP syntax, which is trivial for the C implementation, but for CUDA specifies the relevant decomposition based on the lattice size.

We expose the lattice-based parallelism to each of the TLP and ILP levels of hardware parallelism through use of C-preprocessor macros in the following way. We re-express the original loop over lattice sites using the `TARGET_TLP(baseIndex,N)` macro, where `baseIndex` is an index for lattice sites, and `N` is the total number of lattice sites. The ``base'' terminology will become clearer below.

In our C implementation of targetDP, this macro is expanded as a loop over lattice sites, decomposed between OpenMP threads. Importantly, the TLP loop is strided in steps of a virtual vector length VVL: a tunable parameter that represents the width of

ILP that we wish to present to the hardware. Thus, each TLP thread operates not on a single lattice site but instead a chunk of VVL lattice sites, and `baseIndex` corresponds to the first index in the chunk. In other words, we are strip-mining the original loop.

For our CUDA implementation, it can be seen that this macro appears inside a kernel function and therefore expands as a CUDA thread lookup, where again a virtual vector length is used such that each CUDA thread becomes responsible for a chunk of lattice sites.

The lattice-based operation to be performed for the chunk of VVL sites is implemented using the `TARGET_ILP(vecIndex)` macro prepended to the innermost operation. The `vecIndex` variable is an integer that acts as an offset to the base index within the chunk of lattice sites. The operation that follows this macro can then use the combination `baseIndex+vecIndex` when accessing array data, ensuring that all elements of the lattice chunk are operated on. For C, VVL can be tuned to allow the compiler to generate optimal SIMD instructions. For example, setting VVL to *4m* will create *m* AVX instructions, where *m* is a small integer. *m=1* is an obvious choice, but it can be the case that *m>1* gives better performance. VVL can similarly be tuned for the CUDA implementation, giving latency hiding benefits.

The results of using targetDP in a lattice-based code (Ludwig code) are presented in CRESTA D3.11 ("Experiences with benchmarks and co-design applications").

3.2 CRESTA's standardization effort in the MPI Forum

The CRESTA project followed closely the standardization process in MPI. CRESTA participated in standardization meetings and contributed to discussions on new proposals in MPI.

A member of the EPCC partner in CRESTA, Daniel Holmes, regularly attended the MPI Forum meetings and served as both a Working Group Leader and a Chapter Committee Chair. CRESTA, together with the EPiGRAM project, has on-going involvement and influence within both the Hybrid Working Group and the Point-to-Point Working Group. Significant contributions have been made to the "endpoints" proposal as a direct consequence of the work carried out during the CRESTA project and the first year of the EPiGRAM project.

3.3 CRESTA's standardization effort for OpenACC and OpenMP

The OpenACC non-profit corporation develops and standardises the OpenACC API. Membership of the organisation is on an institutional basis, with single representatives from member and supporter organisations sitting on the Technical Committee. CRESTA was represented on the Technical Committee through the Cray (one if the founder members) and EPCC at the University of Edinburgh representatives.

OpenMP has a subcommittee developing a subset of OpenMP directives that can be used to accelerate applications in a similar manner to OpenACC. James Beyer (Cray US) co-chairs this subcommittee and CRESTA was also directly represented on this subcommittee through Cray UK staff attending the weekly telephone meeting.

4 Compilation and Runtime Environments

In this section, we present the development of CRESTA DSL and adaptive runtime system.

4.1 A Domain Specific Language for auto-tuning and exascale compiler support to exascale

Exascale systems are going to be inherently complex and we believe there is a place for more automated and intelligent software to build and run applications. One specific area (accelerators) illustrated this most aptly.

Accelerators and, in particular, GPUs have emerged as promising computing technologies which may be suitable for the future exascale systems. However the complexity of their architectures and the impenetrable structure of some large applications make the hand-tuning algorithm process more challenging and unproductive. On the contrary, auto-tuning technologies have appeared as a solution to this problems since it can address the inherent complexity of computer architectures.

Early in the project we engaged with the Institute for Computing Systems Architecture at Edinburgh University to tune Nek5000 kernels using a compiler-based machine-learning auto-tuning framework. The results were promising but in practice there were some important issues: firstly they could only deal with C but more importantly they were the only ones with expertise in their tools so we had to hand over our code for tuning. So although compiler-centric technologies were interesting we moved on to work with our own auto-tuning approach.

The CRESTA DSL implementation we developed can explore a tuning parameter space by repeatedly building and running an application. The best run is chosen using a metric from the program execution and currently is done by exhaustive search. The tuning session is control by DSL either from a global configuration file or embedded in application source. This DSL has been extremely useful for the auto-tuning process of the CRESTA co-design application Nek5000 and NekBone, a standalone benchmark for the Nek5000 code. Through different *scenarios* we have been able to explore the performance of Nek5000 using a wide range of parameter settings. Each *scenario* is able to pick the best values for a given set of tuning parameters. The tuning parameters will relate to build and runtime optimization choices that we can choose.

Nek5000 is an open-source code used to simulate incompressible flow with thermal and passive scalar transport. The code consists of 100,000 lines of code and it is many written in Fortran (70,000 lines of code) and C (30,000 lines of code), and uses MPI for message passing communication.

The CRESTA DSL was first used to optimise the performance of NekBone on a Cray XC30 system accelerated with Nvidia K20x GPUs. The results were impressive, giving a 200% speed-up over the default OpenACC parameter settings and over 15% speed-up over an exceptionally optimised OpenACC hand-tuned version. Then, similar work was extended to the full Nek5000 application. And although this was much more challenging due to the high number of code sections and different parameters to tune, the auto-tuner has been able to achieve performance improvements of around 32% compared to the best OpenACC hand-tuned implementation of Nek5000.

In addition, the autotuning mockup was used in two other situations by a member of the Cray performance team. The first was to explore vector lengths of specific parallel OpenACC loops in the Delta5d fusion code. The result was that only marginal gains could be found from such tuning. The second was to investigate performance of the IOR benchmark using various MPI_IO tuning parameters. This resulted in a better understanding of the important parameters for the particular scenario studied.

Once the DSL mockup was exposed to applications some features were added or enhanced. For example scenarios were implemented in the mockup and tuning parameters could be enabled given the context of a particular scenario. Also the

reporting of the results was improved with the addition of a new table of results, comparison with default parameters and new optional csv output format.

4.2 Hybrid and adaptive runtime systems

In order to achieve good application performance on exascale machines, highly system-specific features have to be exploited. This means that best practices in programming and software development have to be relaxed and the resulting code is difficult to port to different systems. Runtime systems help to build portable applications for a broad range of HPC infrastructures in a modular way[5]. The heterogeneous character of recent hardware as well as the parallel program's highly dynamic behaviour not known before their execution require runtime systems to take into consideration the hardware topology as well as monitoring information of the on-going program execution. The runtime system consists therefore of a resource manager, a library for runtime administration of parallel applications, and a performance monitoring and analysis tool. The design is based on a task-oriented programming model.

One of the hardest requirements in the development of simulation applications is their adaptation to different computer systems due to the varying technical parameters that have a huge influence to the numerical performance: cache- and memory hierarchies, the number of cores per CPU, the number of sockets per node, and the characteristics of the interconnect network.

Today, optimisations are typically implemented directly in the code. The necessary effort to do this would grow immensely in the future due to the increasing heterogeneity and diversity of HPC computer systems. A runtime system must aim to improve the performance portability that can be achieved with one certain implementation.

An important requirement for a tool development is the reuse of existing application codes often implemented in Fortran or C. The introduction of new software tools should allow its incremental adoption, keeping the need for reimplementation or adaptation of existing code to a minimum. A further requirement connected to the previous one is the wish that software tools support an adaptive use of best practices, which otherwise would not be applied due to prohibitive implementation effort. Finally, given that hybrid programming models gain more importance, the runtime system may not prevent the use of parallelisation technologies that it does not address itself.

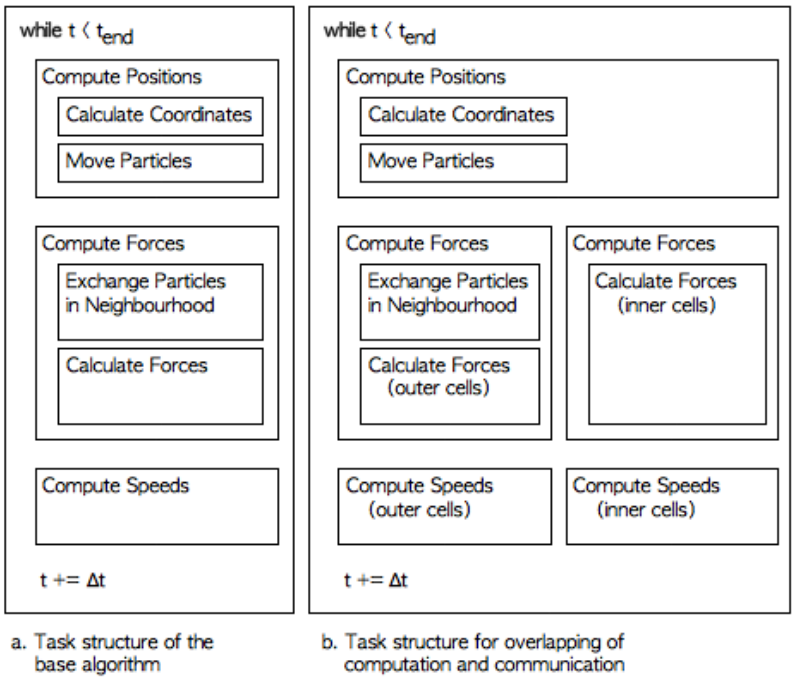
While different application classes put different requirements on runtime support, we focus on numerical simulation applications. Typical requirements of numerical simulations are

- Integration of data and task parallelism,
- Use of multi-level parallelism in the algorithm design,
- Development of algorithms with a high degree of parallel executable tasks, which have a moderate size, can be created very quickly, and avoid global communication operations,
- Usage of multi-threading, asynchronous communication and one-sided communication,
- Consideration of the increasing depth of the memory hierarchy,
- Optimised scheduling and mapping taking into account chip-architectures, memory hierarchies, internal communication abilities, etc. to provide a higher degree of parallelism and decrease memory and communication bandwidth usage.

4.2.1 Programming model and user interface

The runtime-system developed in CRESTA supports a task-oriented programming model featuring hierarchical multiprocessor tasks. Such tasks are computational units

that can be also parallel in themselves and can be subdivided hierarchically again into subtasks. The example in Figure 2 shows which tasks could be defined in a typical algorithm of a molecular dynamic simulation.



a. Task structure of the base algorithm

b. Task structure for overlapping of computation and communication

Figure 2: Example of an application structure using a coarse-grained hierarchical decomposition

The hierarchical nature of the computational tasks and their inner parallelism is clearly visible. Such a task model matches how programmers typically express parallelism during algorithm design and in program descriptions. The runtime system with its task model makes this parallelism explicitly visible in the source code, while traditional and widely used programming languages cannot easily express it.

Hierarchical multi-processor tasks allow to model also hybrid parallelisation and can be used to describe task parallel programs, i.e. sequences of operations, as well as data parallel programs like the parallel work on sub-domains. Hierarchical tasks reflect the hierarchical architecture of most computer systems well. Furthermore, their use integrates with existing programming models like MPI and OpenMP. Finally, the modelling of the simulation program as a task graph allows us to apply a broad range of algorithms from the theory of scheduling and graph partitioning for the mapping of tasks onto a computer system.

The runtime system uses currently a hardware performance model that models a computer system as graph from computational cores over nodes up to the complete system. The nodes of the graph are the computational cores respectively aggregations of them like dies, CPUs, cluster nodes, network partitions, and finally the whole cluster. They are characterised by their computational capabilities expressed through the computational performance. The edges of the hardware graph model the communication capabilities of the connection between the respective nodes. Shared caches or NUMA nodes connected to a few cores provide the fastest communication speeds. The speed then decreases for communication between different sockets, between nodes, and finally different parts of the cluster depending on the interconnect network topology. Therefore, this model does not only reflect the network topology but indirectly also the memory hierarchy within nodes.

The combined use of both models, the task-based programming model and the hardware model allows graph partitioning and mapping algorithms the selection of the most appropriate system part, i.e. a hardware model sub-tree, for the execution of a certain part of the application, i.e. a task sub-tree (see Figure 3). This will be explained in more detail in the description of the runtime administration component.

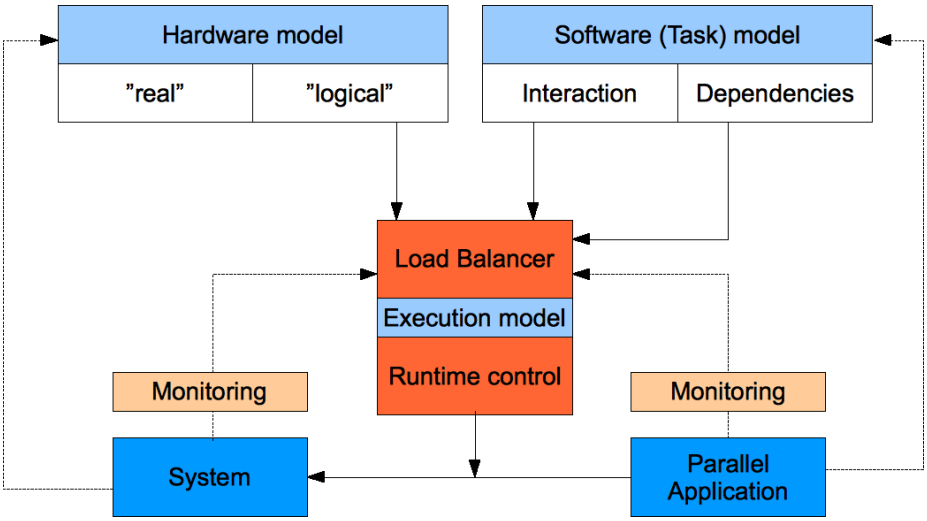


Figure 3: Use of the task and the hardware models to optimize the program execution

The user interface consists of five elements that will be used by an application developer. These comprise the definition of computational tasks, support for the automatic re-mapping of distributed arrays as well as user-defined data that define the state of a computational tasks, a control function to perform dynamic load-balancing, and the management of MPI communicators. This interface has been designed with simplicity in mind in order to allow its convenient introduction into existing simulation codes. Furthermore, the current implementation of the user interface to the runtime system focusing on the dynamic gradual improvement of task mappings and load-balance allows developers to optionally deactivate automatic load-balancing if a certain computer system cannot be well supported for some reason. This can be compared to a parallelisation similar to the OpenMP approach that allows compiling a program optionally with OpenMP support.

A task is identified by a key. The possibility to define tasks hierarchically leads naturally to a tree structure of tasks, examples of task keys are `"task_1"` or `"task_1/subtask_a"`. A number of parameters for a task can be specified in a structure `task_params`. These comprise the number of allocated processors, estimations of computational work and communication, and a callback function that is used to serialize respective de-serialize the task status. The following function can be used for the definition of the task tree.

```
task_tree *define_task(task_tree *parent, int n,
                       string *keys, task_parms *children);
```

One call to `define_task()` adds the number of `n` sub-tasks to the parent task that are allowed to run in parallel. Several calls to this function define a sequence of tasks. Arbitrary task trees may be constructed in that way.

The begin and end of the execution of a certain task can be registered in the code by calls to the functions `begin_task(key)` resp. `end_task(key)`.

It is necessary to transport the state of a computational task between processes in order to move tasks during the runtime. The application developer has to provide a function that can serialise resp. de-serialise the state of a computational task. The state of a task will be serialised in the owner process of a task, transported to the destination process, and finally de-serialised there. A ready-to-use convenience implementation is provided for the transport of arrays, which are one of the most frequently used data structures in numerical simulations. Arrays can be registered at their owning computational tasks and will be handled by the runtime system automatically. This avoids repeating coding tasks of serialisation for the developer as well as allows optimised handling of memory allocations. The prototype for callbacks is defined as

```
int (*task_serialisation_cb)(int opcode, void *buffer);
```

The value of `opcode` defines if the requested operation is a serialisation or a de-serialisation, and `buffer` is used to store the serialised data to write or read from. The registration of arrays at their owning tasks will be done with the function

```
void register_array(task_tree *owner, void *array,
                    int dtype, int n, int *dims);
```

that specifies the dimensions of the array and the type of its elements.

The runtime system supports the re-mapping and execution of computational tasks by means of MPI, whereas the choice of the inner parallelisation technique for multi-processor tasks is under control of the application developer. This would lead to the need of maintaining a directory of task mappings onto MPI ranks in order to perform communication between the owning processes of computational tasks in need of message-passing. The design choice for the runtime system was, to avoid explicit bookkeeping. MPI communicators will be used for that instead. Initially when setting up the calculation in typical numerical simulations, processes determine their communication partners rank-wise. These ranks are defined often as global properties of the MPI processes and updated occasionally, for example when re-distributions of data occur. In a program running under the control of the runtime system, however, the rank numbers of the communication partners become part of the state of computational tasks. They will be moved together with the other data defining the state of a task and used in all subsequent communication operations until an update is required due to re-distributions of data initiated by the simulation application itself. It is the responsibility of the runtime system to provide a MPI communicator to the application that reflects updated mappings of computational tasks onto MPI processes after their re-distribution. This functionality has been implemented by means of communicator management functions as provided by the MPI-2 standard. From an application developer point of view, the programmer defines a so-called load-balancing context that connects the group of a certain MPI communicator with a sub-tree of the task tree. Load-balancing will then be performed amongst the participating MPI processes of the context's communicator. The load-balancing context is defined by using the function

```
MPI_Comm *define_lb_context(MPI_Comm comm,
                            task_tree *root_task);
```

The function returns a MPI communicator for communication operations using the previously defined rank numbers of partner processes.

The runtime system monitors the execution of a parallel program. It is necessary from time to time to hand-over the control to the runtime system. A new task mapping is then calculated based on the previous monitoring. The callbacks specified during the definition of tasks will be activated for the serialisation and deserialization of tasks, and the runtime system manages the transport of these data between the processes. Finally, a new MPI communicator reflecting the new task distribution will be created and returned to the application for subsequent use in communication between the computational tasks. The user triggers these activities at suited points in time by calling the following function, which also returns the new MPI communicator to the application.

```
void perform_load_balancing(MPI_Comm *comm);
```

4.2.2 Software Architecture

The runtime system consists of three main components: a runtime administration component (Rta-C) schedules tasks and monitors their execution status; a monitoring component (Mon-C) provides information on the hardware utilisation, which is for scheduling decisions as well as to complement potentially incomplete or imprecise resource requirement specifications; and finally a performance analysis component (Pan-C) that analyses recorded monitoring data to provide more sophisticated hints for application control, beyond the capabilities of single run monitoring (see Figure 5). Implementations of the components Rta-C and Mon-C have been realised within CRESTA.

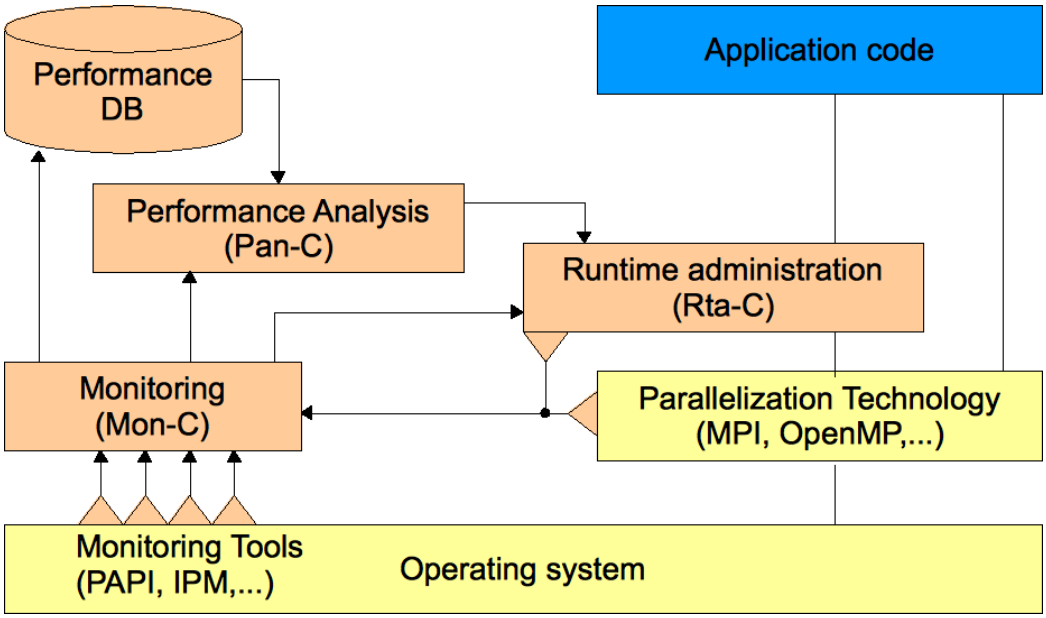


Figure 4: Components of the CRESTA runtime system

4.2.3 Runtime administration component (Rta-C)

Rta-C provides the user API allowing the definition of computational tasks as well as to control the load-balancing execution. It maintains internally the task tree as well as the hardware model. This component receives monitoring data from the monitoring component (Mon-C). Furthermore, it comprises the mapping algorithm as well as the functionality for moving tasks.

Rta-C creates the task tree within each process from the task definitions provided by the application. The cost estimates for computational work and communication volumes provided in the task definitions will be used for the calculation of task mappings on platforms that do not have the capability to monitor these parameters during the execution. Otherwise, these values will be replaced by data acquired by Mon-C as described below.

The hardware model is provided either as a static graph with weighted nodes and edges representing computing and communication capabilities or constructed dynamically during the program execution. The latter is done by measuring communication capabilities during the runtime of the parallel program. The advantage of this approach is that the real communication performance of the nodes allocated to a batch job is determined in the moment of the measurement. Influences from a concrete load on a HPC system as well as effects of dynamic routing configurations can be taken into account in this way. Even occasional updates of the hardware model are possible during long-running simulations.

Mon-C is clocked by the starting and end markers of computational tasks. It provides at least timing information about the execution of the computational tasks. Counter values of executed floating-point operations and MPI communications will be provided if available on the platform. Rta-C maintains a record of these measurements. This is for the time being a moving average value of a configurable number of time steps. The monitoring data are used to update the task definitions and provide in that way an up-to-date picture of the workload during the recent time steps.

The re-mapping of tasks in order to improve the load-balancing can be triggered by the application explicitly or automatically when a certain degree of load-imbalance has been reached. The wall clock time per time step is used as metric of load-imbalance.

The task mapping is calculated in a two-phase process that is illustrated in Figure 5.

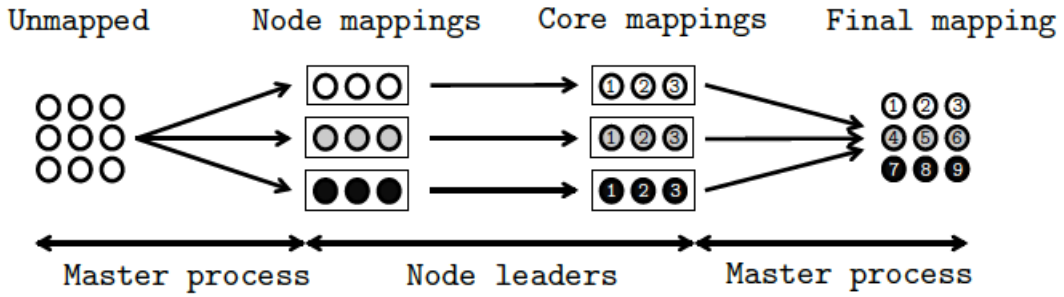


Figure 5: Two-phase calculation of task mapping

The implementation of this functionality has been based on the library SCOTCH, which provides extensible algorithms for graph partitioning and mapping[7]. The mapping of tasks to compute nodes is defined in the first phase. Compute nodes are represented for this calculation in the hardware graph as single nodes with a heavier weight according to the number of cores per node. The results of this calculation are task groups that will be assigned to one node. Afterwards, the mapping of task groups onto the different cores of a node is decided during the second phase. These calculations will be done in parallel on each node. The final task mapping is then distributed in order to allow the reconfiguration of the MPI communicator used in the load-balancing context.

For example, the mapping calculations for Cray XE6 systems that have been used in CRESTA, define the global hardware model with the network links as edges that connect the Gemini network ASICs, which are present in this graph as nodes summarizing the computational capabilities of the two servers that are connected to each of the ASICs. The local model comprises in fact both compute nodes that share one ASIC.

Figure 7 shows the wall clock time per time step for different task mapping strategies of a molecular dynamics simulation. The performance improvement of optimised task mappings in comparison to randomly chosen distributions is clearly visible. The molecular dynamics simulation for short-range potentials has been implemented as pure MPI application based on the linked cell method. It is characterised by neighbourhood communication and can benefit from an optimised task placement. Beside the communication topology, the task placement also has been balanced with respect to the overall number of particles of all MPI processes on a node. Such a cost function does not balance the work done by the different processes, of course. But, it supports that there will be about the same amount of data on each node. This implies that each node has to access about the same amount of data in main memory. Memory access, the bottleneck in memory-bound algorithms, is balanced in that way. The optimised task mappings allowed a reduction of at least 10% wall clock time per time step in all experiments. However, the improvement is of course dependent on the complex interplay of many factors coming from the task decomposition as well as from the properties of the hardware. The experiments confirmed the expectation from the design phase that the runtime system can manage well a large number of smaller tasks, whereas configurations with fewer, heavier tasks provide fewer possibilities for optimisations. The diagram given here is an example of such a job with smaller tasks and demonstrates performance improvements up to 50%. The experiments confirm the expectations that have been put up in the design phase. We expected there from experiments with separate components improvements between 15 and 50% as well as the introduction of about 5% overhead. This expectation has been met with overall performance improvements of at least 10% so far.

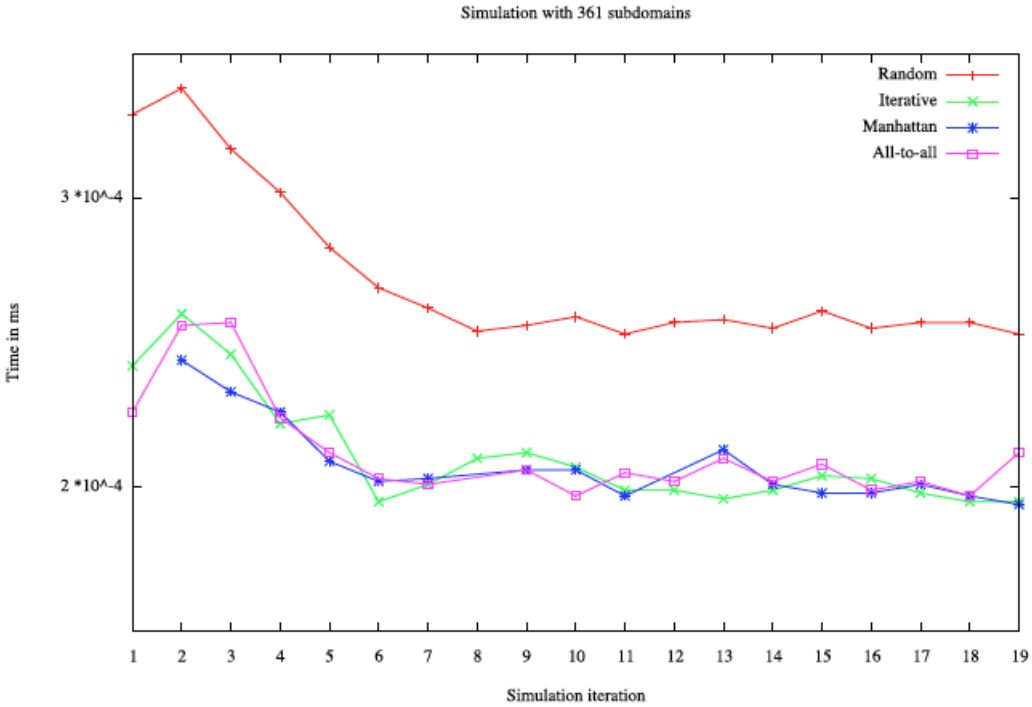


Figure 6: Time per-step for a molecular dynamics simulation using different strategies for task mappings

4.2.4 Monitoring component (Mon-C)

The monitoring component in the runtime system uses the Integrated Performance Monitoring (IPM) tool [8] to capture the performance behavior of MPI applications. IPM provides reports on several program events introducing minimum overhead. Such events can be MPI operations, Posix-I/O file operations, CUDA, or OpenMP events among others. IPM has been widely used by HPC centers such as NERSC to collect more then 310K batch profiles in the past 6 years.

CRESTA WP3 has extended the IPM monitoring tool with the Performance Introspection API (PIA) [9] to provide online feedback to the runtime system as the application runs. This API is designed to be a simple and lightweight interface written in C that can be used from C, C++, and Fortran. The Performance Introspection API provides each process a local view of its own performance behavior through the access to two different data entities, *user-defined code regions* and *activities*.

User-defined regions are measurement intervals defined by the runtime system within the application, for instance, tasks, functions, or blocks of code. These delimited regions can be nested and are annotated in the source code with the routine *ipm_region*. For each one of these regions the associated performance data is fixed and includes performance metrics such as wall clock time of the region, MPI time, the number of executed instances for that region, and hardware performance counters. As all these metrics are accumulated during program execution, the amount of memory needed to store them is small, in the order of a few kilobytes. The following code listing shows how to use the Performance Introspection API to access the total time, MPI time, and number of executed instances for a defined regions called *foo*.

```
void foo( )
{
  // Defining region start
  ipm_region( IPM_START, "foo");
  // Do whatever here
  // Defining region end
  ipm_region( IPM_END, "foo");
}
```

```
int main( int argc, char *argv[])
{
pia_regid_t id;      // Stores region ID
   pia_regdata_t data;  // Stores region data
foo( );
   // Obtain region ID
   id = pia_find_region_by_name("foo");
   // Obtain performance data for that region
   pia_get_region_data(&data, id);
fprintf( stderr, "%f Walltime\n", data.wtime);
   fprintf( stderr, "%f MPI time\n", data.mtime);
   fprintf( stderr, "%d times executed\n", data.count);
}
```

The other entity the runtime can access using the Performance Introspection API is activities. Activities are statistics associated to certain program events such as MPI calls, Posix-IO calls, or OpenMP phases. For instance, the runtime can consult the activity MPI_Recv, obtaining the total number of times the call has been executed, total time inside the call, maximum and minimum execution time, or number of bytes received for the whole run or for a certain defined region. Activities are accessed through their activity ID as shown in the following code snippet:

```
// Activity name
chat *act1 = "MPI_Send";
// Activity ID
pia_act_t id;
// Activity data
pia_actdata_t data;
// Acces the data
pia_init_activity_data(&adata);
id = pia_find_activity_by_name(act1);
pia_get_activity_data(&data, id);
fprintf(stderr,
        "MPI_Send happened %d times and transferred %d bytes,
        adata.ncalls, adata.nbytes);
```

The efficiency and overhead introduced by the monitoring component and the Performance Introspection API has been tested with several experiments.

The first experiment measured the time for accessing performance data associated to user-defined regions by using a synthetic application that creates thousands of user-defined regions, and accesses the performance data of one of them randomly chosen. Figure 6 shows the average access time to a random region using the API as the number of defined regions increases on an AMD Opteron 6274 at 2.2 Ghz. As it can be seen, the access time is small, not exceeding 300 ns for 20,000 regions. Moreover, the access time is almost constant for the first 8,000 regions where all the measurements are in the range of 238-248 ns. This small variability in the measurements was caused by the nature of the experiment in which the region selected was not always the same but randomly selected.

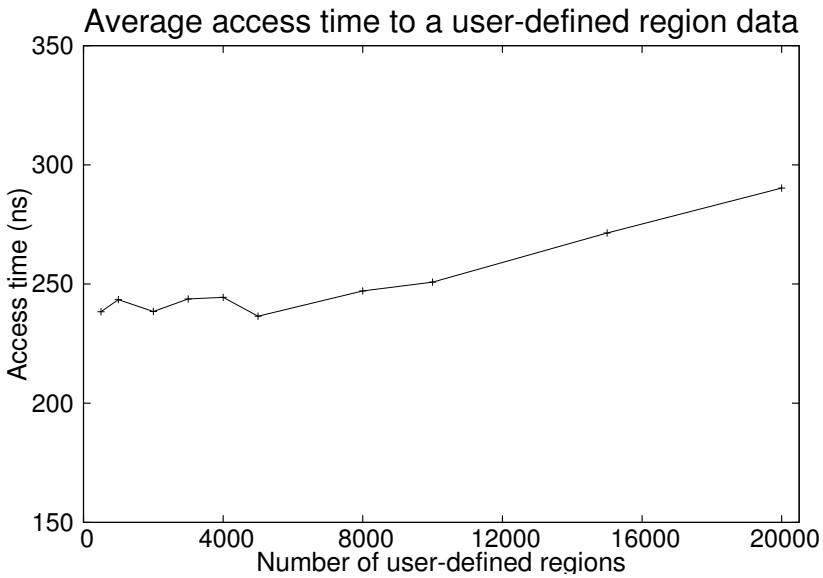


Figure 7: Average time in nanoseconds for accessing the performance data associated to a user-defined region

The second experiment performed measured the access time for activities. The data used for computing activities is stored within IPM in a hash table as explained in [10]. Thus, a benchmark was designed to test the access time for an activity as the hash table fills up. This benchmark stores 10 millions random events in a hash table of 32K entries increasing on each step the number of unique keys used. Figure 7 provides the time in microseconds for accessing the activity MPI_Send among 10 million random events as the number of unique hash keys increases. The experiments were performed again in an AMD Opteron 6274 based system at 2.2 Ghz. As it is shown in the figure, the access time increases slightly as the hash table fills up and collisions become more probable (more unique keys leads to more entries used in the hash table). However, as pointed in [10], in the vast majority of applications the observed fill rate for the hash table is below 50%, being their access time between 800 and 850 microseconds.

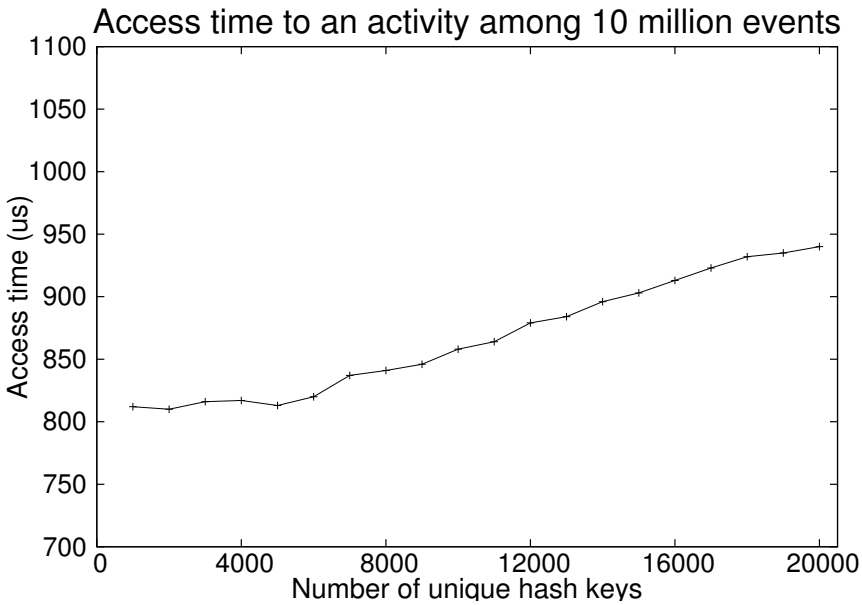


Figure 8: Access time in microseconds to an activity among 10 millions of events in the activity hash table

Finally, the aim of the last experiment was to measure the total overhead introduced by the monitoring component and the Performance Introspection API when used in conjunction with a real application. In this experiment, we used Sweep3D [11], a solver that models 1-group-time-independent discrete ordinates (Sn) 3D Cartesian (XYZ) geometry neutron transport problem. Using the Performance Introspection API we checked on each iteration of the program main loop the total time, the average time, and the number of executed instances of MPI_Send and MPI_Recv. We also accessed the wall clock time, MPI time, and number of instances for the function *sweep*.

We run the benchmark in a Cray XE6 system with AMD Opteron 12-core ”Magny-Cours” (2.1 Ghz) processors. Each node had 24 cores divided between 2 sockets and 32GB of DDR3 memory. The nodes were interconnected with a Cray Gemini Interconnect network. The benchmark was run several times using a base grid of 10x10x400 with weak scaling up to 8,160 processors. Table 1 shows the execution times for Sweep3D with and without the monitoring component and the Performance Introspection API. It can be observed that the overhead introduced does not perturb the application, fluctuating always under 1% due to the natural runtime variations in HPC systems. It is also noticeable that the overhead does not increase with the number of cores as the Performance Introspection API operates locally and does not require any communication between processes.

Table 1: Total execution time for the Sweep3D benchmark and percentage of overhead introduced by the monitoring component (IPM) and the Performance Introspection API.

MPI processes	Original Sweep3D	Sweep3D with IPM + PIA	Overhead
1032	226.1 s	225.768 s	0%
2064	244.975 s	245.437 s	0.19%
4080	267.72 s	269.448 s	0.65%
8160	306.751 s	308.234 s	0.48%

5 Performance analysis tools

Event tracing tools record parallel applications in detail by logging runtime events with a precise timestamp and further event specific metrics. This allows capturing the dynamic interaction between thousands of concurrent processing elements and enables the identification of outliers from the regular behavior. While single events are rather small, event-based tracing frequently results in huge data volumes. In particular, tiny and often used functions such as get/set class methods or helper functions can easily overwhelm any recording trace buffer. Due to the instrumentation all functions that are usually inlined by the compiler are executed and monitored. By itself this provides a very detailed view on an application's behavior. However, if tiny functions are heavily used – like in C++ applications – monitoring such can result in tremendous data volumes and runtime overhead.

In a study including Gromacs [13] we showed that for many applications short running functions contribute 90-99% of the recorded data volume. In the same time, these tiny functions call contribute very little to the analysis and overall understanding of the application behavior. To handle these tiny short-running function calls, in particular, for long running applications we developed the following strategies.

5.1.1 Selective Instrumentation

Since automated instrumentation techniques are most convenient and easy-to-use, the majority of event tracing tools use compiler instrumentation as their default to define events[14][15][16][17]. However, compiler instrumentation automatically instruments all functions regardless of their size and whether they would be inlined. By comparing the set of symbols from the original application (A) with the set of symbols of the fully instrumented application without any symbols from the monitoring system (B).The set of originally inlined functions (I) is the difference of set A from B. By excluding the originally inlined functions from the instrumentation not only the resulting data volume is tremendously reduced but also the runtime overhead of the monitoring. With this approach it was possible to reduce the monitoring overhead for Gromacs by a factor of 3.5 and the resulting trace size by a factor of 400 [13]. In addition, it is also possible to use a profiling run to determine the most often called functions and their average duration. This information can be used to exclude additional functions from the instrumentation (see also [13]).

Hybrid applications that use multiple paradigms can also be instrumented to record only a subset of aspects or paradigms. A hybrid version of Gromacs that uses massage passing (MPI), threading (OpenMP), and accelerators (CUDA) was instrumented to record different subsets of all parallel paradigms. Depending on the number of paradigms the resulting trace size was reduced to 2.8% for recording only MPI and CUDA [18]. It is also possible to apply different levels of details to different nodes of the application to get a complete view over some nodes and a coarse view over the remaining nodes. Figure 9 shows a visualization of Gromacs using different levels of detail for each node and the according trace size reduction (the reduction is always relative to the corresponding node with full instrumentation).

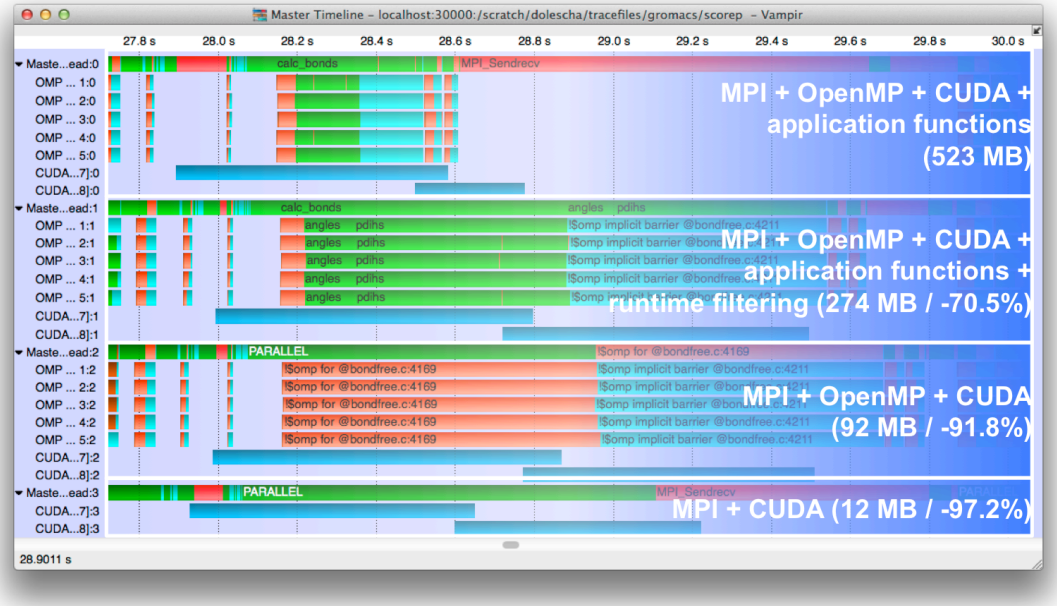


Figure 9: Vampir visualization of Gromacs with different levels of detail for each node. Reduction is relative to the corresponding node with full instrumentation.

5.1.2 Selective Monitoring

Selective monitoring is another approach to decrease the number of collected events. The difference to selective instrumentation is that all code regions of an application are instrumented and recorded but only a subset is finally stored. We focused on two defined code regions: iterations and functions.

In iterative applications it is reasonable to avoid storing every single iteration because most of them show more or less the same behavior. The selection of these iterations can be done statically or dynamically depending on certain parameters. The first method is to statically define which iteration is recorded and stored, e.g., every 10th or 100th iteration. With this it is still possible to analyze the behavior over time but the amount of recorded data is reduced to ten or one percent, respectively. However, iterations with either interesting behavior or a performance problem might be lost. The second method is to record every iteration and dynamically decide whether it is stored or discarded by evaluating its behavior, e.g., only store an iteration when its runtime varies from the average runtime by a defined threshold. To realize such a subsequent removal of iterations we developed and applied a rewind method to rewind the recorded event stream to any pre-defined point (e.g. the beginning of the current iteration), which eliminates everything record after that point. Figure 10 shows an example visualization with Vampir of Gromacs where only every 5th iteration block is stored. This topic is also covered in more detail in [19].

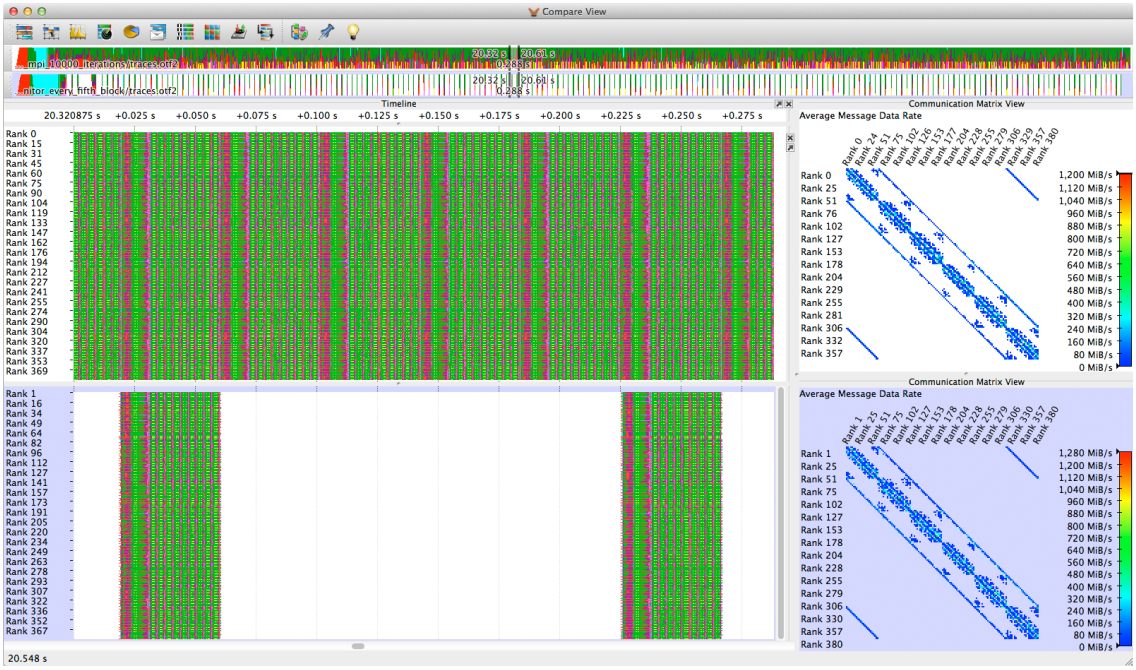


Figure 10: Gromacs on 144 processes fully monitored (top) and selectively monitored (bottom) event trace visualized with Vampir zoomed in to about 6 iteration blocks (Source: [19]).

Next to iterations, specific functions calls can be discarded from the trace. This approach also targets short-running highly frequent functions calls as in Section 5.1.1 but instead of removing an entire function from the trace only individual calls to a function are removed. Besides the trivial approach to stop recording a function after it is called a pre-defined amount of times, we developed an approach that evaluates the duration of each function call and only stores it when its duration is longer than a pre-defined minimum duration. This approach effectively discards all short-running function calls while still keeping the outliers that are of interest for a performance analysis.

We applied a minimum duration of one microsecond, i.e., all function calls shorter than one microsecond are filtered. This way, all short-running functions are eliminated while all important routines including all communication routines remain in the trace. For all applications that heavily use short-running functions the trace sizes can be remarkably reduced down to 0.1% of the original trace size. For Gromacs, this approach reduced the trace size to about 1.7% while still keeping the coarse program behavior [13].

Figure 11 and Figure 12 show the resulting event trace visualized with Vampir. The fully monitored measurement can be seen on the upper half of each Figure (white background), the measurement with duration filtering on the lower half (blue background). Both figures demonstrate that the filtering of short-running functions does not alter the general application behavior; except for the missing short-running functions. The function summary in Figure 11 shows that the total number of function calls is reduced from about 4 billion to 68 million. Figure 12 additionally shows the process timeline of process zero in detail with the calling depth on the vertical axis. The process timeline demonstrates that the highly frequent function calls on calling depth 10 and 11 are effectively eliminated while the outliers that run longer are still contained in the trace. This topic is also covered in more detail in [13].

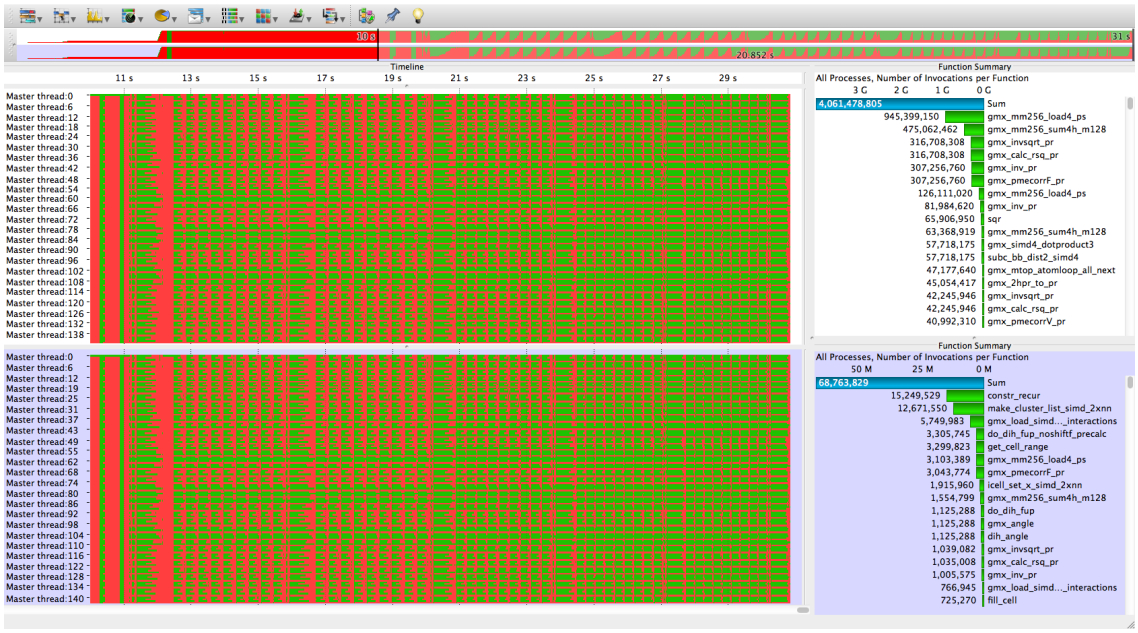


Figure 11: Gromacs on 144 processes fully monitored (top) and selectively monitored (bottom) event trace visualized with Vampir (Source: [13]).

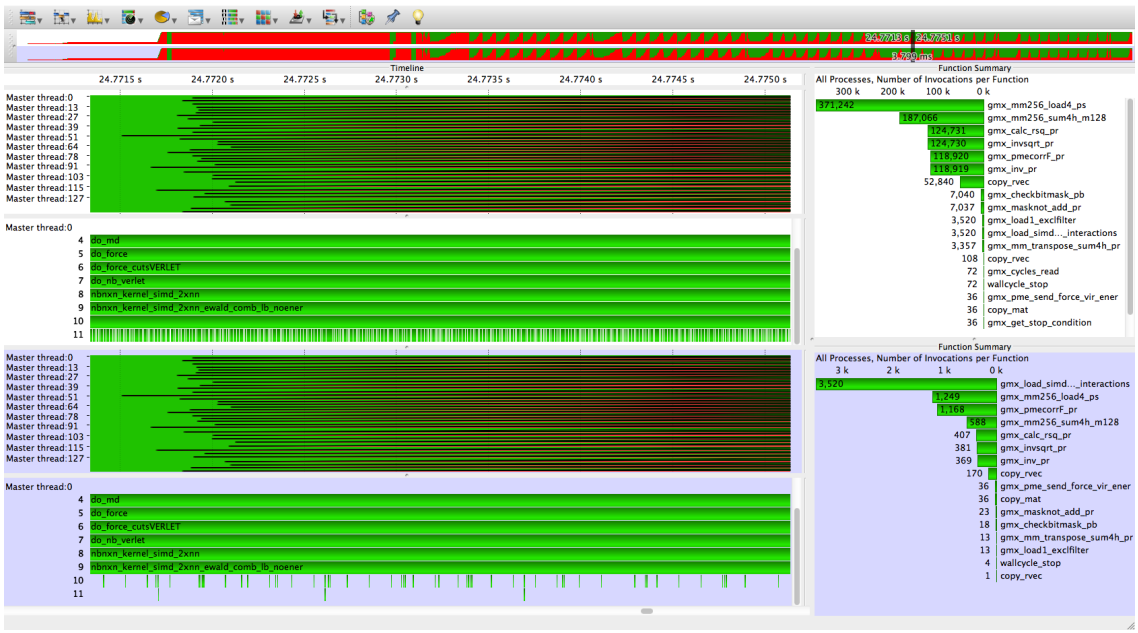


Figure 12: Gromacs on 144 processes fully monitored (top) and selectively monitored (bottom) event trace visualized with Vampir zoomed to an application phase of about 3.8ms (Source: [13]).

5.1.3 Hierarchical buffer management and runtime event reduction

To support the efficient elimination of short-running function but also further techniques we applied a hierarchical memory buffer [20]. Instead of one flat continuous memory buffer the hierarchical memory buffer uses additional hierarchy information such as call stack depth or event class to sort events in a multi-dimensional array. This provides the opportunity to remove events of a specific hierarchy class very efficiently. For example, the elimination of short-running function calls benefits from this memory buffer layout because all function enter and exit events are sorted by their according calling depth and are separated from all other types of events. Thus, the according enter and leave event of a function call are guaranteed to be right next to each other in the memory buffer and, therefore, can be easily and efficiently removed.

In addition, the hierarchical memory buffer allows further techniques to reduce the number of events within the memory buffer. It is particularly designed to keep an entire measurement with in a fixed-sized single memory buffer to avoid the bias introduced by intermediate memory buffer flushes. Moreover, it entirely avoids file system interaction, which provides a solution to the limitations in parallel file systems (see Section 5.2.1). Further techniques that reduce the number of stored events whenever the memory buffer is exhausted are described in detail in [21]. These techniques include, for instance, a reduction of events by their calling depth or their event class to reduce the level of detail for function calls or different paradigms automatically during runtime.

5.1.4 Message matching

While the approaches in Section 5.1.2 and 5.1.3 are essential to reduced the number of recorded event, unfortunately, the analysis of the communication behavior may partly or completely fail when even a single specific event is missing; especially MPI [10]. The basis of a correct post-mortem communication analysis is the correct matching of send and receives calls of each message. This can be done either by a replay of the communication based on the recorded MPI events or by matching MPI send and receive events by their order of occurrence [16][14]. In both ways, whenever multiple MPI messages have the same communicator and message tag the associated events can only be matched by their order of occurrence, e.g., first send event with first receive event and so on. Consequently, if one send or receive event is missing, the correct matching of send and receive events and, therefore, the post-mortem communication analysis fails. But, a correct post-mortem communication analysis is essential to understand complex communication behavior and to identify performance problems. In addition, all metrics derived from MPI events like latency or bandwidth rely on a correct matching, as well. Therefore, it is quite unsatisfying, to lose all this analysis options by dropping even a single event.

To circumvent those restrictions, we developed an approach to make each MPI event distinguishable from others with the same communicator and message tag by introducing an unique sequential message identifier. With this approach it is possible to clearly identify, which MPI events are missing and, thus, it is possible to correctly match MPI send and receive calls even with missing MPI events. With this, it will become feasible to apply the described selective monitoring techniques without sacrificing a detailed communication analysis.

In [19] we described and evaluated this approach in detail and demonstrated that the slight increase in memory allocation and overhead due to the additional information is more than justified considering the tremendous reduction that can be achieved with the mentioned selective monitoring and runtime event reduction approaches.

The approach and implementation are based on the Open Trace Format 2 (OTF2) [22], a state-of-the-art Open Source event trace library used by Score-P and the performance analysis tools Vampir, Scalasca, and TAU [14][16][17]. Nevertheless, the methods can also be generalized on other event based tracing libraries.

5.2 Exascale challenges

In this section, we present the exascale challenges in dealing with file system limitations, with new programming paradigms and hybrid applications and with monitoring energy and interconnection network performance.

5.2.1 Dealing with file system limitations

Post-mortem performance analysis techniques have to handle the amount of information of a whole measurement run and usually store this information in entire on the parallel file system. At an exascale level, creating one file per measured processing element results in disaster for parallel file systems. Current file systems can create only a few thousand files per second [23]. Two approaches that are dealing with the file system limitations and are applied to event tracing are SIONlib and the I/O Forwarding

Scalability Layer (IOFSL) [24][25]. Both approaches try to merge many logical files into a single or a few physical files. While SIONlib relies on the file system's capability to handle large sparse files to pre-allocate segments for the logical file handles within a single file, the I/O Forwarding and Scalability layer, as the name suggests, provides an I/O forwarding layer to offload I/O requests to dedicated I/O servers that can aggregate and merge requests before passing them to the actual file system.

Both approaches have proved to support monitoring at high scales. VampirTrace successfully recorded a full system run on the Jaguar system at Oak Ridge National Laboratory with 200.000 processes and Scalasca used SIONlib to record a full system run on the JuGene system at the Jülich Supercomputing Center with almost 300.000 processes [25][24].

Since version 1.0 Score-P supports the usage of SIONlib but was restricted to pure MPI applications. With the upcoming release, Score-P 1.4 will support hybrid programs including accelerators, as well.

5.2.2 New paradigms and hybrid applications

5.2.2.1 CUDA and OpenACC

As mentioned in the Section 3, in the last years CUDA/OpenACC capable devices became more and more popular in the High Performance Computing area since they are promising more floating point operations per seconds than a typical CPU will ever provide in a user application.

Monitoring of CUDA applications can be done either via the CUDA Profiling Tools Interface (CUPTI) or by a library wrapping approach. CUPTI provides different APIs that can be used to get insight into the CPU and GPU behavior of CUDA applications. The benefits of CUPTI in comparison to the library wrapping approach are the reduced perturbation of the kernel execution and precise event (kernel) time information. An exemplary study with the Nek5000 benchmark can be found in [27].

Since version 1.3 Score-P is able to monitor CUDA activities via CUPTI and OpenACC activities via a shared library wrapping approach. The use of the new developed generic one-sided RMA event model allows us to monitor memory transfers between host and graphic card as one-sided communication.

5.2.2.2 SHMEM

SHMEM is a PGAS paradigm very similar with one-sided paradigm in MPI to pass data between cooperating parallel processes on logically shared distributed memory.

The one-sided communication operations of Cray SHMEM can be easily recorded with Score-P's generic RMA event model. Cray SHMEM allows the coexistence of MPI and, therefore, initialization and finalization of the measurement system can be easily used with the MPI interface of Score-P. The instrumentation can either be done by a library wrapping approach or by defining weak symbols and use of the strong symbols provided by the Cray SHMEM library, this approach is very similar to the PMPI interface of MPI. Figure 13 shows the visualization of the performance monitoring of a Cray SHMEM application.

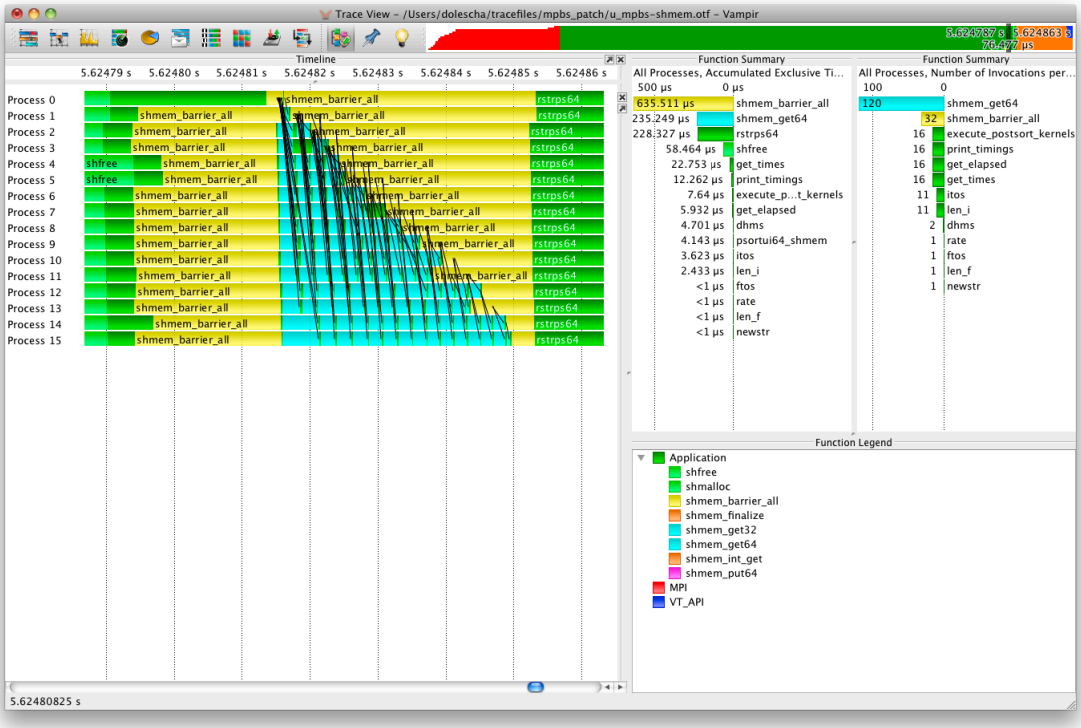


Figure 13: Performance visualization of a massive parallel bucket sort parallelized with Cray SHMEM. The master timeline shows very impressively the master-slave communication implemented with shmem_get64 operations colored in light blue between the different processes surrounded by tow shmem_barrier_all operations colored in yellow.

5.2.2.3 Support for hybrid applications

Score-P supports the tracing of most of the parallel paradigms and standards including MPI, SHMEM, OpenMP, Pthreads, CUDA, and OpenCL/OpenACC. In addition, all this parallel paradigms can be recorded simultaneously.

We monitored Gromacs with various numbers of threads of execution and different parallel paradigms from pure MPI applications to hybrid versions of Gromacs using MPI, OpenMP, and CUDA (see Figure 14).

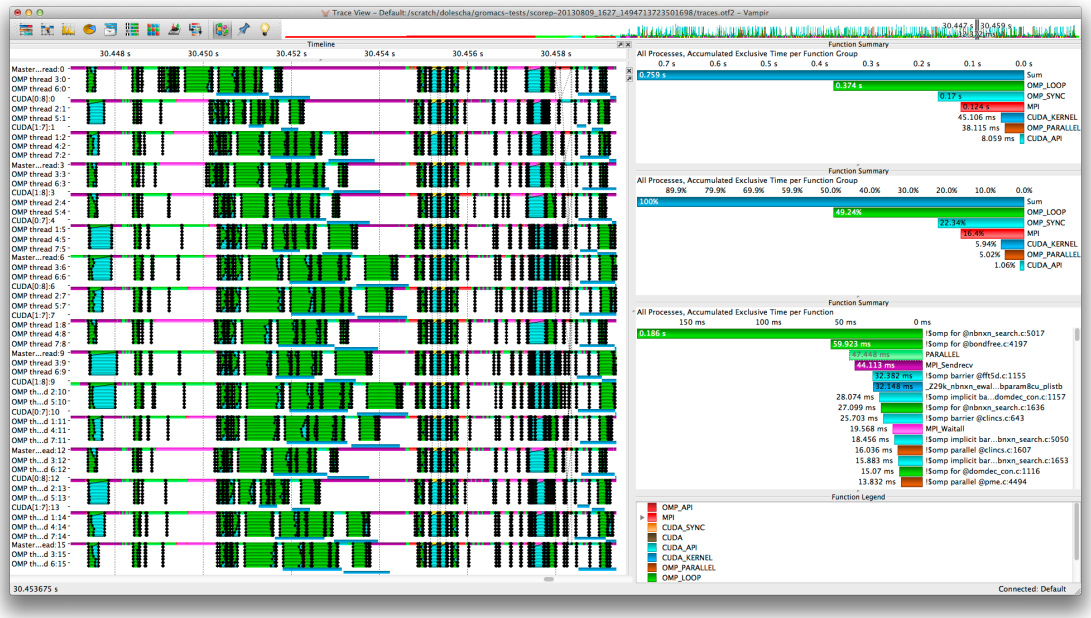


Figure 14: Performance visualisation of a hybrid version of Gromacs parallelized with MPI, OpenMP, and CUDA running on 8 nodes with 16 cores and 2 Nvidia K20 each. Every process uses one Nvidia graphic card and sends its kernels coloured in blue to two different streams.

5.2.3 System behavior: energy and network

Energy and power consumption are increasingly important topics in High Performance Computing. Wholesale electricity prices have recently risen sharply in many regions of the world, including in the European states, prompting an interest in lowering energy consumption of HPC systems. Environmental (and political) concerns also motivate HPC data centers to reduce their “carbon footprints”. This has driven an interest in energy-efficient supercomputing, as shown by the rise in popularity of the “Green 500” list of the most efficient HPC systems since its introduction in 2007.

However, energy efficiency goes beyond hardware design. Delivering sustained but energy-efficient performance of real-world applications will require software engineering decisions, both at the system-ware level but also in the applications themselves. Such application decisions might be made when the software is designed or at runtime via an auto-tuning framework.

For these to be possible, fine-grained instrumentation is needed to measure energy and power usage not just of overall HPC systems but also of individual components within the architecture. This information also needs to be accessible not just to privileged system administrators but also to individual users of the system, and in a way that is easily correlated with the execution of their applications.

Score-P has been able to record external generic and user-defined hierarchical performance counters since version 1.2. This is done with a flexible “metric plugins” interface to address the complexity of machine architectures both today and in the future. The metric plugin interface provides an easy way to extend the core functionality of Score-P to record additional counters, which can be defined in external libraries and loaded at application runtime by the measurement system. We built a Score-P metric plugin to monitor the application external energy and power information. Figure 15 shows the visualization of Gromacs with energy and power consumption for host and accelerator in Vampir. A monitoring study of energy and power consumption on Cray supercomputers was done in [28].

Figure 15: Color-coded visualization of 4000 iterations of a hybrid version of Gromacs running on four nodes (with each node hosting one MPI process with six CPU threads and two GPU CUDA streams running on the accelerator) with according timelines for the events on all four nodes (topmost) and corresponding energy (second timeline), instantaneous power (third timeline), average board power derived from energy (fourth timeline), instantaneous accelerator power (fifth timeline), average accelerator power derived from accelerator power (lowest timeline) for the four nodes, and according statistics for the exclusive time on the right part of the figure.

With systems getting larger and more complex, networks within HPC systems are getting more and more complex as well. Since network problems or high network load can tremendously affect the behavior of parallel applications it is important to enable an analysis of the correlations between network and application behavior.

Similar to external energy counters, network statistics and counters can be monitored and integrated in an application trace with the Score-P metric plugin interface by using an according plugin that calls PAPI interface asynchronously per node. Figure 16 shows the correlation of application behavior and network activity with Vampir.

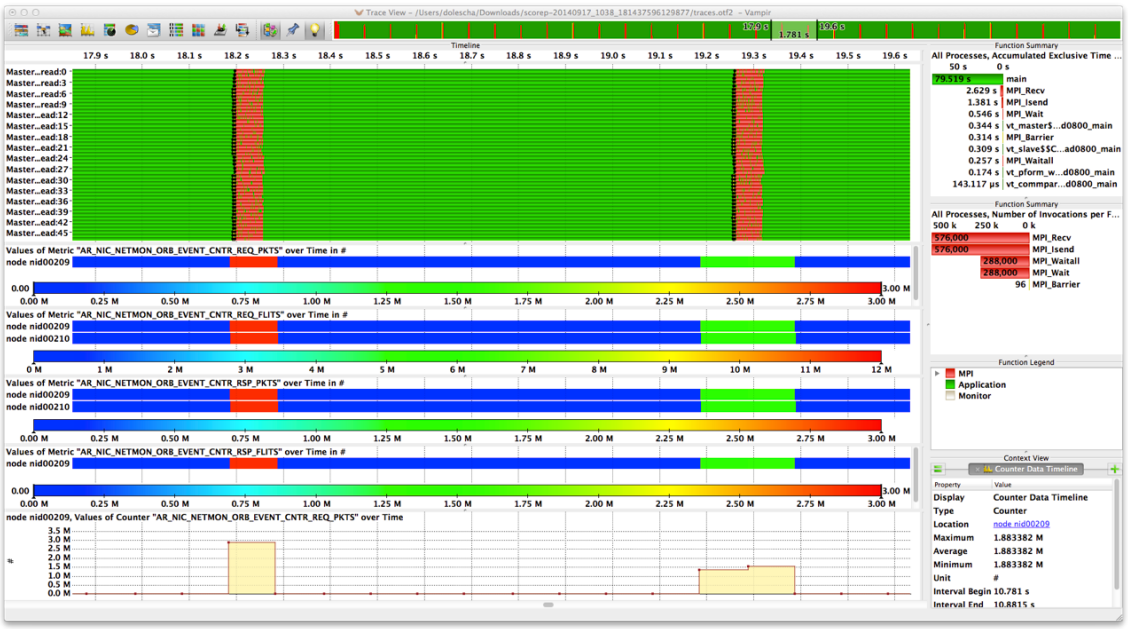


Figure 16: Vampir screenshot showing the application behavior and correlated network activity.

5.3 Selective visualization in Vampir

5.3.1 Selective visualization of program phases and processes

Exascale performance analysis will require solutions to reduce the amount of data that tools display. One such option is to only display “interesting” locations that behave differently, which requires an automatic approach for pre-processing. Vampir uses pre-calculated summary information to partially load and analyze large event trace information and to visualize only specific segments of the whole monitoring run. In addition, it allows the exclusion of specific threads of execution from the analysis and visualization. Figure 17 shows an example for this feature while loading a NEK5000 trace file.

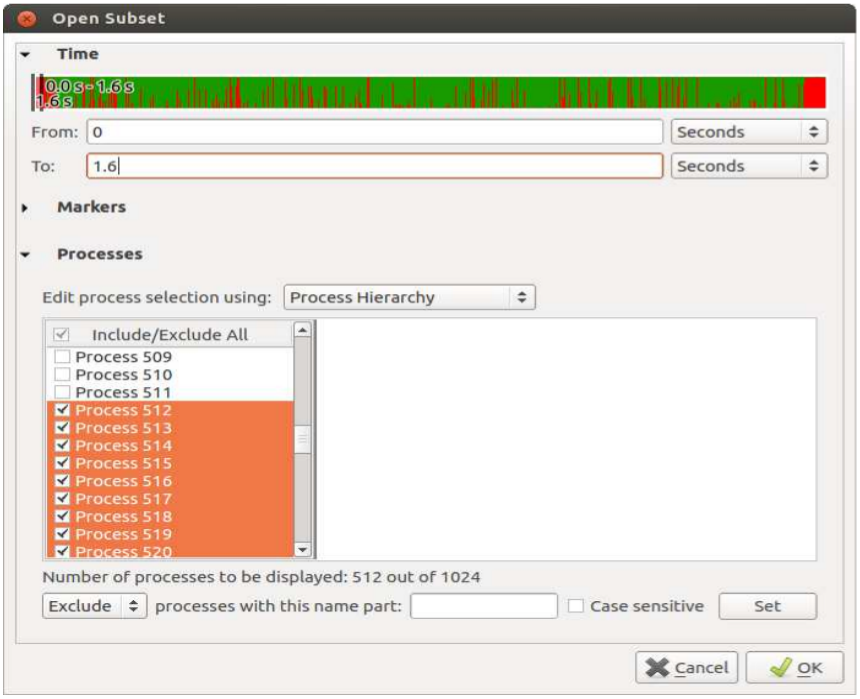


Figure 17: Vampir's Partial Loading Dialog showing the loading of the NEK5000 trace in the time range of 0 seconds to 1.6 seconds for the processes 512-1024.

5.3.2 Critical path analysis

Identifying the performance bottlenecks or critical program regions can be difficult, in particular, for long running applications, applications using a large number of processing elements, or applications using multiple parallel paradigms. Multiple layers of parallelism need to be exploited to efficiently utilize the available resources. To support application developers and performance analysts we propose a technique for identifying the most performance critical optimization targets in distributed heterogeneous applications. We have developed CASITA, a tool that uses an execution trace and the knowledge about the programming models MPI, OpenMP and CUDA as well as their hierarchy among each other to build a distributed event dependency graph. After locating wait states in this graph, we detect their root cause and compute the critical path, an important property for performance optimizations. Compared to existing analysis approaches, we incorporate the hierarchy of multiple programming models and derive a metric from both the time an activity spends on the critical path and the waiting time it caused. For the purpose of visualization, CASITA enriches the input trace with additional counter information so that results can be inspected in the Vampir trace viewer. Figure 18 shows the critical path as well as caused wit time for a hybrid use case with MPI, OpenMP, and CUDA. This topic is also covered in more detail in [29].

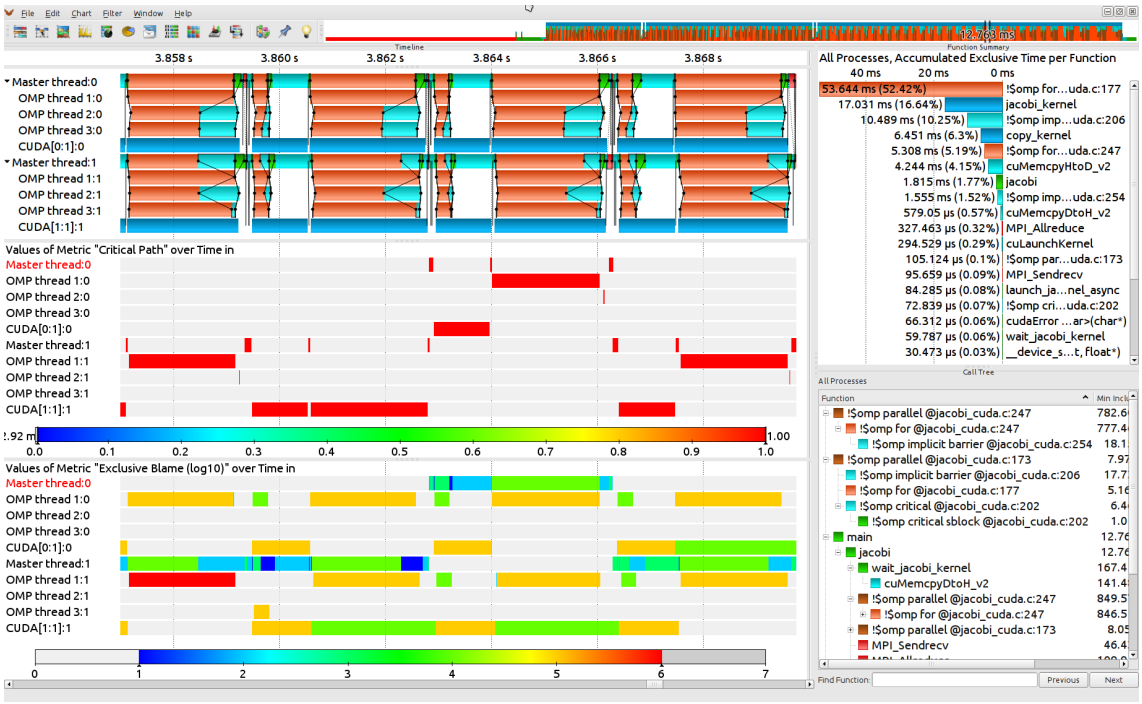


Figure 18: Vampir visualization of a simple use case with MPI, OpenMP, and CUDA. The application behavior is shown in the topmost timeline, the critical path in the second timeline and the cause wait times in the bottom timeline.

5.3.3 Alternative visualization with circular hierarchies

Load imbalances are often the root cause for inefficiencies resulting in an increased waiting time during the communication phase of the processes and, therefore, in a decreased parallel efficiency. Tools for automatic analysis often provide insight into a predefined set of performance problems only, whereas inefficiencies are often not obviously visible in comprehensive performance visualization tools. Therefore, there is a need for a more intuitive visualization of performance data that assists application developers similar to automatic analysis in identifying inefficiencies and their root cause. Furthermore, it allows a discussion of the gathered data from multiple other perspectives and the drawing of new conclusions and results.

We applied a scalable approach that combines the visualization of communicating entities of a distributed system with an arbitrary number of correlated performance metrics, such as MPI function call counts and runtimes, number of sent messages or the frequency of occurrence for a group of functions. Our visualization of performance data is based on the established concept of circular hierarchies. They allow developers to intuitively deduce the communication scheme and existing imbalances in a distributed application. We further correlate additional performance metrics with the respective communication resources in the same view, which provide valuable information about the cause of potential inefficiencies. Taking the actual system topology into account, communication can be categorized (e.g., intra-node and inter-node) and performance information aggregated to the desired level of detail in the system hierarchy. Aggregation of communication entities is necessary for multiple reasons: First, it is hard or even impossible to conceive data for extreme-scale programs consisting of tens of thousands of processes when visualized in its raw form, because the human perception is limited. Second, the size of screens is limited and only a specific amount of data can be presented at once (see also [30]).

Figure 19 gives an example of this circular visualization for Gromacs with 384 processes in an MPI-only run.

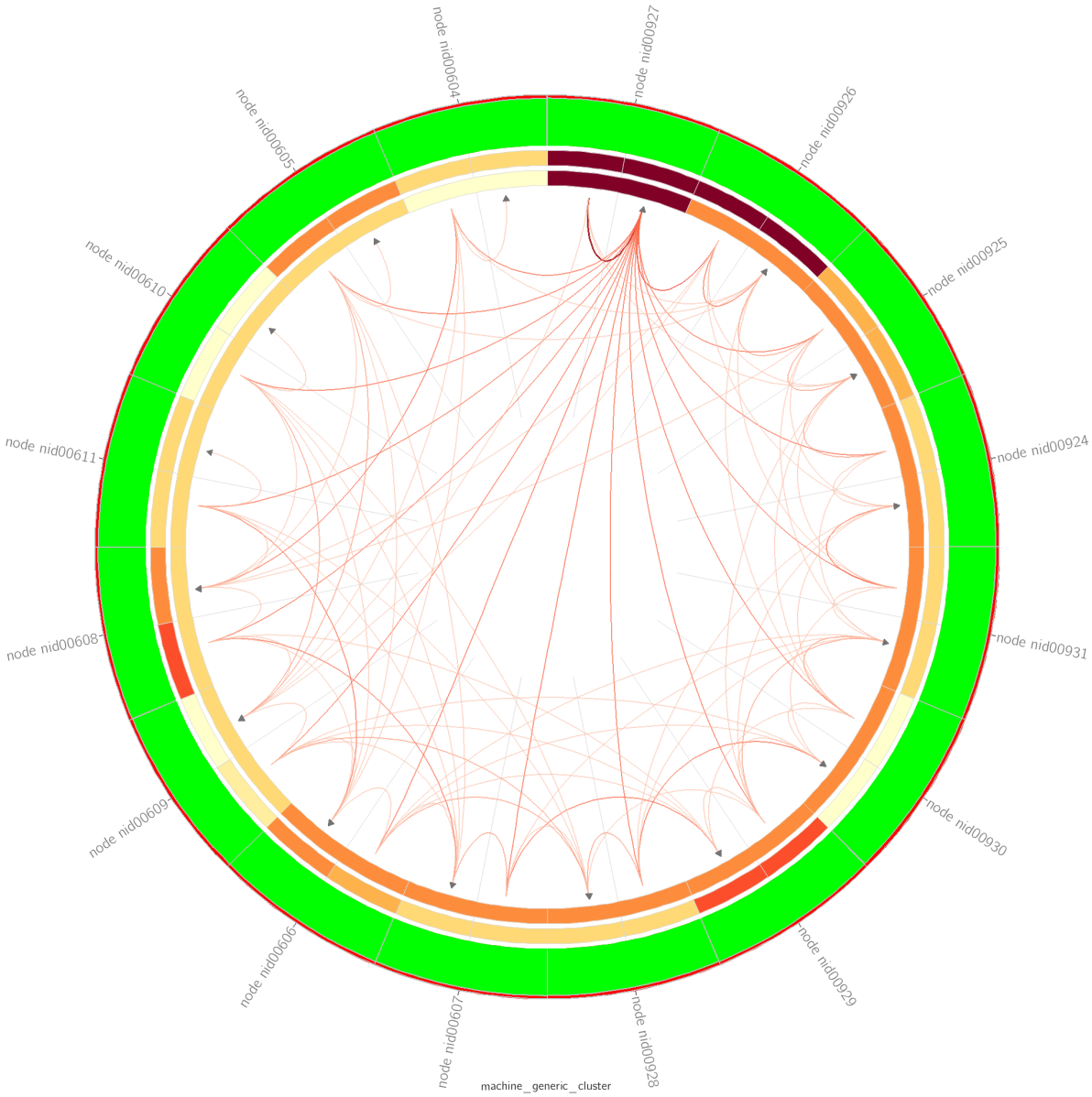


Figure 19: Circular visualization of the communication behavior of Gromacs with 384 processes. The outer circle depicts functions groups (MPI in red, user code in green), the middle circle the message volume (ranging from 1.8 GB in light yellow to 2.1 GB in dark red), and the inner circle the point-to-point messages between the nodes (thicker arrows mean more communication).

5.3.4 Online performance analysis

Projections for potential exascale architectures and systems highlight a need for large numbers of cores per node or the use of accelerators. In both cases, the compute cores share the available main memory, which is projected to increase at a far lower rate than the core count. Thus, available memory per core is likely to decrease for exascale systems. Projections indicate amounts such as 20 MB per core. Current accelerators such as Xeon Phi underline this trend with model versions with about 100 MB of memory per core. At the same time, a similar scenario exists for the available I/O resources. I/O bandwidth is projected to also increase at a lower rate than the number of compute cores. These two trends pose challenges to traditional trace-based performance optimization workflows, e.g., for the Vampir tool suite: First, with less memory per core, traces must include little detail or be short in duration. However, fine-grain detail can uncover interesting performance behavior and long running applications are common for many HPC workloads. Additionally, the limited I/O resources will make storage of extremely large exascale performance traces challenging. The available storage capacity and the available I/O bandwidth can both

become bottlenecks. Within CRESTA, we investigated an online workflow for trace-based performance analysis[26]. In this workflow, an in-memory tracing library provides a mechanism to cope with reduced availability of main memory per core. The library can automatically discard and condense tracing data before it would exceed the available main memory. This allows an automatic adaption of fine-grain tracing for short time periods and coarse-grain tracing for long running applications. Additionally, the library avoids the use of the I/O system altogether. Instead, an online analysis directly operates on tracing data and visualizes it to the tool user while the application runs. We investigated memory consumption of the library and event rates of benchmark applications with our early prototype implementation. These numbers provide a promising approach to cope with the architectural challenges of potential exascale systems.

6 Debuggers

In this section, we present extensions and modifications to Allinea DDT and MUST debuggers and the integration of MUST into Allinea DDT.

6.1 Extension to Allinea DDT

One of the major extensions is the development of a plugin infrastructure to enable integration with the MUST tool – which we describe subsequently. Initial plugin infrastructure development had the side effect of directly enabling Allinea to create performance profiling support which ultimately led to the Allinea MAP profiling tool.

New support for integrating the workflow of HPC applications with tools was developed. This was inspired by the need to sit tools within existing frameworks, the parameters and datasets created for applications are often provided by workflows. This did not fit with the typical GUI launch provided by tools. Rather than acting as standalone tool with its own launch system, we explored the capability of a launch mode that required significantly no changes to the user's own method of launching jobs.

Enabling user access to tools in the presence of complex system limits such as no X support, or strict firewalls, means that HPC access to tools is not as easy for users as it should be. This was identified as one of the major issues during the application survey in year 1. Direct experience with the ECMWF IFS CRESTA codesign application and the act of debugging remote systems such as national supercomputers led to a lower bandwidth solution for interactive tools – this work was subsequently extensively developed into the product.

There was little interest in new programming models during the project – with Coarray Fortran (ECWMF IFS) being one notable exception. As this was already supported by Allinea there was no action other than a demonstration of this by Allinea.

One new platform of major significance appeared during the project. Initially CRESTA had a strong GPU focus – but the arrival of Intel Xeon Phi allowed us to examine this platform and the tool issues of developers. A number of applications expressed interest in this. We tackled both Offload, Native and Symmetric mode. Offload required significant experimentation and development as the platform was still instable and preproduction. Whilst the platform may now be unlikely to survive, with the forthcoming KNL being a self-hosted CPU rather than an accelerator, the changes made to support this platform, provided an opportunity to address heterogeneous architecture debugging.

6.2 Extension to MUST

This section describes extensions to the runtime MPI correctness checking tool MUST and its underlying parallel tools infrastructure GTI. Goals of the extension are:

- Increased scalability of MUST's correctness checks, especially of its point-to-point analysis, collective analysis, and deadlock detection
- An investigation of support for novel parallel programming paradigms with a PGAS paradigm as demonstrator
- Contributions towards more efficient development of HPC runtime tools

6.2.1 Scalable MPI correctness analysis

The development of MUST within the CRESTA project was based on version 1.1 of the tool. This tool version provided a wide range of runtime MPI correctness checks. However, this version executed correctness analyses that involved information from two or more processes on a single process. Thus, with increasing scale this process becomes a scalability bottleneck, ultimately limiting the approach for use cases with about one hundred application processes. We extended MUST according to deliverable D3.4.2 "Debugging Design Document: Update of D3.4.1" to overcome this limitation.

We performed a study [34][35] of the time complexities that are associated with an analysis of MPI operations as part of a runtime deadlock detection. This analysis provides a low-overhead approach to handle individual MPI operations. This analysis remained centralized, i.e., was executed on a single process, but the achieved reductions in the analysis are a prerequisite for a distributed and scalable implementation. Additionally, we used this study to generalize our wait-for graph model to capture the dependencies of complex MPI operations, such as an MPI_Waitall that waits for multiple non-blocking wildcard receive operations. The generalization and improvements to our centralized deadlock detection also motivated an investigation [36] of improved error visualization techniques that target use cases such as mismatches in MPI tags or communicators.

Based on these initial studies and improvements we incorporated scalability into MUST in three steps:

- A distributed concept for MPI point-to-point analysis,
- A distributed concept for MPI collective analysis, and
- A theoretic foundation and concept for a distributed deadlock detection.

Analysis of MPI point-to-point operations must mimic the message matching of an MPI implementation. It monitors all send and receive operations of all MPI processes. This task must not be implemented in a centralized manner, as to not introduce a scalability bottleneck. An analysis [39] of this workload identifies the use of a so-called *intralayer* communication direction in a Tree-Based Overlay Network (TBON) as a promising approach to distribute the matching of point-to-point messages. This study also describes the implementation that we use in MUST and compares it to a TBON approach that does not apply intralayer communication. The comparison uses two benchmark suites (SPEC MPI2007 and the NAS Parallel Benchmarks) on two compute systems (Sierra a cluster at the Lawrence Livermore National Laboratory and Juqueen a BlueGene/Q system at the Research Center Jülich) with up to 16,384 application processes. The results are promising and highlight good scalability and wide applicability of the approach. In comparison to existing MPI correctness approaches, our concept is scalable and provides full MPI type matching capabilities, without the use of heuristics that can introduce false positives or false negatives.

In order to analyze MPI collective operations in a scalable manner, we investigated [37] the use of hierarchical correctness checks that execute on all layers of a TBON. This concept provides an efficient and scalable concept for almost all correctness analyses that apply to these operations. Type matching checks for a subset of the collective operations, e.g., for MPI_Alltoallv, form an exception and do not directly map to a TBON hierarchy. For these operations we utilize our intralayer communication to distribute their workload efficiently. Our investigation of these extensions relies on the same compute systems and benchmarks as for the point-to-point operations. They demonstrate scalability and highlight drastic performance improvements in comparison to a centralized reference implementation.

Our initial study of analysis costs for MPI deadlock detection [34][35] highlights that running a graph-based approach continuously is impractical. Following this result, we formalized a transition system that captures the semantic of MPI [38]. This transition system uses a single state vector to represent an execution state of an MPI application. As opposed to existing MPI transition systems that try to provide non-determinism coverage, our small execution states enable an efficient distributed implementation in MUST. Processes on one layer of the TBON execute this transition system. Each process manages the state for an exclusive subset of the state vector. We then combine our intralayer communication with hierarchical communication in the TBON to synchronize the state of these processes. As a result, we derive a distributed implementation of the transition system in MUST. We study one of the before mentioned benchmarks with up to 2,048 processes for this implementation. A comparison to the centralized implementation in MUST demonstrates dramatic performance improvements. We will publish subsequent measurements on the second

compute system with up to 16,384 processes for an improved version of this implementation.

Overall, our scalability extensions of MUST enable a first distributed tool for runtime MPI correctness analysis, which does not use heuristics. Existing tools either use a centralized implementation as in MUST version 1.1, or they rely on simplifications in their analysis that can yield false positives or false negatives (unreported errors or incorrect reports). At the same time, we documented our approach in detail to make it accessible for other tools and other parallel programming paradigms. Beyond that, the complete implementation of MUST relies on the parallel tools infrastructure GTI that targets simplified tool development with reduced time-to-solution.

6.2.2 Parallel programming paradigms

As to investigate runtime correctness checking for novel and upcoming parallel programming paradigms, we target prototype support for a PGAS (Partioned Global Address Space) paradigm in MUST. We selected the paradigm OpenSHMEM for this purpose, since we saw an increasing adoption and increasing support in other tools. Additionally, OpenSHMEM can cooperate with an existing MPI parallelization, which simplifies implementation in GTI and MUST. We report on the prototype checks in MUST in the following steps:

- Instrumentation and infrastructure extensions,
- Prototype checks, and
- Experiments.

In order to analyze OpenSHMEM directives for their correctness, MUST has to observe them at runtime. The OpenSHMEM reference implementation, as well as the CRAY implementation, provides weak symbols for this purpose. We provide a description of these calls to GTI, which enables it to generate wrappers for them. We generate our initial description for the OpenSHMEM calls with a script that parses a C header with all library calls. OpenSHMEM implementations that do not provide weak symbols will require an adapted instrumentation scheme that relies on library interposition.

Additionally we extended GTI to support a generalized handling of function instrumentation and to forward return values to MUST analysis in case they are needed.

Based on the available instrumentation for all OpenSHMEM directives we implement the following prototype correctness checks in MUST:

- Direct argument checks:
  - Error if size argument in get/put is negative
  - Warning if size in get/put is zero
- Out of bound memory access:
  - Error if memory access in get/put operations exceeds allocated memory regions

The direct argument checks highlight the reusability of our MUST analysis. Even though these existing analyses target MPI functions, we could successfully reuse the checks by mapping these analyses to OpenSHMEM function arguments.

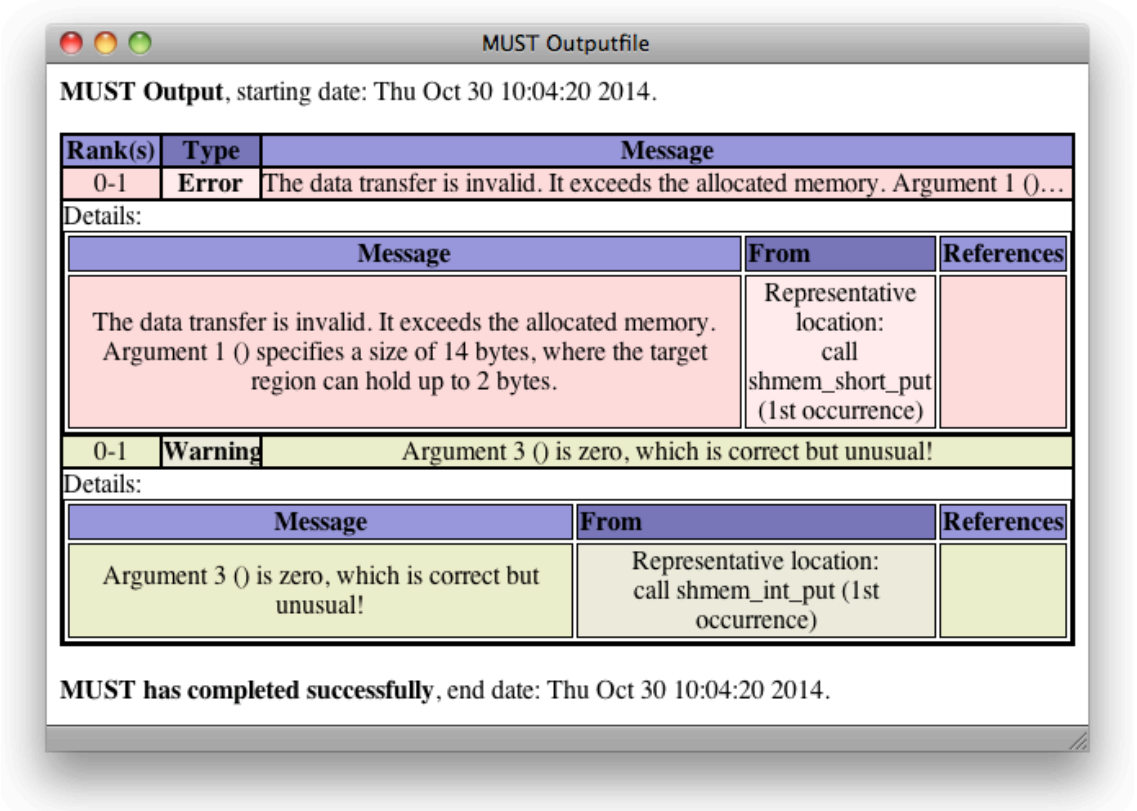


Figure 20: MUST output for a simple OpenSHMEM test case with an invalid data access and a put of count 0.

To access remote memory with OpenSHMEM, chunks of memory needs to be remotely addressable. Therefore the OpenSHMEM standard provides a malloc interface function. Memory allocated by this function can be addressed in put or get operations. The analysis we provide keeps track of these malloc operations and detects memory accesses that are outside of the allocated bounds. We use test cases to evaluate our prototype checks. For one such test case that runs without a crash or an error message with the current version of the OpenSHMEM library, we detect the issues in Figure 20. Our analysis successfully detects an out of bound access that the OpenSHMEM library fails to detect and also issues a warning for a zero size memory access.

6.2.3 Parallel tools infrastructures

MUST is implemented on top of GTI, an infrastructure for parallel tools. As a result, the MUST implementation consists of modules (dynamically loadable libraries) that implement so-called *analyses*, i.e., the actions of the tool. These modules can be reused across different tools and across different parallel programming paradigms. Our prototype checks for OpenSHMEM demonstrate this by reusing basic correctness checks for MPI. We see the development of parallel tools infrastructures such as GTI as an important step towards more efficient tools development in the future. Our goal is that GTI can provide as many common tool services to tools developers as possible. This includes instrumentation, tool startup/shutdown, acquisition of additional processes and threads to offload tool activities from application processes/threads, and providing communication services. We rely on a TBON concept that demonstrated scalability to millions of compute cores in existing approaches.

We concisely describe the underlying abstraction of GTI to document its notion of events and analyses in a report [43]. This report serves to highlight advantages of development with GTI and to provide its underlying concept to other tool infrastructures. As our experience in the development with MUST indicated, some types of analyses do not map well onto a TBON hierarchy. We introduce the intralayer communication direction [39] for this purpose. Existing tools infrastructures only provide such a communication system on the application processes. With the implementation

in GTI, tool developers can utilize a point-to-point style communication on other layers of the TBON. This enables the scalable correctness checks of MUST.

An important notion is that MUST requires a mechanism to tolerate an MPI related application crash. It must be able to analyze all events that occur before the crash, in order to detect whether it resulted from incorrect MPI usage. Within CRESTA we studied [41] different techniques to provide such a crash handling scheme and selected a fully asynchronous solution. It allows us to utilize asynchronous communication and event analysis that enables scalability. GTI implements this scheme with a combination of signal/error handlers and an alternate means of communication.

Our tests at increased scale identified out-of-memory issues for complex MUST analyses. We observed these issues especially for long running applications with high event rates. An investigation [40] of them yielded a detailed understanding of the importance of event selection. We implemented two techniques to overcome this issue and to enable correct operation of MUST in all of our test cases. The techniques that we propose and the event-selection problem apply to different types of event-based tools and can occur for simple tasks such as MPI message matching already.

Finally, we implemented a prototype tool for trace-based online performance analysis on top of GTI. This use case [42] highlights the advantages of tool development with GTI. The development of the prototype did not require extensions of GTI and allowed us to completely focus on developing the functional parts of the tool. Furthermore, we could reuse portions of the MUST implementation for the prototype, even though both tools serve widely different use cases.

6.3 Integration of MUST in Allinea DDT

A combination of both MUST and Allinea DDT is promising since:

- Some errors that MUST detects do not manifest into a crash within an application run, thus, MUST output can make programmers aware of unnoticed issues in the software;
- Some MPI usage errors are a consequence of another software fault, understanding these errors is simpler with a debugger such as Allinea DDT.

Within CRESTA we developed a combination of Allinea DDT and MUST that is based on an existing concept of configurable breakpoints. When MUST detects an MPI usage error it triggers a function to which Allinea DDT automatically sets a breakpoint. Thus, the debugger can stop the execution of the parallel application and notify the user of the tool. This enables a detailed investigation of the execution state that surrounds the detected error.

This concept is challenged by MUST's non-local correctness analyses. MUST implements correctness analyses that require information from more than one process (non-local) on additional processes. Additionally, it executes these checks asynchronously, i.e., when MUST detects a correctness issue the application may have advanced to a subsequent execution state already. Previous approaches to couple debuggers with runtime correctness tools did not support non-local analyses as a consequence. We developed the workflow that Figure 21 illustrates to overcome this limitation.

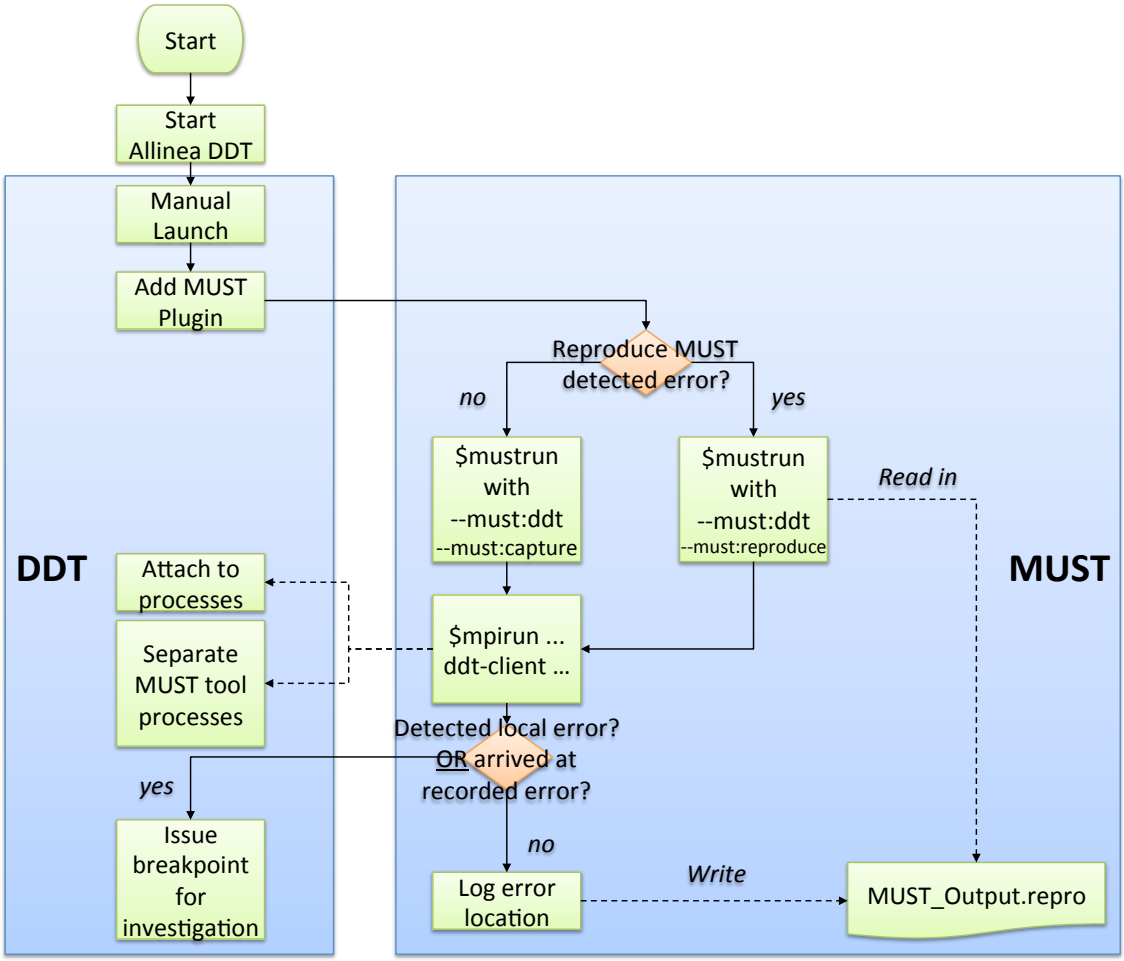


Figure 21: Workflow for the DDT-MUST integration

The user starts Allinea DDT as usual and selects a MUST plugin to enable the integration. In order to simplify a simultaneous startup of both tools we use the manual launch mode of Allinea DDT, it allows the user of the tool to start the application under the control of MUST and to connect it with Allinea DDT. Launching the application with *mustrun* works as usual, but uses the additional flag *--must:ddt*. Specifying the additional flag *--must:capture* lets MUST write a log file for all errors it detected. MUST then automatically ads the *ddt-client* command to connect the application with the running GUI of Allinea DDT. When MUST detects a process local correctness issue it directly triggers a breakpoint that enables investigation in Allinea DDT. If it detects a non-local correctness issue, it notifies the user, but immediate investigation is not possible since the error was detected asynchronously. Assuming a deterministic application, the user can use the log file that details all errors that MUST detected (MUST_Output.repro) to run the application a second time (with the *--must:reproduce* option). In that case, MUST reads in the error log file and triggers its respective warnings and errors when the application executes these commands. This mode even allows MUST to trigger breakpoints for MPI operations that are only involved in (but do not cause) the MPI usage error. As an example, if MUST detects a datatype mismatch between a send and a receive operation, using the *--must:reproduce* option, a user will not only get breakpoints for the send and the receive, but also for operations that create or commit the involved datatypes. This scheme both enables investigation of errors in Allinea DDT and it allows us to breakpoint at all involved operations of an MPI usage error.

7 Conclusions

In this deliverable, we have presented the development, extensions and modifications to different frameworks that have been developed by CRESTA WP3 to enable efficient execution and development of parallel applications on exascale machines.

We have described the development of a new framework, called “targetDP”, to support thread and instruction level parallelism for lattice-based codes. CRESTA participation in MPI Forum OpenACC and OpenMP committees has been briefly described.

We presented a first mock-up implementation of the CRESTA DSL specification to enable automatic tuning of OpenACC codes.

The software architecture and the performance of two components (runtime administration and monitoring components) of the CRESTA run-time system have been discussed.

Extensions and modifications to the Score-P and Vampir performance monitoring and analysis tools have been presented. Selective instrumentation and monitoring, hierarchical buffer management, runtime event reduction and message matching have been implemented. In addition, we reported how to deal with file system limitations, to support performance monitoring for new programming systems and application using hybrid approaches, and how to monitor energy and network performance hardware counters.

Together with the extensions and modifications to Allinea DDT and Dresden Technical University MUST debuggers, we presented the integration of the MUST MPI correctness checker into Allinea DDT parallel debugger.

8 References

[1] Kogge, Peter, Keren Bergman, Shekhar Borkar, Dan Campbell, W. Carson, William Dally, Monty Denneau, Paul Franzon, William Harrod, and Kerry Hill. "Exascale computing study: Technology challenges in achieving exascale systems." (2008).

[2] Dongarra, Jack, et al. "The international exascale software project: a call to cooperative action by the global high-performance community." *International Journal of High Performance Computing Applications* (2009)

[3] State of the art and gap analysis - Development environment, Project CRESTA Deliverable 3.1 (2012).

[4] Adaptive runtime support design document (Update), Project CRESTA Deliverable 3.2.2 (2013)

[5] Michael Schliephake, Xavier Aguilar, Erwin Laure: Design and Implementation of a Runtime System for Parallel Numerical Simulations on Large-Scale Clusters. Procedia Computer Science, Volume 4, Proceedings of the International Conference on Computational Science, ICCS 2011, 2011, Pages 2105-2114.

[6] Tunstig, Sebastian: System modelling for process mapping onto scattered computational nodes in high performance computing clusters. MS thesis KTH Royal Institute of Technology. Stockholm, 2014.

[7] Pellegrini, Francoise; Roman, Jean: SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. Proceedings of HPCN'96, Brussels, Belgium. LNCS 1067, pages 493-498. Springer, April 1996. F. Pellegrini and J. Roman.

[8] The Integrated Performance Monitoring tool (IPM). www.ipm2.org

[9] Aguilar, Xavier, Erwin Laure, and Karl Fürlinger: "Online Performance Introspection with IPM." *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 2013.

[10] Fuerlinger, Karl, Nicholas J. Wright, and David Skinner: "Effective performance measurement at petascale using IPM." *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*. IEEE, 2010.

[11] Initiative, Accelerated Strategic Computing: "The ASCI sweep3d benchmark code." (1995).

[12] Jing Gong, Stefano Markidis, Michael Schliephake, Erwin Laure, Dan Henningson, Philipp Schlatter, Adam Peplinski, Alistair Hart, Jens Doleschal, David Henty, and Paul Fischer: Nek5000 with OpenACC. Proceedings of EASC, 2014.

[13] Michael Wagner, Jens Doleschal, Andreas Knüpfer und Wolfgang E. Nagel, “Selective Runtime Monitoring: Non-intrusive Elimination of High-frequency Functions”, in High Performance Computing Simulation (HPCS), 2014 International Conference on, pages 295-302, 2014.

[14] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, “The Vampir Performance Analysis Tool Set,” in Tools for High Performance Computing. Springer, pp. 139–155, 2007.

[15] A. Knüpfer, C. Rössel, D. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, “Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir,” in Tools for High Performance Computing 2011, Springer, pp. 79–91, 2012.

[16] M. Geimer, F. Wolf, B.J. Wylie, E. Abraham, D. Becker, and B. Mohr, “The Scalasca Performance Toolset Architecture,” Concurrency and Computation: Practice and Experience, vol. 22, no. 6, pp. 702–719, 2010.

[17] S. S. Shende and A. D. Malony, “The Tau Parallel Performance System,” International Journal on High Performance Computing Applications, vol. 20, no. 2, pp. 287–311, 2006.

[18] Jens Doleschal, Thomas William, Bert Wesarg, Johannes Ziegenbalg, Holger Brunst, Andreas Knüpfer, and Wolfgang E. Nagel, “Towards Detailed Exascale Application Analysis – Selective Monitoring and Visualisation”, in Proceedings of EASC, 2014.

[19] Michael Wagner, Jens Doleschal, Andreas Knüpfer and Wolfgang E. Nagel, “Runtime Message Uniquification for Accurate Communication Analysis on Incomplete MPI Event Traces”, in Proceedings of the 20th European MPI Users' Group Meeting, Madrid, Spain, pages 123-128, ACM, 2013.

[20] Michael Wagner, Andreas Knüpfer, and Wolfgang E. Nagel, “Hierarchical Memory Buffering Techniques for an In-Memory Event Tracing Extension to the Open Trace Format 2”, in Parallel Processing (ICPP), 2013 42nd International Conference on, pp. 970–976, 2013.

[21] Michael Wagner and Wolfgang E. Nagel, “Strategies for Real-Time Event Reduction”, in Euro-Par 2012: Parallel Processing Workshops, ser. Lecture Notes in Computer Science. Springer, vol. 7640, pp. 429–438, 2013.

[22] Domenic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf, "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries", in Applications, Tools and Techniques on the Road to Exascale Computing, volume 22 of Advances in Parallel Computing, pages 481–490, 2012.

[23] S. R. Alam, H. N. El-Harake, K.R Howard, N. Stringfellow and F. Verzelloni, “Parallel I/O and the Metadata Wall“, Parallel Data Storage Workshop (PDSW’11), 2011.

[24] W. Frings, F. Wolf, and V. Petkov, “Scalable massively parallel i/o to task-local files”, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ser. SC ’09, ACM, pp. 17:1–17:11, 2009.

[25] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole, “Enabling Event Tracing at Leadership-Class Scale through I/O Forwarding Middleware”, in Proceedings of the 21th International Symposium on High Performance Distributed Computing, ser. HPDC ’12. ACM, pp. 49–60, 2012.

[26] Michael Wagner, Tobias Hilbrich, and Holger Brunst, “Online Performance Analysis: An Event-based Workflow Design Towards Exascale”, in Proceedings of the 16th International Conference on High Performance Computing and Communication (HPCC), 2014. Online draft: http://rcswww.zih.tu-dresden.de/~hilbrich/drafts/online-performance.pdf

[27] Jing Gong, Stefano Markidis, Michael Schliephake, Erwin Laure, Dan Henningson, Philipp Schlatter, Adam Peplinski, Alistair Hart, Jens Doleschal, David Henty, and Paul Fischer, “Nek5000 with OpenACC”, in Proceedings of EASC, 2014.

[28] Alistair Hart, Harvey Richardson, Jens Doleschal, Thomas Ilsche, Mario Bielert und Matthew Kappel, “User-level Power Monitoring and Application Performance on Cray XC30 Supercomputers”, 2014.

[29] Felix Schmitt, Jonas Stolle, and Robert Dietrich, "CASITA: A Tool for Identifying Critical Optimization Targets in Distributed Heterogeneous

Applications" in 44rd International Conference on Parallel Processing Workshops (ICPPW), 2014.

[30] Felix Schmitt, Robert Dietrich, René Kuß, Jens Doleschal, and Andreas Knüpfer, “Visualization of Performance Data for MPI Applications Using Circular Hierarchies, 1st Workshop on Visual Performance Analysis (VPA), 2014.

[31] G. Mozdzynski, M. Hamrud, N. Wedi, J. Doleschal, H. Richardson: A PGAS Implementation by Co-design of the ECMWF Integrated Forecasting System (IFS). In High Performance Computing, Networking, Storage and Analysis 2012, 2012.

[32] W. Frings, F. Wolf, and V. Petkov: Scalable massively parallel i/o to task-local files. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, ser. SC ’09. New York, NY, USA ACM, pages 17:1–17:11, 2009.

[33] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole: Enabling Event Tracing at Leadership-Class Scale through I/O Forwarding Middleware. In Proceedings of the 21th International Symposium on High Performance Distributed Computing, ser. HPDC ’12. ACM, pages 49-60, 2012.

[34] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. „MPI Runtime Error Detection with MUST: Advances in Deadlock Detection.“ In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC ’12, pages 30:1–30:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[35] Tobias Hilbrich, Joachim Protze, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. „MPI Runtime Error Detection with MUST: Advances in Deadlock Detection.“ Scientific Programming, 21(3–4):109–121, 2013.

[36] Joachim Protze, Tobias Hilbrich, Bronis R. de Supinski, Martin Schulz, and Matthias S. Müller. „MPI Runtime Error Detection with MUST: Advanced Error Reports.“ In Alexey Cheptsov, Steffen Brinkmann, José Gracia, Michael M. Resch, and Wolfgang E. Nagel, editors, Tools for High Performance Computing 2012, pages 25–38. HLRS, Springer Berlin Heidelberg, 2013.

[37] Tobias Hilbrich, Fabian Hänsel, Martin Schulz, Bronis R. de Supinski, Matthias S. Müller, Wolfgang E. Nagel, and Joachim Protze. „Runtime MPI Collective Checking with Tree-Based Overlay Networks.“ In Proceedings of the 20th European MPI Users’ Group Meeting, EuroMPI’13, pages 129–134, New York, NY, USA, 2013. ACM.

[38] Tobias Hilbrich, Bronis R. de Supinski, Wolfgang E. Nagel, Joachim Protze, Christel Baier, and Matthias S. Müller. „Distributed Wait State Tracking for Runtime MPI Deadlock Detection.“ In Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, SC’13, pages 16:1–16:12, New York, NY, USA, 2013. ACM.

[39] Tobias Hilbrich, Joachim Protze, Bronis R. de Supinski, Martin Schulz, Matthias S. Müller, and Wolfgang E. Nagel. „Intralayer Communication for Tree-Based Overlay Networks.“ In 42nd International Conference on Parallel Processing (ICPP), Fourth International Workshop on Parallel Software Tools and Tool Infrastructures, pages 995–1003, Los Alamitos, CA, USA, 2013. IEEE Computer Society Press.

[40] Tobias Hilbrich, Joachim Protze, Michael Wagner, Matthias S. Müller, Martin Schulz, Bronis R. de Supinski, and Wolfgang E. Nagel. „Memory Usage Optimizations for Online Event Analysis.“ To appear in: Solving Software Challenges for Exascale, EASC ’14. Springer Berlin Heidelberg, 2014. Online version: https://e-reports-ext.llnl.gov/pdf/772313.pdf

[41] Joachim Protze, Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, Wolfgang E. Nagel, and Matthias S. Müller. „MPI Runtime Error Detection with MUST: A Scalable and Crash-Safe Approach.“ To appear in: 43nd International Conference on Parallel Processing (ICPP), Fifth International Workshop on Parallel Software Tools and Tool Infrastructures, Los Alamitos, CA, USA, 2014. IEEE Computer Society. Online draft: http://rcswww.zih.tu-dresden.de/~hilbrich/drafts/crash-handling.pdf

[42] Michael Wagner, Tobias Hilbrich, and Holger Brunst. „Online Performance Analysis: An Event-based Workflow Design Towards Exascale.“ To appear in: The 16th IEEE Inter- national Conference on High Performance Computing and Communications, HPCC 2014, 2014. Online draft: http://rcswww.zih.tu-dresden.de/~hilbrich/drafts/online-performance.pdf

[43] Tobias Hilbrich, Martin Schulz, Holger Brunst, Joachim Protze, Bronis R. de Supinski, and Matthias S. Müller. „An Event-Action Mapping Abstraction for Parallel Tools Infrastructures.“ Submitted to: Proceedings of the 2015 IEEE 29th International Parallel and Distributed Processing Symposium, IPDPS ’15, Washington, DC, USA, 2015. IEEE Computer Society. Online draft: http://rcswww.zih.tu-dresden.de/~hilbrich/drafts/mapping-abstraction.pdf