# D3.11 – Experiences With Benchmarks and Co-design 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, Jing Gong, 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), George Mozdzynski (ECWMF), David Lecomber (Allinea) |
| **Reviewer(s)** | Jan Westerholm (ABO), Achim Basermann (DLR) |

| **Dissemination level** | |
|---|---|
| PU | *PU* |

## 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 | 4/11/2014 | Changing Performance Monitoring part, added Allinea Contribution | Stefano Markidis (KTH) |
| 0.9 | 25/11/2014 | Response to internal reviewers | Stefano Markids (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 experiences gained with applying the methods and tools developed in WP3 to benchmarks and co-design CRESTA applications developed in WP6. We describe the experience with benchmarks and application for each WP3 task (“Programming models”, “Compilation and runtime environments”. “Performance analysis tools”,”Debuggers”). For each framework developed in WP3, a critical review of outstanding issues is performed and future research directions are outlined.

We describe first the experiences gained with the PGAS programming model by developing a Coarray Fortran benchmark suite, using the Coarray Fortran in the IFS application to calculate Legendre Transforms and implementing Fast Fourier Transforms in UPC. In addition, we report the first results using the targetDP programming framework in Ludwig, a lattice Boltzmann application.

We investigate the use of compiler support for GPU programming by porting the NekBone, a skeleton version of Nek5000 code, to multi-GPU systems and present the performance results. We describe the co-design work with OpenACC in GROMACS. We use a first implementation of an auto-tuning system for OpenACC code to tune the OpenACC version of the Nek5000 code. The co-design work, involving the development of the adaptive runtime system and Nek5000, is described, and the use of different components of the runtime systems in benchmarks is presented.

The new features of Score-P and Vampir (support for new programming systems and new hardware counters, selective monitoring and enhanced scalability) are used in CRESTA applications: Nek5000, OpenFOAM, IFS, HemeLB, Gromacs.

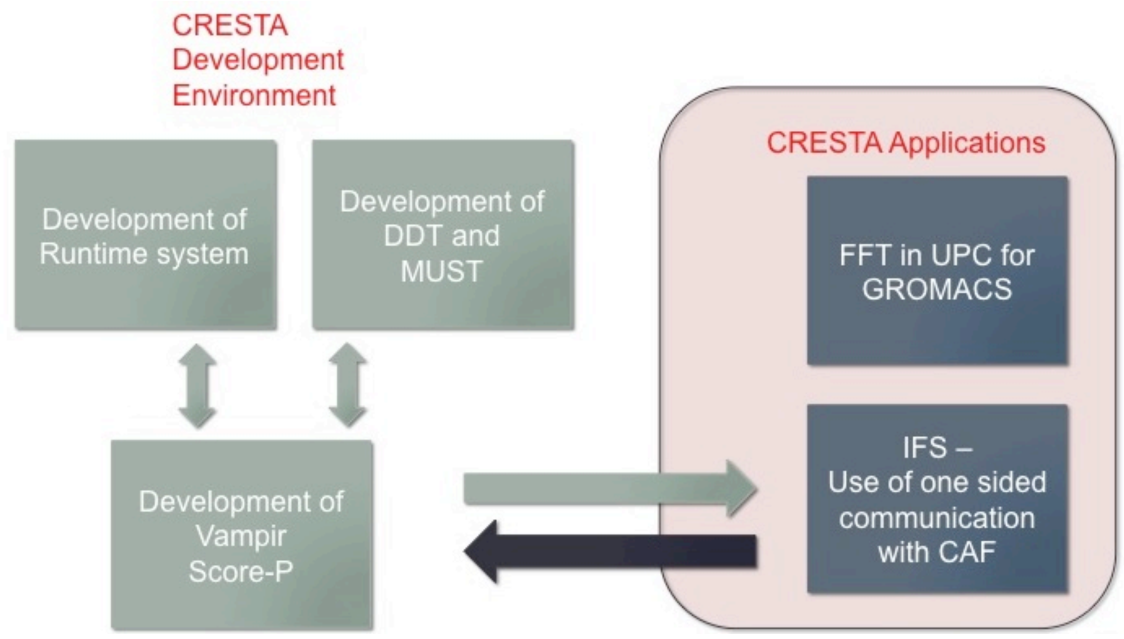
The Allinea DDT and MAP tools and MUST correctness checker are used in HemeLB CRESTA application to detect and analyze software errors and correctness on large-scale HemeLB simulations.

# 2 Introduction

This deliverable describes on the experiences gained with applying the methods and tools developed in WP3 to benchmarks and co-design CRESTA applications. We describe the experience with benchmarks and application for each WP3 task:

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

The design of the different frameworks for exascale applications has been guided by the software co-design process within the CRESTA project. Figure 1 presents an example of the co-design activity aimed at the design of some of the WP3 frameworks.



CRESTA Development Environment
Development of Runtime system
Development of DDT and MUST
Development of Vampir Score-P
CRESTA Applications
FFT in UPC for GROMACS
IFS – Use of one sided communication with CAF

**Figure 1: Example of co-design in the CRESTA development environment**

Within the CRESTA project, two applications, IFS (from CRESTA WP6) and the computation of Fast Fourier Transform (from CRESTA WP4), use Partitioned Global Address Space (PGAS) languages Cray Co Array Fortran (CAF) and Unified Parallel C (UPC) in selected regions of the codes. Cray CAF and UPC are based on the DMAPP API. The performance monitoring and analysis of the PGAS code in IFS and in the FFT require support for Cray CAF and UPC languages in Score-P and Vampir. In particular, it is important to understand when and how the remote memory access occurs in the applications to optimize the code. For this reason, a support for Vampir and Score-P for PGAS languages has been designed and implemented in prototype version. Both application and Vampir and Score-P developers worked together giving reciprocal feedback during the development of the tools and application. In addition, the development of debuggers and of the run-time systems also benefited from the support for PGAS in Vampir and Score-P. Since the performance monitoring tools and debuggers are based on similar technologies, the experience gained with Vampir and Score-P can be be used in developing debugger support for Cray PGAS languages. Because the CRESTA runtime system has a performance-monitoring component, it benefited from improvement of Score-P and Vampir.

The goal of this deliverable is to present experiences gained with applying the methods and tools developed in WP3 [1] to benchmarks and co-design CRESTA applications. The deliverable is organized as follows. The third section presents the experience gained with PGAS programming model in benchmarks and IFS application. The fourth section reports the use of OpenACC in the Nek5000 application, the use of the auto-tuner for OpenACC in the Nek5000 application, and the co-design work with Nek5000 to develop the CRESTA adaptive runtime system. The fourth section describes the use of Vampir and Score-P new features in Nek5000, HemeLB, OpenFOAM, IFS, Gromacs applications. The fifth section presents the use of Allinea DDT and MAP tools and MUST correctness checker in HemeLB. Finally the sixth section concludes the deliverable summarizing the results.

## 2.1 Purpose

The goals of this deliverable are:

- To present experiences with different benchmarks and applications with the PGAS programming model.
- To present the use of the targetDP framework in the Ludwig application.
- To present use of compiler support for GPU programming with OpenACC in Nek5000 code.
- To present the use of CRESTA auto-tuner for OpenACC codes in Nek5000
- To present the co-design work involving the development of the CRESTA adaptive runtime system and Nek5000 and the experiences with different component of the runtime system in benchmarks.
- To present the experiences with Score-P and Vampir with different CRESTA applications.
- To present the use of Allinea MAP and MUST in the HemeLB CRESTA application.

## 2.2 Glossary of Acronyms

| | |
|---|---|
| **AVX** | Advanced Vector eXtension |
| **CAF** | Coarray Fortran |
| **D** | Deliverable |
| **DSL** | Domain Specific Language |
| **CUDA** | Compute Unified Device Architecture |
| **GPU** | Graphics Processing Units |
| **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

In CRESTA WP3, we focused on investigating the use of the Partitioned Global Address Space (PGAS) programming model in benchmarks and applications. PGAS languages such as Unified Parallel C [2] have been the subject of much attention in recent years, in particular due to the exascale challenge. There is a widespread belief that existing message-passing approaches such as MPI will not scale to this level due to issues such as memory consumption and synchronization overheads. PGAS approaches offer a potential solution as they provide direct access to remote memory. This reduces the need for temporary memory buffers, and may allow for reduced synchronization and hence improved message latencies. Some modern distributed memory architectures allow for remote memory access directly over the interconnect, meaning the PGAS model maps directly onto the underlying hardware. PGAS features have been introduced into the Fortran 2008 standard with coarrays [3]. Programming using coarrays has many potential advantages compared to MPI. Amongst these are simplicity, compiler checking and scope for automatic optimization of communications by the compiler. Coarrays can also be introduced incrementally to existing MPI codes to improve performance-critical kernels.

## 3.1 Coarray Fortran Benchmark Suite

Since Fortran coarrays are in their relative infancy, and full compiler support has only recently emerged, it is important to understand the performance characteristics of parallel operations. Benchmark results are important as they guide both the applications programmer and the compiler or library developer. Applications programmers can make informed decisions about the most appropriate parallel features to use, and estimate performance in advance. Compiler and library developers can easily measure the performance characteristics of their implementation, and target areas of weakness. Although these developers will have their own internal performance tests, user-driven benchmark suites are very important as they can highlight those features of most interest to applications programmers.

An initial prototype Fortran coarray benchmark suite had been produced by EPCC [4], and this was further developed, distributed and evaluated under CRESTA. The benchmark measures:

1. single contiguous point-to-point read and write;
2. multiple contiguous point-to-point read and write;
3. strided point-to-point read and write;
4. all basic synchronization operations;
5. halo-swapping in a multi-dimensional regular domain decomposition;
6. reference results from MPI for selected key operations.

The benchmark is available on the WEB [5] and initial performance results on the Cray XE6 were reported at the 2012 Cray User Group [6]. The benchmark made it clear when the compiler was able to pattern-match the communications calls and optimize them using techniques such as vectorization. It also uncovered a performance bug which was reported to Cray and fixed for later compiler releases.

When the new Cray XC30 was released, the benchmark was used to compare the performance of its new ARIES network with the GEMINI network of the previous XE6 system. These results were presented at EASC2014 [7], showing significant improvements in bandwidth and latency. For example, in Figure 2 and Figure 3 we show the time taken for remote writes of small amounts of data with Fortran coarrays on these two platforms, using three different kinds of synchronization: global (sync all), point-to-point (sync images) and no synchronization. The equivalent MPI results are also measured. These results show that although the latency of the network has not changed significantly, coarray performance on the XC30 is improved due to better synchronisation times. This conclusion was confirmed by direct measurements of these overheads taken in the synchronization section of the benchmark suite. It is also interesting to note the MPI performance is extremely good: the MPI standard has been in existence for two decades and library implementers are very good at optimizing its performance, especially for small message sizes.

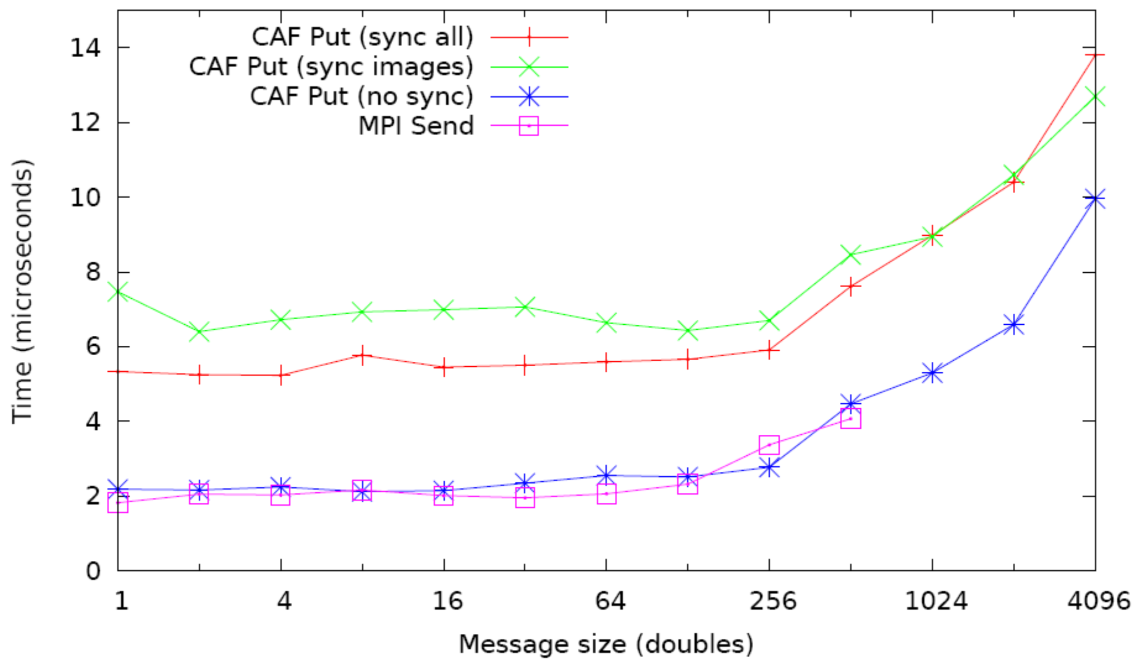


**Figure 2: Point-to-point performance of remote writes on Cray XE6**

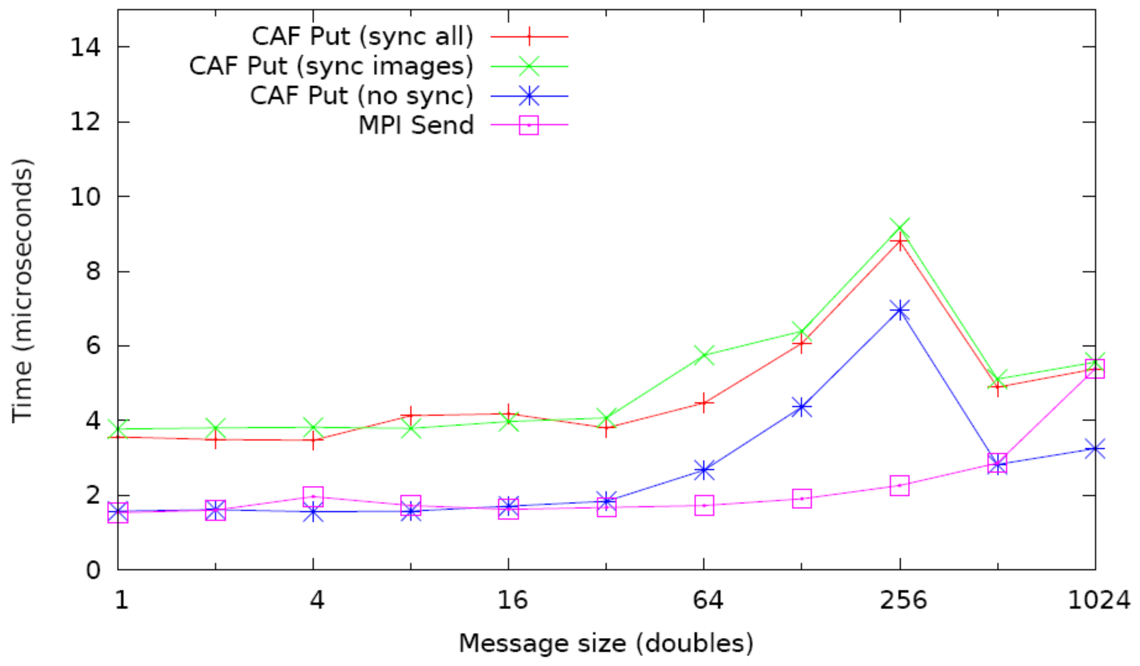


**Figure 3: Point-to-point performance of remote writes on Cray XC30**

The benchmark is still under development. For example, we plan to include a new synchronization mechanism (event post and event wait) recently introduced into the Fortran standard.

## 3.2 Coarray Fortran in IFS

IFS is a numerical weather prediction application within the CRESTA project. This is a production code used to provide medium-range weather forecast products up to 10 to 15 days ahead.

For IFS the focus of developments in CRESTA was primarily to use Fortran2008 coarrays within OpenMP parallel regions to overlap computation with communication and thereby improve performance and scalability. The importance of this research is such that if these developments are successful then the IFS model may continue to use the spectral method to 2030 and beyond on an exascale sized system. This research is further significant as the techniques used should be applicable to other hybrid MPI/OpenMP codes with the potential to overlap computation and communication.

Within the CRESTA project we used Fortran2008 coarrays to overlap these communications with the computations in the Legendre and Fourier transforms. For the Legendre transforms this is being done per wave number within an OpenMP parallel region. In the original approach the computation and communication are done sequentially, with no overlap. In the new scheme (using coarrays) each thread is computing and then communicating its computed data to the respective tasks of its 'communicator' group. While Fortran2008 has no coarray groups/teams construct it is nevertheless trivial to compute a mapping to a set of image numbers. Experience has shown that the Cray DMAPP library is not thread safe with the CCE compiler version 8.0.6 and earlier releases and a workaround has been used to locate coarray transfers in OMP CRITICAL SECTIONs with a small performance penalty for doing so today. The coarray puts are expected to be non-blocking, and only waited on for completion on a subsequent SYNC IMAGES statement. For the direct Legendre transforms a similar approach is used, the original approach and new coarray approach. Here the coarray gets in each thread are clearly blocking until data arrives and then progress onto computation. We we have focused on the Legendre transform and the PGAS approach to overlap computation with communication, by performing these in a single OpenMP parallel region which operates over spectral wave numbers. A similar scheme is employed for calculating the Fourier transforms in IFS. Instead of spectral wave numbers, the Fourier transforms operate over latitudes, where tasks in Fourier space have a subset of latitudes and a subset of atmospheric levels. In Figure 4, from Ref. [9], the performance improvement from the coarray optimizations (LCOARRAYS=T means all the optimizations are on), which peak at 21% at around 40K cores then dip about 5% after this.

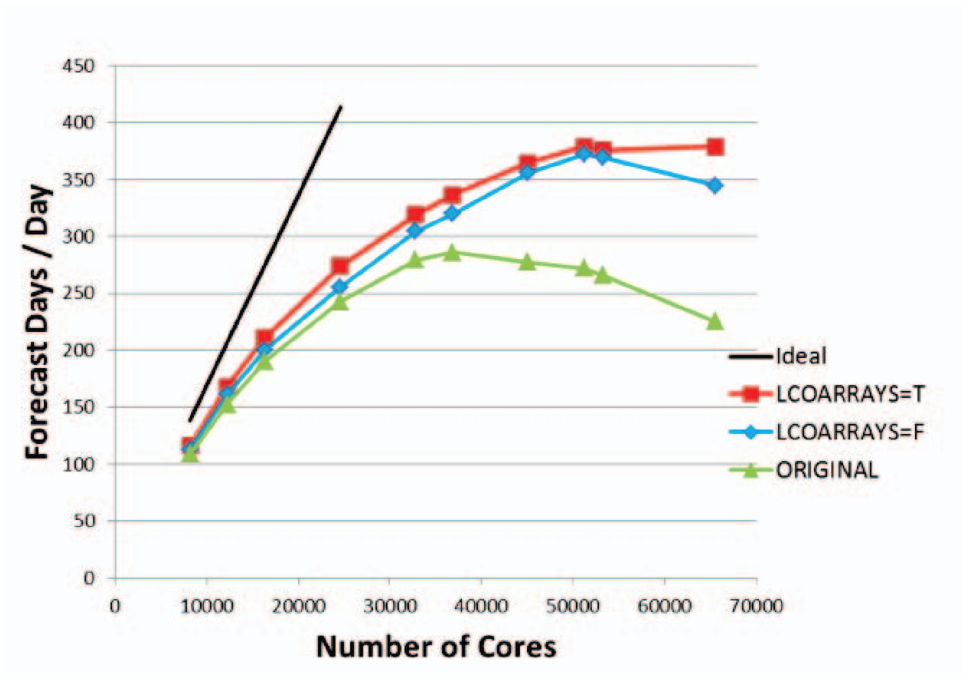


**Figure 4: Scalability of different versions of the IFS code**

## 3.3 FFT in UPC

Within CRESTA, we examined two fundamentally different approaches to performing the transpose operation of a 3D-FFT [10] in UPC. Since the advent of Cray Baker systems, Cray delivers a network system with support for remote memory access, meaning that data located on remote processes can be accessed without involvement of the remote processor. This feature is available to end-users on Cray machines through the Distributed Memory Application (DMAPP) API, which supports compiler-based or library based one-sided communication. The model captures the idea of having several processes running the same code in its own address space, but also having access to remote memory segments of other processes through PUT/GET semantics. DMAPP provides a layer for interfacing with the remote memory access capacity of the hardware, and the functions provided by DMAPP can be roughly divided into three different variants:

- **blocking functions**: A process may resume execution after a call to a blocking function, only after the results of this operation is globally visible to the entire system.
- **Non-blocking explicit:** The explicit non-blocking function returns a synchronization identifier, which may be used to determine when the effects of the operation are globally visible.
- **Non-blocking implicit:** For an implicit non-blocking function, the results are only guaranteed to be globally visible after a synchronization call by the initiator of the function.

We examined the traditional approach to transposing the data, by using a regular blocking transpose operation at the end of computation, and we compare this with a non-blocking transpose operation, where we send data to the other processes as soon as it is available. These two versions use the blocking functions of DMAPP, and the non-blocking implicit functions, respectively.

One may calculate the 3-D Fourier transform of a matrix A of dimension n_x X n_y X n_z by a series of 1-D Fourier transforms, one in each direction, x, y, z. To parallelize the FFT algorithm, the data in the matrix A is distributed over the processes, such that each process has a set of x-y planes of A in local memory. This means that we may perform a number of 2-D Fourier transforms of all the planes local to each process, without any form of communication. However we need to transpose the matrix A in order to perform the final and last step, the one-dimensional FFT in the Z-direction. After the transpose operation, this final 1-D FFT operation may be performed without any communication. Figure 5 outlines the basic, major steps of the 3D-FFT algorithm.

**Algorithm 1** Transpose operation UPC

```
//upc_forall
for $i = 1 : n_z$ do
    for $j = 1 : n_y$ do
        fft($A_{*,j,i}$)
    end for
    for $j = 1 : n_x$ do
        fft($A_{j,*,i}$)
    end for
    $s \leftarrow n_y / P_M$

    for $k = 0 : P_M - 1$ do
        upc_memput($A_{*,k \cdot s:(k+1) \cdot s-1,i}, C_{i,*,k \cdot s:(k+1) \cdot s-1}$)
    end for
end for
$C \leftarrow$ local_transpose($C$)
```

**Figure 5: First Algorithm to calculate the FFT with UPC**

We have implemented a modified version of the NAS FT benchmark which overlaps communication and computation. The essential difference is that, as soon as we have calculated the FFT of a 2D-plane in a process, we instantly proceed with *PUT*ing the results into the remote memory segment of the process that will need this plane for the final Z-direction FFT. This means that the actual transpose takes place at the same time as the computation. This may be contrasted with the original NAS FT version, where we first calculate the FFT of all planes local to each process, and then in a separate step perform the global transpose as in the Figure 6.

**Algorithm 2** Transpose operation UPC

```
//upc_forall
for $i = 1 : n_z$ do
    for $j = 1 : n_y$ do
        fft($A_{*,j,i}$)
    end for
    for $j = 1 : n_x$ do
        fft($A_{j,*,i}$)
    end for
    $s \leftarrow n_y/P_M$

    for $k = 0 : P_M - 1$ do
        upc_memput($A_{*,k\cdot s:(k+1)\cdot s-1,i}, C_{i,*,k\cdot s:(k+1)\cdot s-1}$)
    end for
end for
$C \leftarrow$ local_transpose($C$)
```

**Figure 6: Second algorithm to calculate the FFT with UPC**

Here we instead employ the PUT semantics, so that each process, as soon as it has finished calculating a plane, puts the result in the memory area of the right processes.

The computational tests have been carried out on KTH Cray XE6 supercomputer "Lindgren". Each node consists of two AMD Opteron 12-core Magny-Cours (2.1 GHz) processors and 32GB DDR3 memory that is shared between the two processors. The nodes themselves are connected via a Cray Gemini network. We used the Cray compiler for the UPC version of our program, again with the flags -O3, -h vector 3 and –h restrict=a (assume no aliasing). The Class D problem of NAS benchmark has been chosen to test the FFT UPC implementation. This test consists of solving FFT on 2048 x 1024 x 1024 grid points 25 times. We carried out scaling test up to 1028 cores, comparing two FFT implementations: one version uses a blocking algorithm and a blocking remote memory access (UPC blocking), while the other one uses an algorithm that allows overlap of communication and computation and implicit non-blocking remote memory access. Figure 7 shows the parallel speed-up for the two UPC implementations.

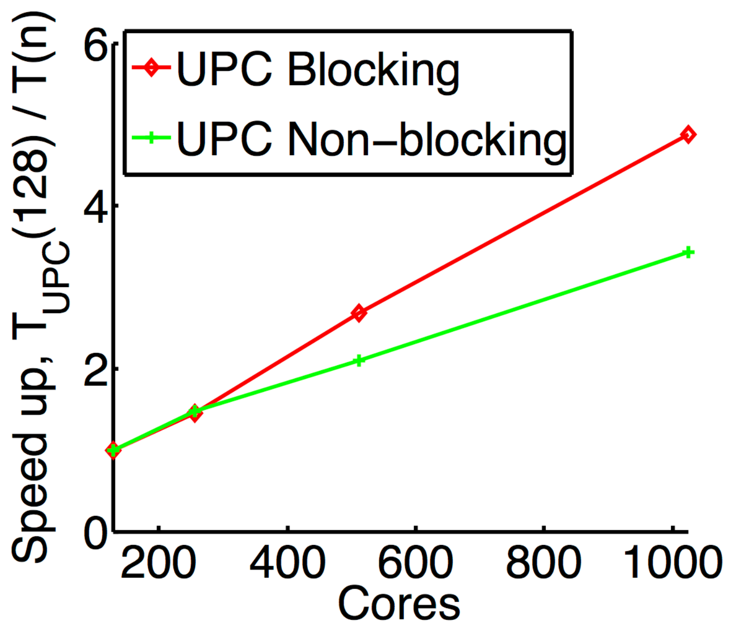


**Figure 7: Relative parallel speed-up of the different UPC FFT implementations.**

Finally, we have compared the performance of our FFT with FFT implemented in MPI and OpenMP on 128 Cores. Our UPC FFT implementation using overlapping communication and computation proves faster than the MPI and OpenMP version. The execution time for the UPC version is 209 seconds while the version with MPI OpenMP in Fortran takes 236 seconds.

## 3.4 TargetDP in the Ludwig code

The work on the development of “targetDP” (presented in CRESTA D3.7) was motivated by development of the Ludwig complex fluid simulation package at EPCC at University of Edinburgh. This versatile software is able to simulate a variety of soft matter substances such as mixtures, particle suspensions and liquid crystals, with relevance to many large industrial concerns such as foodstuffs, paints and coatings, and oil recovery. The basis is hydrodynamics using the lattice Boltzmann (LB) technique, coupled with a free energy based approach for various order parameters, the dynamics of which are solved via standard finite-difference techniques. We have recently developed Ludwig so that it can use many GPUs in parallel as well as traditional CPU based supercomputers. The difficulty in maintaining duplicate source code for the two architectures is a key motivation for the work described here. Furthermore, the existing version relies on the compiler to find ILP and map to SIMD instructions, but the extents of innermost loops in the code are dictated by the model and typically do not map perfectly onto the vector hardware. The lattice-based operations in real applications such as Ludwig are typically much more complex than the example given above, but the same methodology can be applied. To demonstrate effectiveness and evaluate performance, we have implemented targetDP within a real computational kernel extracted from Ludwig. This ``binary collision'' code performs an LB collision operation on a mixture of two fluids.

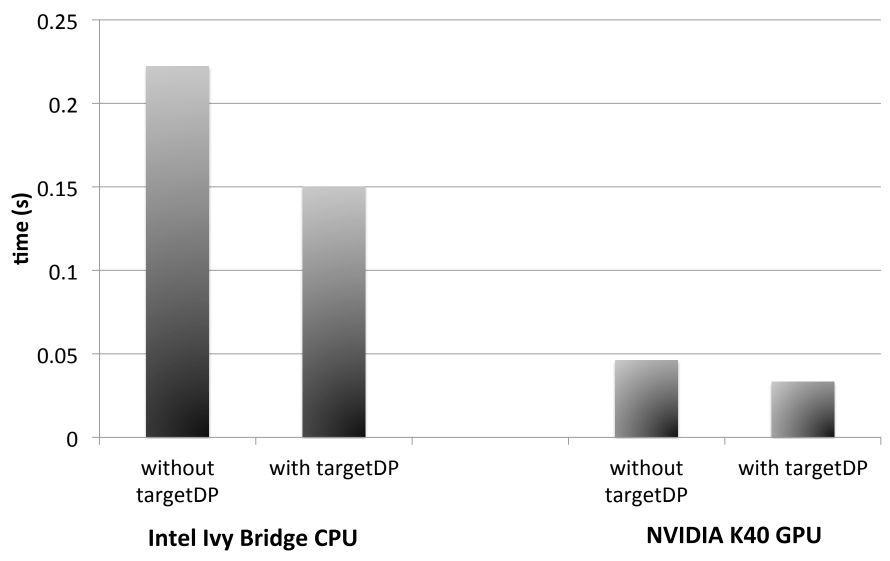


**Figure 8: Performance of TargetDP**

In Figure 8 we show the effect on performance, for this benchmark, of our targetDP framework for both CPU (2.7 GHz, 12-core E5-2697 Intel Ivy Bridge) and GPU (NVIDIA K40) architectures, noting that the same source code is used for the targetDP results on both. It can clearly be seen that the use of targetDP not only offers performance portability, but it also significantly increases performance in each case; this is due to the intelligent exposure of ILP. Within the original CPU code, each innermost most loop is over the discrete lattice momenta (here of extent 19) or over spatial dimensions (i.e. of extent 3), neither of which map perfectly onto the AVX vector length of 4. The compiler is not able to generate optimal AVX instructions, thus leaving the vector units under-utilised. With our targetDP implementation, we instead expose the lattice-based parallelism to the compiler as ILP. We tune the VVL, with 8 being the optimal value (i.e. the compiler generates 2 AVX instructions for each innermost loop). This tailored ILP optimisation gives almost a 1.5X performance improvement the original code (which has been augmented with OpenMP for a fair comparison). Similarly, for the reasons described above, exposing ILP within each kernel offers performance benefit on the GPU. In this case we tune VVL to be 2, and we see a performance boost of 1.4X. Incidentally, the GPU targetDP benchmark implementation outperforms the CPU by 4.5X.

We have successfully secured funding from the UK ARCHER eCSE programme to build on this work by fully implementing targetDP within Ludwig, such that the code will be performance portable across the range of leading-edge HPC architectures. The concepts and technology of targetDP are also applicable to other applications and areas, and we will strive to facilitate uptake.

## 3.5 Outstanding Issues and Future Work

During the CRESTA project period, WP3 focused on investigating the use of PGAS CAF and UPC in benchmarks and applications. This approach resulted in visible improvements of scalability as in the case of the IFS code, showing that PGAS can effectively be used in applications to achieve higher scalability. The main challenge in using PGAS approaches in a real-world application, such as IFS, was to ensure interoperability between different programming systems (MPI, OpenMP and CAF) that use common resources. Often the use of this approach resulted in runtime errors due in some cases to compiler bugs. Future work will focus on studying the interoperability of different programming approaches.

# 4 Compilation and Runtime Environments

Heterogeneous HPC architectures are becoming increasingly prevalent in the Top500 list with CPU-based nodes being enhanced by accelerators or coprocessors optimized for floating-point calculations. This trend is likely to increase as we move towards exascale capable systems and it is vital that the relevant HPC applications are able to exploit this heterogeneity.

Whilst accelerators offer a large boost in peak system speed, it is difficult to translate this into sustained applications performance. For GPU accelerators, applications are typically rewritten in a low-level language such as CUDA or OpenCL. This is a productivity drawback, with developers having to maintain multiple versions of their code without any guarantee of portability. In addition, the HPC community is nervous about investing substantial software development effort in converting applications to use a programming language that is not portable between different architectures. On the other hand, OpenACC, a collection of compiler directives specified by the programmer to identify areas that should be accelerated, enable existing HPC applications to run on accelerators with minimal source code changes.

## 4.1 Accelerating Nek5000 with OpenACC

Within CRESTA, we have used NekBone, a skeleton application of Nek5000. Nek5000 was chosen as one of the CESTA co-design applications under investigation. It is an open-source code used for the simulation of incompressible fluid flow and it is employed in a broad range of domains, including the study of thermal hydraulics in nuclear reactor cores, the modeling of ocean currents and the simulation of combustion in mechanical engines.

NekBone has been configured to capture the basic structure and user interface of the extensive Nek5000 software and exposes its main computational kernel to reveal the essential elements of the algorithm-architectural coupling that is relevant to Nek5000.

NekBone has been successfully ported to multi-GPU systems using OpenACC compiler directives. The focus of this work was on porting the most time-consuming routines of the NekBone to GPU system: the *ax3D* and *gs_op* subroutines. To port NekBone to GPU systems required little effort and a small number of additional lines of code. In fact, after the porting, the total number of lines of NekBone was 41,953 including 45 OpenACC directives. Approximately, one OpenACC directive was used per 1,000 lines of code.

The naive implementation using OpenACC led to little performance improvement: from 16 Gflops obtained on the CPU, we reached 20 Gflops with the naive OpenACC implementation. The optimization of matrix-matrix multiplication required evaluating the computational cost of loop-nesting to assist the developer in guiding the OpenACC loop scheduling. By simply instructing the compiler to collapse four nested loops in the matrix-matrix multiplication, we reached approximately 43 Gflops, doubling the performance of the naive OpenACC implementation. In addition we ported and optimized NekBone on a multi-GPU system by working on the *gs_op* subroutine. The optimized version for Multi-GPU system gave a parallel efficiency of 79.9 % on 1024 GPUs of the Titan supercomputer as visible in the Figure 9.

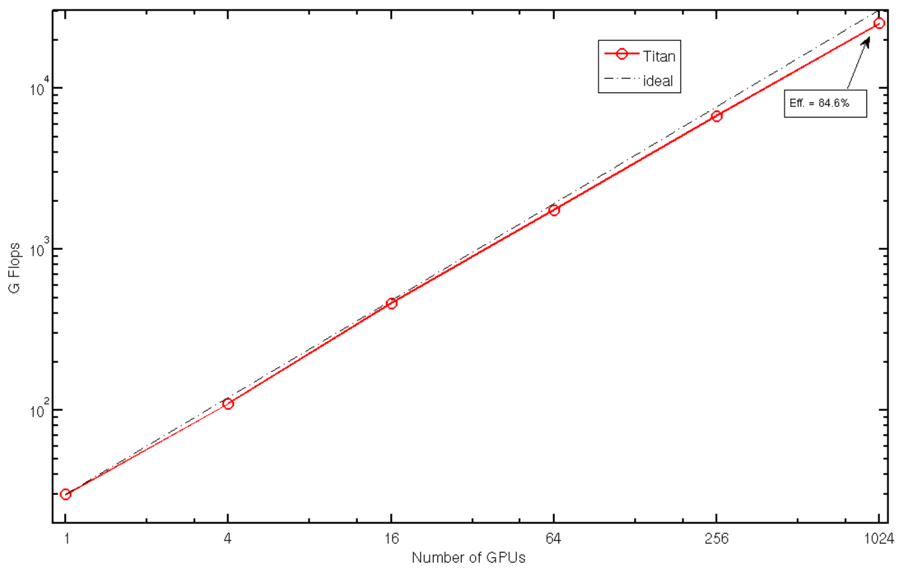


**Figure 8: Scaling of the Nekbone application on Titan supercomputer.**

### 4.1.1 Outstanding Issues and Future Work

We used OpenACC compiler support to port the NekBone mini-application, that is skeleton version of Nek5000, to multi-GPU systems. The next step in this work is to use OpenACC in the full Nek5000 application. For doing this, we need to use OpenACC in the Nek5000 pre-conditioners and in the Multi-grid linear solver. In addition, the possibility of directly transfer data from a GPU memory to another GPU memory in the Nek5000 gather-scatter operator will be investigated.

## 4.2 OpenACC co-design with GROMACS

The porting of Nek5000 to exploit the GPUs on ORNL's Cray XK7 Titan system demonstrated how the OpenACC programming model exploits system software (compilers and runtime libraries) to accelerate HPC applications in a productive and portable manner. This also demonstrated a degree of co-design, as the Nek5000 developers within CRESTA filed a number of functionality and performance bugs that were found in the Cray Compilation Environment (CCE). These bugs were then fixed by the Cray Programming Environment (PE) compiler development team, and the improved product is now available for all Cray customers.

This is, however, only responsive and co-design in the broadest sense. With the GROMACS code, CRESTA sought to complete the co-design loop in a more proactive manner. The GROMACS code has been ported to run on one or more Nvidia GPUs using CUDA. As with most of the performance-intensive parts of the GROMACS code, considerable effort has been spent to hand-optimise these CUDA kernels to obtain high levels of performance using detailed knowledge of the underlying hardware.

The OpenACC programming model aims to provide a high-level alternative for programming GPUs. It is likely that this will entail a performance sacrifice compared to the more low-level CUDA. This has been measured for many codes and is typically around 10%, which is an acceptable cost for many developers when compared with productivity and maintainability advantages of using OpenACC. To measure this margin requires access to codes that have both OpenACC and CUDA versions, and the CUDA in these codes is typically quite generic and has not had a great deal of tuning.

GROMACS' hand-tuned CUDA provided the Cray compiler engineers with a much tougher challenge that could be used to improve the CCE OpenACC performance. A detailed report of this work is provided in Annex A. The work focused on the "nxnbn" kernel that dominates the runtime when simulating non-bonded systems. Based on the CUDA, an equivalent C routine was written, and accelerated to run on a GPU using

OpenACC directives. This OpenACC nxnbn kernel could then be swapped into the code using the CUDA interoperability in the OpenACC standard.

The performance of the two versions was then compared using a representative non-bonded problem. Initial results were as expected, with the OpenACC version only giving around half of the performance of the optimised CUDA. Considerable effort then went into understanding the reasons behind this reduced performance.

Three main causes were identified. Firstly, the OpenACC kernel used a lot more registers than the CUDA version. This led to "register spilling", where data is placed into slower shared memory instead of machine registers, which impacts performance. In response to this, a CCE compiler flag was added that allows users to limit the maximum number of registers that are used in an OpenACC kernel. The second issue was that the original kernel made calls to CUDA intrinsic functions that increased performance. Work was then done in the CCE optimizer (OPT) and code generation (CG) phases to make use of these same intrinsic function sets (without user intervention). Finally, the CUDA driver code optimised the shared memory/cache configuration, based on whether shared memory was used in the kernels. This was not originally done in CCE, but functionality was added to the OPT and CG phases to detect this automatically as well.

With these three modifications, the OpenACC performance was now within 10-15% of the hand-tuned CUDA. This was viewed as a good achievement by the GROMACS developers. The end result was not a full OpenACC port of GROMACS; only one of 24 CUDA kernels was studied. All 24 would need to be ported to move entirely to OpenACC. In addition, with an existing CUDA code that is faster, there was little appetite from the developers to move to OpenACC at the current time.

The main result was, instead, the improvements in the compiler (from CCE version 8.3 onwards). None of the modifications are specific to GROMACS and can therefore benefit a wide range of OpenACC codes. They also require little or no user intervention (one compiler flag in one of the three cases), so the benefits are largely transparent to the user.

Overall, these successes demonstrate the advantages of co-design in both directions, with the applications leading to improved systemware, and this improved systemware then giving improved application performance, both for the original co-design application but also for a wider class of codes

## 4.3 Autotuning of an OpenACC version of Nek5000

In CRESTA we have developed an autotuning technology that can address the inherent complexity of programming the latest and future computer architectures. The autotuner provides a framework in which an application developer can try out various optimization strategies in an automated fashion to maximize their application performance. This autotuner explores a tuning parameter space by repeatedly building and running the application. From these the best run is chosen using a metric obtained from the program execution that currently is done by exhaustive search. To accomplish a tuning run, the source is appropriately preprocessed and compiled and an optimization process is organized.

We have carried out an extensive autotuning study on NekBone since it is understood that any improvement achieved on the computational structure of NekBone could also be applied to Nek5000.

### 4.3.1 Implementation

NekBone is configured to very closely resemble the basic structure of Nek5000. In NekBone a matrix is initialized and then a linear system is solved twice for every computational cycle using a Conjugate Gradient (CG) solver. A large number of small rectangular matrix multiplications take place at each solver iteration. Previous work in

CRESTA has demonstrated that the computation of those matrix multiplications dominates the execution time of NekBone. Therefore we focused on an OpenACC implementation of a large number of different algorithms used to calculate the matrix-matrix multiplications.

The main subroutines to optimize implement independent matrix-matrix multiplication kernels and there are three difference cases that could be considered for a given number of elements, *N*, depending on the sizes of the matrices involved in the multiplication. Those three cases of *C = A* B* are:

- Case 1: *A [$N^2$xN] x B [NxN] = C [$N^2$xN]*
- Case 2: *A [NxN] x B [NxN] = C [NxN]*
- Case 3: *A [NxN] x B [Nx$N^2$] = C [Nx$N^2$]*

An example of those kernels can be seen in the code shown below:

```
do j = 1, n3
      do i = 1, n1
            c( i, j ) = 0.0
            do k = 1, n2
                  c ( i, j ) = c ( i, j ) + a ( i, k ) * b ( k, j )
            end do
      end do
end do
```

To execute this kernel on a GPU using OpenACC we included additional compiler directives assuming that the data had already been copied to the GPU, for instance:

```
!$ACC PARALLEL LOOP PRESENT(a,b,c) PRIVATE(i,j,k)
do j = 1, n3
      do i = 1, n1
      c( i, j ) = 0.0
            do k = 1, n2
                  c ( i, j ) = c ( i, j ) + a ( i, k ) * b ( k, j )
            end do
      end do
end do
!$END PARALLEL LOOP
```

Although this should be enough to get part of the code running on a GPU further investigation is required for an optimum performance. In order to find the suitable kernel we created a number of different implementations of the above kernel using different parameters and OpenACC optimizations. These implementations were then enumerated so that the CRESTA autotuner could identify and compare them.

Over ten different implementations of each matrix-matrix multiplication kernel were included in the autotuning benchmark providing many different computation paths for the NekBone kernel and exploring the following types of optimizations:

- specific hard-coded versions for different values of n1, n2 and n3 so that these would be constant at compile time;
- different loop orderings;
- loop unrolling;

- hand tiling the matrices into blocks for better cache reuse;
- calls to DGEMM BLAS routines;
- matrix values stored explicitly in temporary scalars;
- loop collapsing.

Among the most important OpenACC parameters that were used to optimize the kernels were VECTOR_LENGTH, GANG, WORKER or COLLAPSE. An example of autotuning kernel can be seen below:

```
!$ACC PARALLEL PRESENT(a,b,c) PRIVATE(i,j,k) VECTOR_LENGTH(VLENGTH)
!$ACC LOOP GANG WORKER VECTOR COLLAPSE(2)
do j = 1, n3
      do i = 1, n1
#ifdef SCALAR
            tmp = 0.0
#else
            c(i, j) = 0.0
#endif
            do k = 1, n2
#ifdef SCALAR
                  tmp = tmp + a(i,k) * b(k, j)
#else
                  c ( i, j ) = c ( i, j ) + a ( i, k ) * b ( k, j )
#endif
            end do
#ifdef SCALAR
            c(i,j) = tmp
#endif
      end do
end do
!$END PARALLEL LOOP
```

The autotuning session of the CRESTA autotuning framework can be controlled by a domain-specific language, DSL, either from a global configuration file or embedded in the application source. The DSL component helps the autotuning framework to optimize an application over a set of tuning parameters. One of the most useful characteristics of this autotuning framework is what has been termed *scenario characterization parameters* where for each scenario we aim to pick the best values for a set of tuning parameters. The tuning parameters will relate to build and runtime optimization choices which we can choose to give, for instance, the best runtime.

After a large number of tuning sessions the autotuner demonstrated that there was a particular routine faster than all the others for a given a set of parameters. This routine has the particularity of using COLLAPSE(4) as part of the OpenACC optimization.

```
!$ACC PARALLEL PRESENT(a,b,c) PRIVATE(i,j,k) VECTOR_LENGTH(VLENGTH)
!$ACC LOOP GANG WORKER VECTOR COLLAPSE(4)
do imat = 1, lelt
!dir$ nonblocking
        do j = 1, n3
                do i = 1, n1
                        tmp = 0.0
                        do k = 1, n2
                                tmp = tmp + a(i,k,imat) * b(k, j)
                        end do
                        c(i,j,imat) = tmp
                end do
        end do
end do
!$END PARALLEL LOOP
```

Therefore the chosen routine was introduce in NekBone and compared to a previous OpenACC hand-tuned implementation carried out on the CRESTA project. The latter performance results can be seen on Figure (left). Figure Figure 9 represents global performance of our optimized NekBone application depending on the number of elements, *nel*, used in the simulation and the size of the matrix, *N.* As the size of the matrix is increased NekBone used more memory to run the application, which is the reason why the application runs into memory limits on the GPU at large value of *N*.

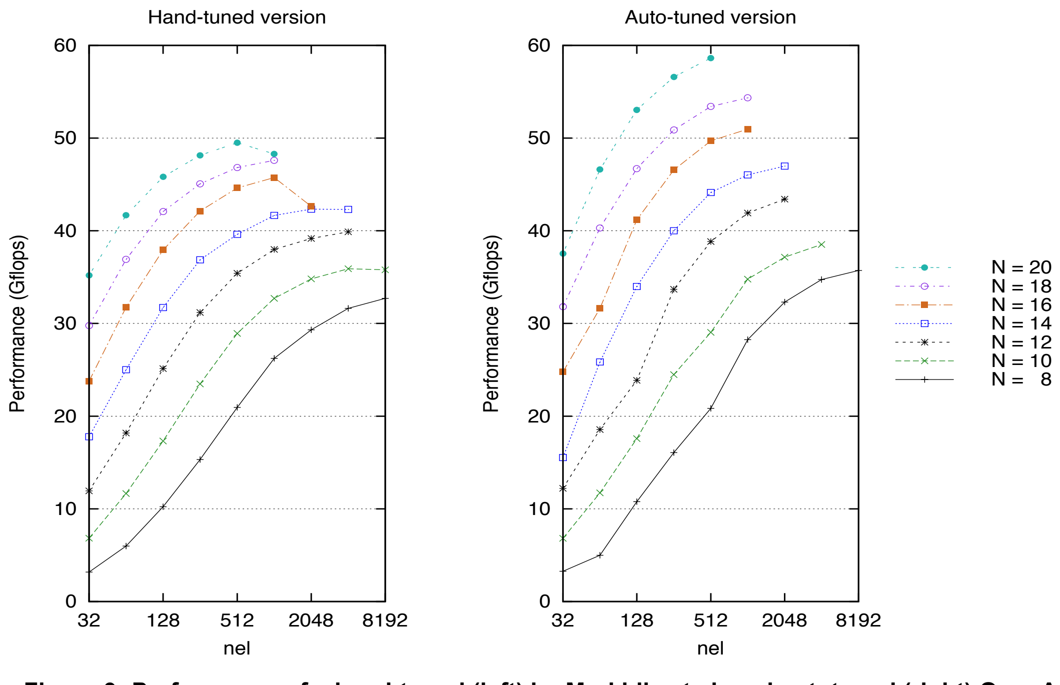


**Figure 9: Performance of a hand-tuned (left) by Markidis et al. and autotuned (right) OpenACC NekBone**

To illustrate the effect of parameter tuning, Figure 9 (right) shows the performance results of the autotuned version of NekBone that demonstrates the performance improvement over the hand-tuned version. In Figure 10 we have represented the ratio between our autotuned performance results over the hand-tuned performance results achieved by Markidis et al. and when using default OpenACC settings.

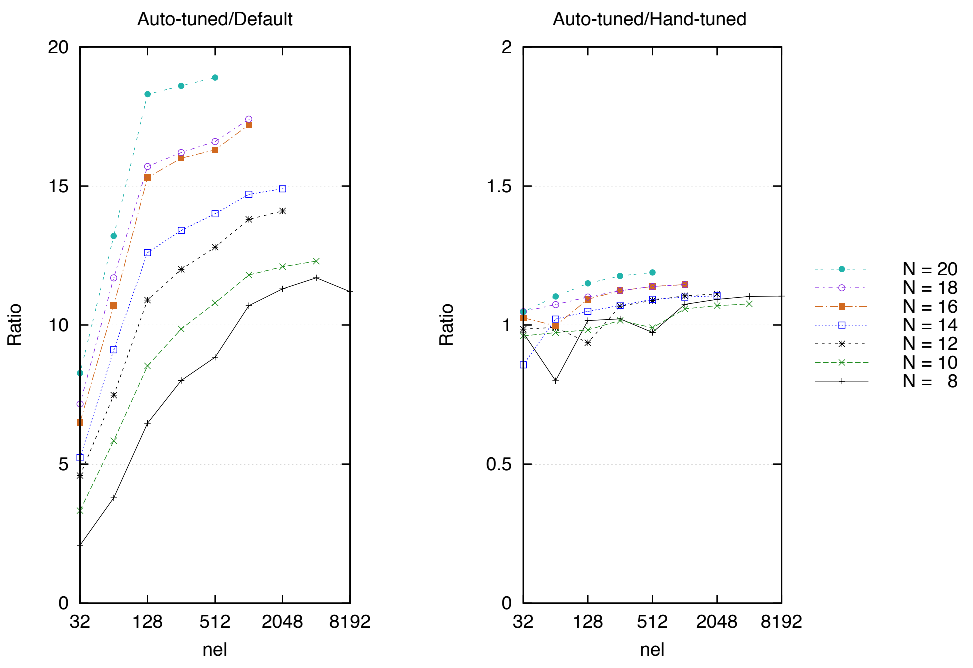


**Figure 10: Performance ratio of auto-tuned, hand-tuned and default OpenACC settings**

### 4.3.2 Outstanding Issues and Future Work

Thanks to the new NekBone structure developed for this purpose and the exhaustive exploration of different parameter values carried out by the autotuner we have accomplished a simpler, better structured and faster implementation of NekBone. Furthermore, the exploration of different OpenACC optimization algorithms has revealed that loop collapsing techniques have given the best performance improvements among all the other optimization techniques previously mentioned. Scalar reduction showed little performance improvement, however the vector length seemed to influence the performance with its optimum value for 128 and 256.

Although the autotuner pointed to different kernel settings during the tuning session, we were able to identify cases where kernels performed very differently when run in isolation compared to being run in the main NekBone code. After further investigation it was discovered that when run in isolation the Cray compiler was able to non-block some sections of the kernel code whereas it was not when run in the main NekBone. The addition of additional directive `!dir$ nonblocking` solved the problem outperforming the best hand-tuning efforts.

Across a wide range of representative cases, the autotuning increased the performance of NekBone by nearly 200% compared to the default OpenACC settings. Furthermore, we also compared to an OpenACC hand-tuned version of NekBone and for representative problem sizes, the autotuned version always performed within a few percent of the hand-tuned version and outperformed it by over 15% for the largest systems.

## 4.4 Hybrid and Adaptive Runtime System With Nek5000

In this section, we report on the co-design approach used in the development of the adaptive runtime system.

### 4.4.1 *Fast collective MPI communication*

The efficiency of the runtime system [16] depends also on the availability of fast collective MPI communication operations for the exchange of the software and hardware model data, performance measurements as well as control information in order to execute the decisions of the system. These data must be distributed with low latency despite the fact that the data is often short and collective MPI operations have a comparatively high latency for them. Furthermore, the re-mapping of computational tasks makes it necessary to move all user data that define the status of computational tasks between different nodes. Conceptually, this can be done with collective communication operations like MPI_Alltoallv too. But, it is necessary to provide separate send and receive buffers to them in order to achieve high performance. Widely used implementations of MPI_Alltoallv are very slow when they are used with the option MPI_IN_PLACE.

The CRESTA application NEK5000 [17] is a PDE solver with a long development history and contains a communication module that has been optimized for its typical short, latency-bound messages. This communication module can use the regular collective MPI communications as well as own implementations of them. One of these implementations is based on the crystal_router algorithm.[18] This algorithm allows sending messages of arbitrary length between arbitrary nodes in a hypercube network. It is advantageous especially in irregular applications where the exact nature of the communication is not known before it occurs or where the message emergence changes dynamically.

#### 4.4.1.1 Implementation of Fast Collective Communication Operations

Communication operations in hypercube networks are often implemented by routing algorithms that iterate over the dimensions of the cube and execute in each step one point-to-point communication operation with the partner node at the other end of the respective edge. The result of the binary xor function with the processor numbers of sender and receiver node as arguments provides a routing path that can be used to transport the message. Therefore, messages can be delivered in algorithms following this pattern from each node to each other node in at most d communication steps where d is the dimensionality of the hypercube network. Such a choice of paths provides load balancing in the communication of several typical applications as well as it is optimal if all processors are used in a load balanced way.

Algorithm 1 explains how the transport of messages between arbitrary processes works. First, all messages are stored in a buffer for outgoing messages of the sender process (`msg_out`). During the iteration over the different channels (i.e. the bits of rank numbers), some messages will be transmitted in each iteration step according to their routing path. For that, those messages that must be transferred through a certain channel will be copied from `msg_out` to a common transfer buffer (`msg_next`). The buffer `msg_next` of each process will be exchanged through the active channel of the current iteration step with the respective buffer of a partner process. Thereafter, all messages that had to be routed from this partner over this channel can be found in `msg_next`. They will be inspected there. Messages that are addressed to the receiving process will be copied into the buffer for incoming messages (`msg_in`) from where they can be accessed by the application code later. Messages that have to be forwarded further in one of the following iteration steps will be kept and put into `msg_out`.

**Algorithm 1:** Pseudocode of the crystal router algorithm, adapted from [18].

```
begin crystal_router
  declare buffer msg_out;  /* buffer for messages to send    */
  declare buffer msg_in;   /* buffer for received messages   */
  declare buffer msg_next; /* buffer for messages to send    */
                           /* in the next communication step */
  for each msg in msg_out do
    if dest_rank(msg) == myrank then
      copy msg into msg_in;
  end for
  for each dimension of the hypercube i = 0,...,d-1 do
    for each message msg in msg_out do
      if (dest_rank(msg)&myrank) ^ $2^i$) then
        copy  msg into msg_next;
    end for
    exchange buffer msg_next with process(myrank ^ $2^i$));
    for each message msg in msg_next do
      if dest_rank(msg) == myrank then
        copy msg into msg_in;
      if msg needs to be routed further then
        copy msg into msg_out;
    end for
  end for
end crystal_router
```

We developed a synthetic benchmark for the analysis of the original crystal router algorithm. Its design has been based on the communication pattern in NEK5000.[19]

The measurements have been done on KTH's system Lindgren described previously in Section 3.3.

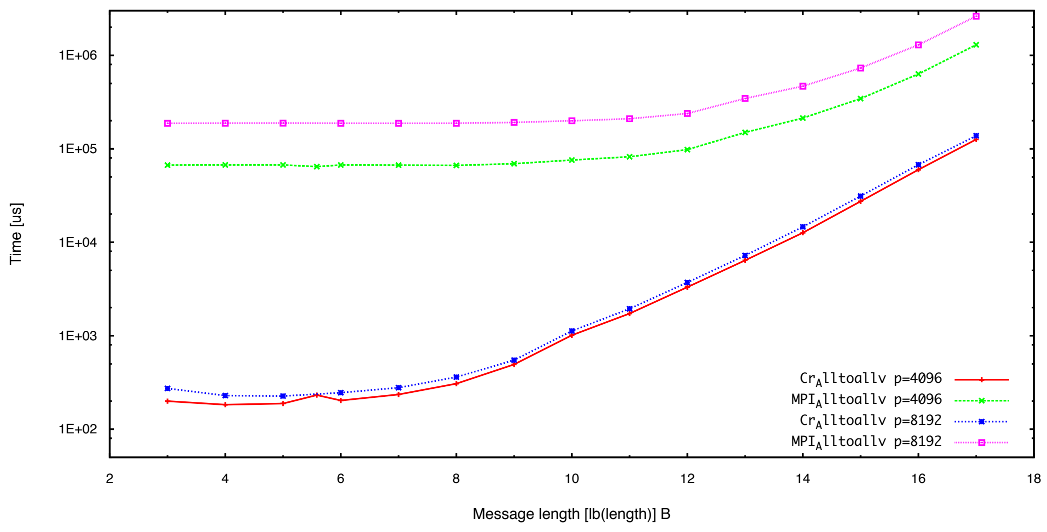


**Figure 11: Benchmark of personalized all-to-all communication implemented with the crystal router based function Cr_Alltoallv and the MPI function MPI_Alltoallv using MPI_IN_PLACE. Each process sends and receives data from 26 neighboring processes. The measurements have been executed with 4096 respectively 8192 processes.**

Figure 12 shows a comparison of the function MPI_Alltoallv as it is provided on the system with the crystal_router based implementation Cr_Alltoallv using the option MPI_IN_PLACE on 4096 and 8192 cores.

The function Cr_Alltoall is faster for all message lengths, however, the speed is much higher especially for short messages. There are achieved until 2.5 orders of magnitude of the runtime. Also on 1024 and 2048 cores are faster speeds up to two orders of magnitude reached as shown in Figure 13.

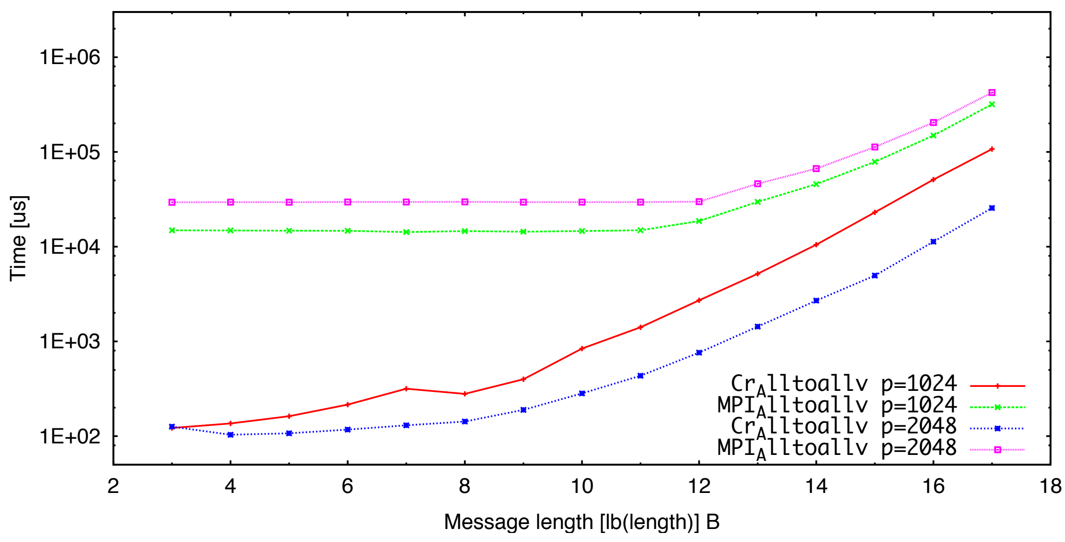


**Figure 12: Benchmark of personalized all-to-all communication implemented with the crystal router based function Cr_Alltoallv and the MPI function MPI_Alltoallv using MPI_IN_PLACE. Each process sends and receives data from 26 neighboring processes. The measurements have been executed with 1024 respectively 2048 processes.**

The comparison with runs using separate send and receive buffers in Figure 14 demonstrates that this way to use MPI_Alltoallv is much more efficient, while there is no large difference of the runtimes for Cr_Alltoallv. The speed difference between MPI_Alltoallv and Cr_Alltoallv is about one order of magnitude for short messages.

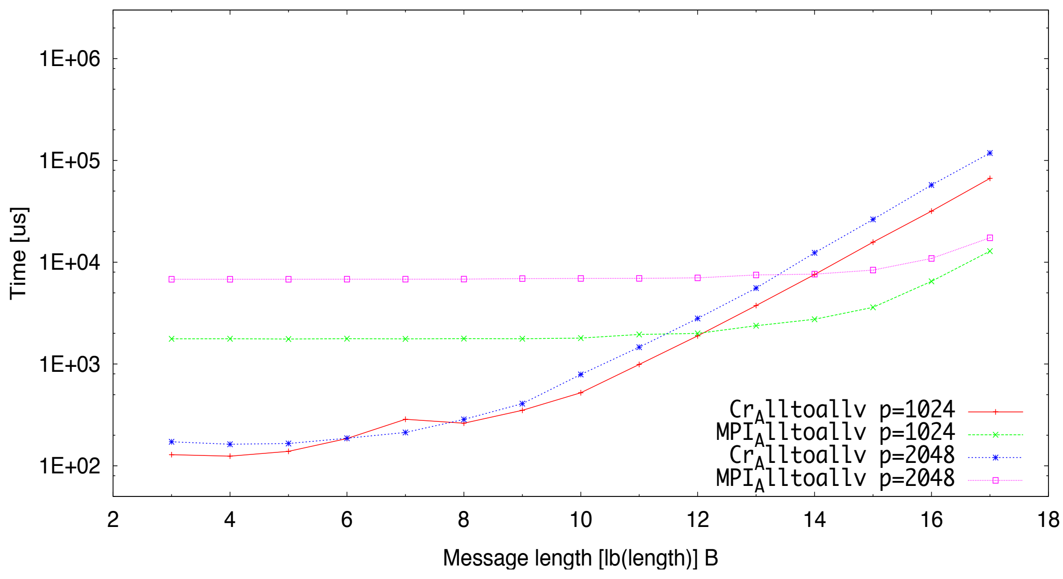


**Figure 13: Benchmark of personalized all-to-all communication implemented with the crystal router based function Cr_Alltoallv and the MPI function MPI_Alltoallv using separate send and receive buffers. Each process sends and receives data from 26 neighboring processes. The measurements have been executed with 1024 respectively 2048 processes.**

The crystal_router has been chosen as a central algorithm for the development of a communication module inside the adaptive runtime system because it shows a superior exchange performance especially for short messages up to 4 kilobyte and large-scale parallel runs on recent computer systems. It showed a uniform scaling over the whole range of job sizes. This is possible because it bundles short messages into larger packages that will be transferred at once. The influence of latency is reduced in that way, and MPI library optimizations with respect to the bandwidth of larger message lengths become useable for shorter messages too. The crystal_router is sensitive slightly to the distance of the communicating processes and to a larger extend to the number of communication partners per process, i.e. the degree of sparsity in the communication pattern. These comparatively small variations and the high overall efficiency that is achieved at the same time are an effect of the algorithm's properties. The message bundling and the algorithm design guarantee the message delivery within a fixed number of communication steps. Finally, the hypercube algorithm involves all nodes equally into the transport of messages during each communication step.

### 4.4.2 Performance Monitoring of MPI Applications Using *Event Flow Graphs*

In addition to developing a monitoring component with online introspection capabilities, WP3 has also been exploring innovative methods for exascale performance monitoring of MPI applications. More specifically, WP3 has investigated methods for efficient performance data storage. Typical performance analysis tools either collect lossless traces with time-stamped events ordered in time, or generate profile reports with aggregated statistics. Profiling methods are very scalable, however, they do not keep the temporary nature of the data. In addition, they can miss microscopic performance problems due to the summarization process. In contrast, tracing methods give the whole picture of what happened with the application but they are infeasible at an exascale level due to the amount of data generated. Thus, WP3 has been exploring a new approach for application characterization using *event flow graphs* [11], [12] which balances the low overhead of profiling methods with the lossless properties of tracing.

Event Flow Graphs are directed weighted graphs in where nodes represent the MPI calls performed by the process, and edges the transitions between those calls. In other words, the edges model the computation phases between two MPI calls. Thus, event flow graphs can keep the temporary nature of the events without storing any explicit temporal information such as timestamps. As these graphs keep the temporal order of events, they can serve as a compressed representation of event traces. We can reconstruct the ordered full sequence of MPI calls performed by the application by just traversing the graph from its initial to its final node.

We implemented this approach within the monitoring component of the runtime system, and tested it with several mini-applications of the NERSC-8/Trinity Benchmark suite [13]: AMG, an algebraic multigrid solver for linear systems on unstructured grids; GTC, a 3D Particle-in-cell code (PIC) with a non-spectral Poisson solver used for gyrokinetic particle simulation of turbulent transport in burning plasma; MILC, a code for simulating four dimensional SU(3) lattice gauge theory to study quantum chromodynamics (QCD); SNAP, a proxy application that models the performance of a modern discrete ordinates neutral particle transport application, PARTISN [14]; MiniDFT, a plane-wave DFT mini-kernel that computes self-consistent solutions for the Kohn-Sham equations; MiniFE, a mini-application that implements different kernels representative of implicit finite-element applications; MiniGhost, a mini-application that implements a difference stencil across a homogenous three dimensional domain.

The experiments were performed on a Cray XE6 with 2 twelve-core AMD MagnyCours at 2.1 GHz per node. The nodes are interconnected through a Cray Gemini Network, each of them having a total of 32 GB DDR3 memory. The benchmarks were compiled with Intel 12.1.5 and run using the small test case that is provided for each one of them.

The first experiment performed measured the overhead introduced into the benchmarks by the monitoring component when collecting performance information, generating the graphs, and writing those graphs to disk. The experiments were run using strong scaling for all the benchmarks except for SNAP, MILC and GTC. Figure 15 shows the percentage of overhead introduced over the total running time. As it can be seen in the figure, the overhead introduced to generate the event flow graphs is almost negligible, being always below 2%.

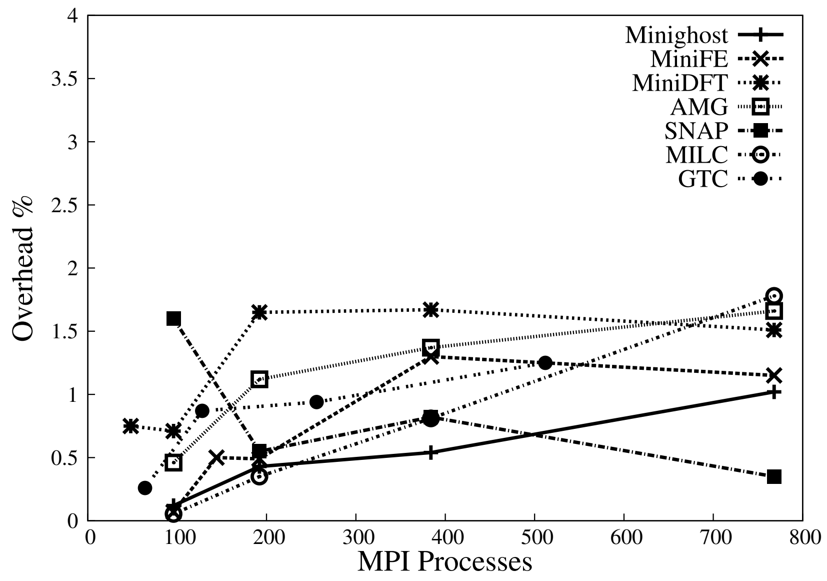


**Figure 14: Percentage of overhead over total running time introduced in the NERSC-8/Trinity benchmarks when generating their event flow graphs.**

The second experiment measured the achieved compression ratio for each benchmark in terms of file size between our event flow graphs and a trace generated by the monitoring component. In other words, how many times smaller are our event flow graph files compared to trace files. It is important to remark that both graphs and traces contained exactly the same amount of information for each MPI call: call name, bytes sent or received, communication partner rank and callsite. Furthermore, each one of the event flow graphs can generate exactly the same traces as the ones collected for the comparison. The following table contains the average compression ratio for each one of the benchmarks:

**Table 1: Average compression ration with different benchmarks**

| Benchmark | Ranks | Average compression ratio |
|---|---|---|
| AMG | 96 | 1.76 |
| GTC | 64 | 46.60 |
| MILC | 96 | 39.03 |
| SNAP | 96 | 119.23 |
| MiniDFT | 40 | 4.33 |
| MiniFE | 144 | 19.93 |
| MiniGhost | 96 | 4.85 |

The results in the table demonstrate that event flow graphs are good representations of compressed traces, showing compression ratios ranging from around 2% up to 119%. In terms of file size, the amount of disk space required to store the traces for a run with 96 cores of SNAP is 1.1GB whereas the space required for the event flow graphs is only 10 MB.

Finally, we performed another set of experiments to measure the increase ratio in file size of graphs and traces as we increase the number of simulation time steps, since one of the main aspects affecting the amount of data generated when monitoring applications is their running time. Figure 16 shows that traces increase linearly with the number of simulation steps whereas event flow graphs do not. For most of the benchmarks the small increment in the graph file size is caused by the addition of new edges to the graphs due to the execution of different call paths as the number of simulation steps increases. However, applications that execute the same loop over time such as the 5-stencil code [15] have constant event flow graph size irrespective of the number of simulation steps. For applications like that, the only difference between graphs from runs with different simulation times is their node cardinality.

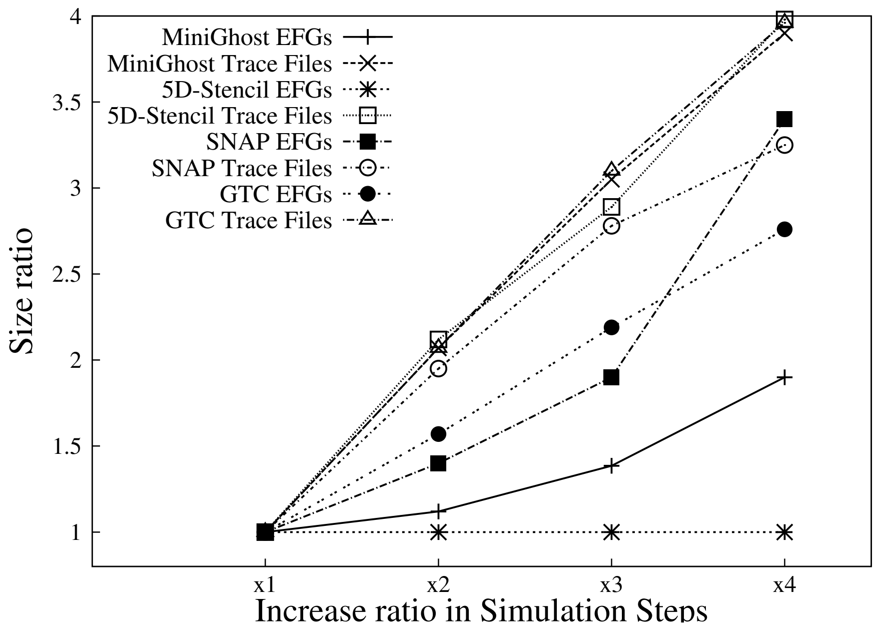


**Figure 15: Increase in file size when increasing simulation steps.**

In summary, event flow graphs combine the low overhead of profiling methods with the lossless information capabilities of tracing, thereby, being a good compressed representation of event traces. We evaluated this new approach with several mini-applications from the NERSC-8/Trinity Benchmark suite, achieving promising results of file compression ratios up to 119x with overheads below 2%. Moreover, the use of applications with longer running times would allow even better compression ratios because the same paths in the application are executed more times. Although this work is in an early stage, we believe it has strong potential to be a way towards developing performance analysis tools effective at an exascale level.

#### 4.4.2.1 Monitoring Component (Mon-C) - Outstanding Issues and Future Work

At the moment, the data accessible online via the Performance Introspection API is fixed and only provides accumulated statistics along time. In the future, we plan to extend the functionality of the monitoring component by first, allowing the runtime to configure online the data collected as it is being generated, and second, providing incremental data, that is, profile history or incremental profile snapshots. Thereby, removing the burden from the runtime to manage the incremental data collected since the last time it was accessed. For instance, the total time of a function per call, or certain loop metric per each loop iteration.

We also want to implement a global performance view for the whole application. In other words, the mechanisms to allow one process access directly the performance state of other processes.

The Performance Introspection API needs to be extended as well to query performance information per thread about OpenMP regions as the application runs. Moreover, the monitoring component should be extended to capture performance data from other programming models different than MPI such as PGAS languages or OpenMP Task extensions.

Finally, another aspect that requires an extended effort due to its complexity is the re-use of historical collected data to help the runtime in its decision making progress, as well as the automatic analysis of this data to detect performance bottlenecks. We want to keep exploring efficient methods for storing and obtaining knowledge from historical data with performance analysis purposes. First, we are going to extend the Performance Introspection API with metrics computed from previous runs. For instance, allowing the runtime to access runs the average time for a certain function in previous runs of an application. Second, built on top of the current experiences gained from the CRESTA project regarding task scheduling, we are going to explore what useful knowledge and performance trends can be extracted from historical performance data to help the runtime in its decision making progress for task scheduling.

##### *4.4.2.2 Event Flow Graphs for MPI Monitoring - Outstanding Issues and Future Work*

We will continue the study of event flow graphs in the monitoring and analysis of MPI parallel applications as it opens up many possibilities, from developing new tools based in the graph approach to the use of graphs for automatic performance analysis. First, we will explore the utilization of different algorithms for automatic graph analysis, for instance, detecting loops in the graphs and relating them to the application. Second, our current implementation of event flow graphs does not allow the reconstruction of traces with continuous data such as timestamps. Thus, we have started to explore statistical methods for reconstruction of sequences of continuous data, for example, hardware counters or timestamps. Finally, we also want to investigate inter-node trace compression across ranks. Our current version always generates one graph per process. However, it is usual in parallel applications that a set of processes has similar or identical behaviour. In such cases, the graphs generated by those processes will be similar as well, and thus, they can be compressed into a single graph that could be used to describe that whole set of processes with similar execution.

### 4.4.3 Outstanding Issues and Future Work

The implementation of an adaptive runtime system in CRESTA clearly confirms the expected benefits from such software for parallel applications. Methods for performance improvements can be generalized and implemented separately from concrete applications. Given the availability of an API allowing a non-intrusive introduction of the runtime system in parallel codes, performance improvements can be achieved with moderate effort and without the need of extended program refactoring. On the other hand, larger refactoring cannot be avoided in order to achieve a good match between the computer architecture and the software design. The runtime system cannot completely encapsulate and hide aspects of the computer architecture from the application. However, future work on the CRESTA runtime system can contribute to the development of efficient approaches for large-scale applications.

The current implementation focuses on MPI support. The use of other parallelization technologies in hybrid simulation codes is left to the application developer for the time being. Hybrid parallelization is, however, seen as a promising method for recent and upcoming parallel computer systems. The next step for the runtime system is therefore its extension in order to support MPI in combination with OpenMP, multi-threaded processes and OpenSHMEM with runtime services for dynamic load-balancing.

At first, the mapping calculation has been implemented as global optimization with a central master process. This implementation will last only for a limited time and needs to be complemented by a component based on distributed parallel algorithms in order to compensate the increasing complexity of the graph operations for mapping and scheduling for larger systems.

The current support for MPI will also be extended. The existing implementation uses a one-to-one relation between computational tasks and ranks within a certain load-balancing context. Future work will provide an efficient solution that can place an arbitrary number of computational tasks from one load-balancing context into one MPI process. One approach could be the use of MPI endpoints as they are discussed in the MPI forum.

## 5 Performance Analysis Tools

The optimization process for parallel applications usually consists of five steps. The first step is debugging and correctness checking to ensure a correct program. The second step is to get a coarse view on the application behavior and find program phases that contain potential bottlenecks with profiling and automatic trace analysis. These program phases can then be reviewed in detail with a visual performance analysis. The gained information can then be used to optimize and rerun the application.

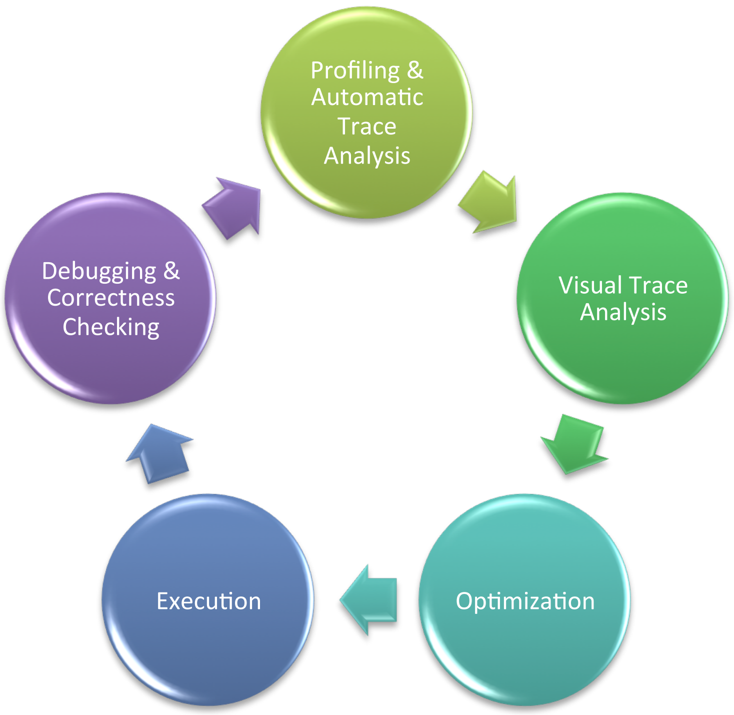

Profiling & Automatic Trace Analysis
Visual Trace Analysis
Optimization
Execution
Debugging & Correctness Checking

**Figure 16: Performance optimization and analysis workflow.**

### 5.1 Tracing New Paradigms and Energy

This section covers approaches to monitor and analyze new parallel paradigms and system metrics such as energy and network information.

#### 5.1.1 Tracing CoArray Fortran within the IFS Kernel

Partitioned Global Address Space (PGAS) models are available as library-based paradigms, e.g., Global Address Space Programming Interface (GASPI), SHMEM, as language extensions, e.g., UPC, Coarray Fortran (CAF).

To exchange data between the different memory locations PGAS languages use RMA (Remote Memory Access) operations as their underlying communication substrates. Therefore, we investigated one-sided communication models and developed a generic event model to record RMA operations in the OTF2 trace format for range of one-sided APIs and libraries. Within CRESTA the Coarray Fortran co-design team was established to investigate the possibilities and potentials of this PGAS language to overlap communication and computation within a world leading production application like ECMWF's Integrated Forecasting system (IFS). It turns out that the monitoring of Cray's Coarray Fortran fine granular operations will be only possible by using a source-to-source instrumentor or by indirect monitoring of the underlying communication library, i.e., monitoring of the Cray DMAPP library, due to the fact that the language constructs are processed in the compiler runtime. The same holds for the Cray UPC implementation.

On Cray systems Coarray Fortran and UPC routines make use of the libpgas library, which uses the DMAPP library as underlying communication library. The calls to this library can be intercepted with a library wrapping approach and one-sided communication operations can be recorded with the generic one-sided RMA event model (see Figure 17). Initialization and finalization with hierarchical unification can be done using MPI as underlying communication layer. Figure 18 shows the visualization of a short Coarray Fortran example with Vampir. It can be observed that there are tiny functions, which are called very frequently like for example dmapp_c_pset_test. For tiny functions, which are called very frequently, it is advisable to disable the detailed monitoring and to enable only profiling to prevent the monitoring system to be swamped by these functions or in worst case if the overhead is too high to disable the monitoring of this class of functions.

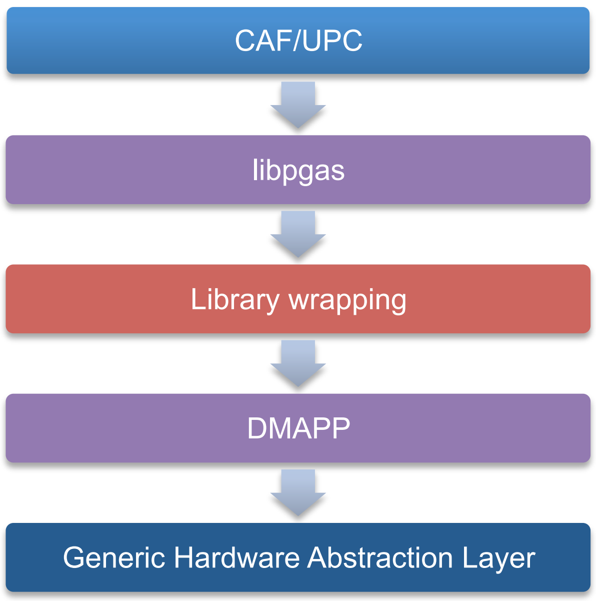


**Figure 17: Cray DMAPP software layers: interception of calls from the libpgas library to the DMAPP library by the library wrapping approach.**

The library wrapper for the DMAPP library can be created using vtlibwrappgen:

```
vtlibwrapgen -f dmapp_filter.txt –l \
    /opt/cray/dmapp/3.2.1-1.0400.3965.10.12.gem/lib64/libdmapp.so \
    -g DMAPP -o dmappwrap.c \
    /opt/cray/dmapp/3.2.1-1.0400.3965.10.12.gem/include/dmapp.h

vtlibwrapgen -v --build -o libvt_dmapp dmappwrap.c
```

To use the generated library wrapper it must by linked dynamically like this:

```
FC= vtf90
CC= vtcc
LIBS= -dynamic -lpgas-dmapp -lvt_dmapp \
    -L/opt/cray/cce/8.0.4/CC/x86-64/lib/x86-64
```

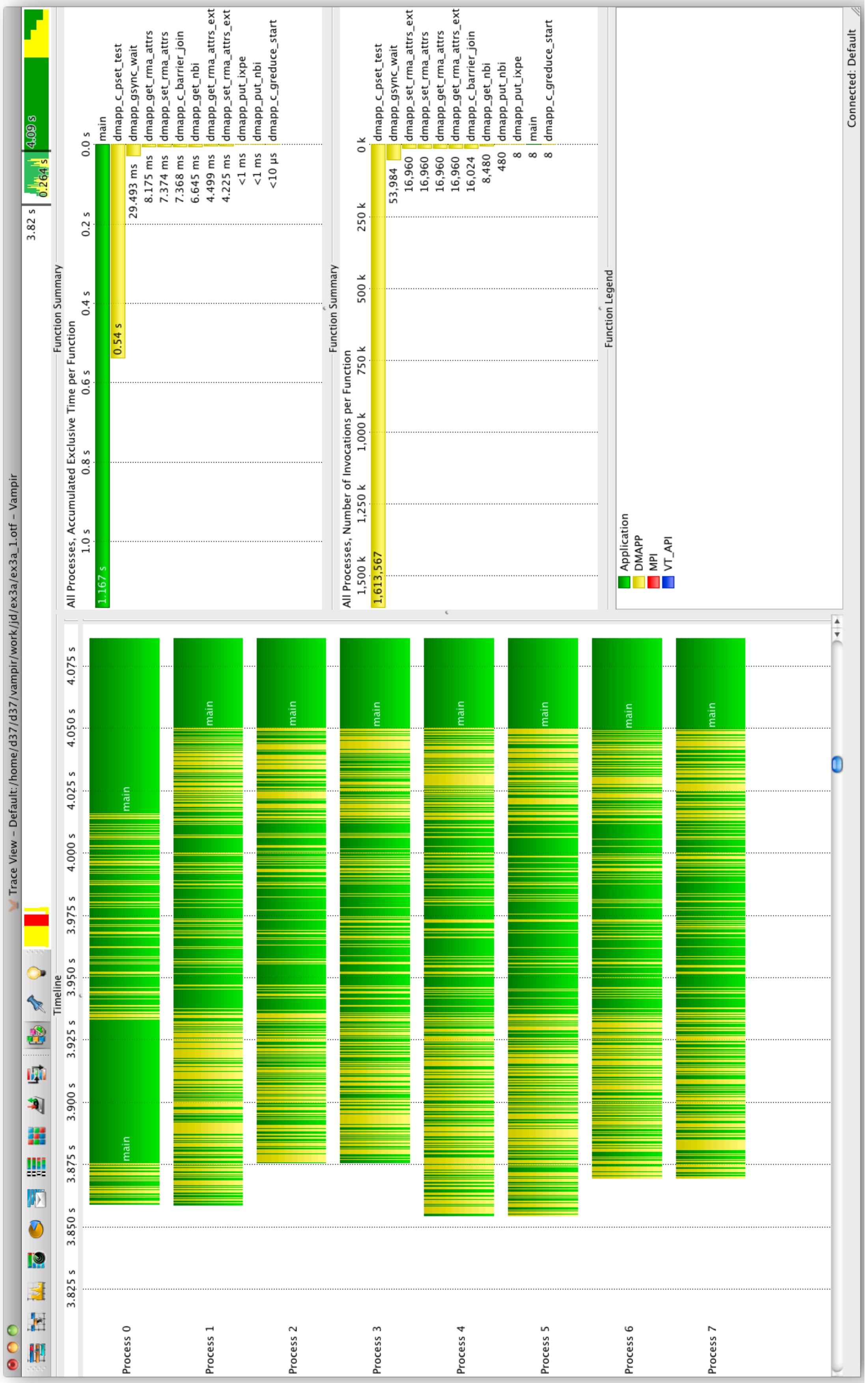

Trace View – Default: /home/d37/d37/vampir/work/jd/ex3a/ex3a_1.otf – Vampir
3.82 s
0.264 s
4.09 s
Timeline
3.825 s
3.850 s
3.875 s
3.900 s
3.925 s
3.950 s
3.975 s
4.000 s
4.025 s
4.050 s
4.075 s
Process 0
Process 1
Process 2
Process 3
Process 4
Process 5
Process 6
Process 7
main
Function Summary
All Processes, Accumulated Exclusive Time per Function
1.0 s
0.8 s
0.6 s
0.4 s
0.2 s
0.0 s
1.167 s main
0.54 s dmapp_c_pset_test
29.493 ms dmapp_gsync_wait
8.175 ms dmapp_get_rma_attrs
7.374 ms dmapp_set_rma_attrs
7.368 ms dmapp_c_barrier_join
6.645 ms dmapp_get_nbi
4.499 ms dmapp_get_rma_attrs_ext
4.225 ms dmapp_set_rma_attrs_ext
<1 ms dmapp_put_ixpe
<1 ms dmapp_put_nbi
<10 µs dmapp_c_greduce_start
All Processes, Number of Invocations per Function
1,500 k
1,250 k
1,000 k
750 k
500 k
250 k
0 k
1,613,567 dmapp_c_pset_test
53,984 dmapp_gsync_wait
16,960 dmapp_set_rma_attrs_ext
16,960 dmapp_set_rma_attrs
16,960 dmapp_get_rma_attrs
16,960 dmapp_get_rma_attrs_ext
16,024 dmapp_c_barrier_join
8,480 dmapp_get_nbi
480 dmapp_put_nbi
8 dmapp_put_ixpe
8 main
8 dmapp_c_greduce_start
Function Legend
🟩 Application
🟨 DMAPP
🟥 MPI
🟦 VT_API
Connected: Default

**Figure 18: Performance visualization of the Cray DMAPP communication library with Vampir. It is important to see that dmapp_c_pset_test is called very often and therefore should not be recorded in detail to reduce the overhead of the monitoring.**

### 5.1.2 Tracing OpenACC Usage within the Nekbone Kernel

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.

Host-side activities of OpenACC capable devices can be either monitored by instrumenting the library (if source code is available) or by using a shared library wrapper approach that uses the LD_PRELOAD mechanism.

Besides the host-based recording, some activities of the kernel can be monitored directly. For example, kernel execution and data transfers.

Monitoring of CUDA applications can be done either via the CUDA Profiling Tools Interface (CUPTI) or by the previously-mentioned 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. This topic is also covered in more detail in [20].

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. To enable the monitoring of these events the application has to be linked against the monitoring library and the following runtime environment variables must be set:

```
SCOREP_CUDA_ENABLE=kernel,memcpy,driver,concurrent
SCOREP_CUDA_BUFFER=3M
```

### 5.1.3 Tracing Energy Consumption

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.

We describe ways that users can monitor the energy and power consumption of their applications when running on the Cray XC supercomputer range. We exploit some of the new power measurement and control features that were introduced in the Cray Cascade-class architectures. This topic is also covered in more detail in [21].

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 on Cray platforms during the application measurement to run asynchronously per node.

To use the power monitoring plugin it must be build on the target system and the application must be instrumented at the desired level of detail. Setting the according environment variables activates this power monitoring plugin:

```
export SCOREP_METRIC_PLUGINS=pm_plugin
export SCOREP_METRIC_PM_PLUGIN=”all”
```

Figure 19 to Figure 21 show different visualizations of applications and benchmarks using the energy and power monitoring with Vampir.

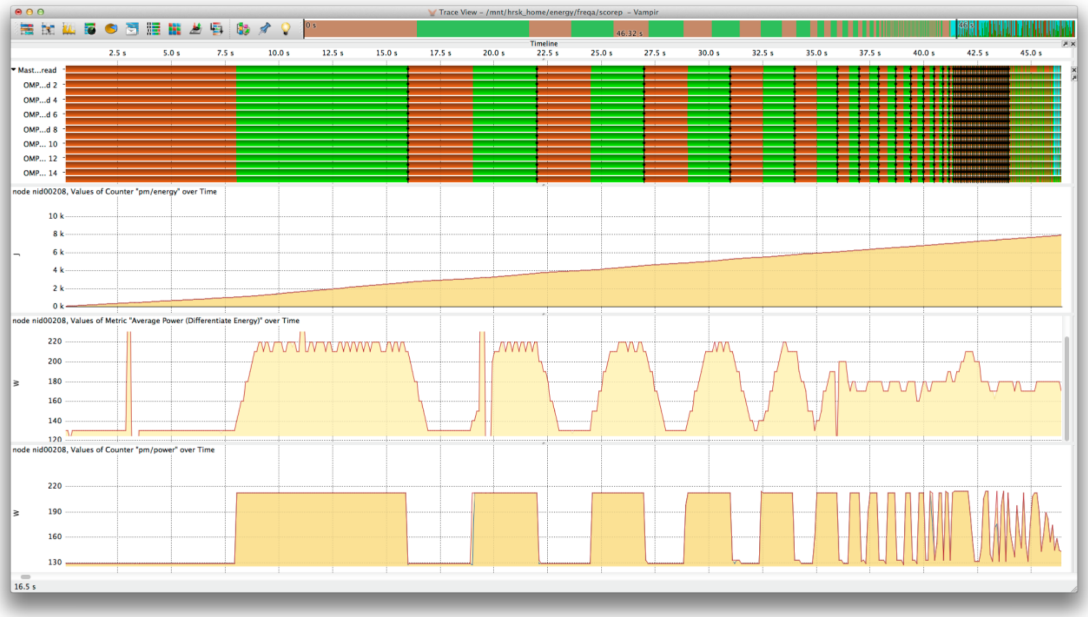

Trace View – /mnt/hrsk_home/energy/freqa/scorep – Vampir
Timeline
2.5 s
5.0 s
7.5 s
10.0 s
12.5 s
15.0 s
17.5 s
20.0 s
22.5 s
25.0 s
27.5 s
30.0 s
32.5 s
35.0 s
37.5 s
40.0 s
42.5 s
45.0 s
Mast...read
OMP...d 2
OMP...d 4
OMP...d 6
OMP...d 8
OMP... 10
OMP... 12
OMP... 14
node nid00208, Values of Counter "pm/energy" over Time
node nid00208, Values of Metric "Average Power (Differentiate Energy)" over Time
node nid00208, Values of Counter "pm/power" over Time
16.5 s

**Figure 19: Load-idle benchmark with color-coded visualization of the load-idle regions (topmost timeline, load-idle regions are colored in green respectively in brown) and corresponding energy (second timeline), average power derived from energy (third timeline), and instantaneous power information (lowest timeline) with Vampir.**

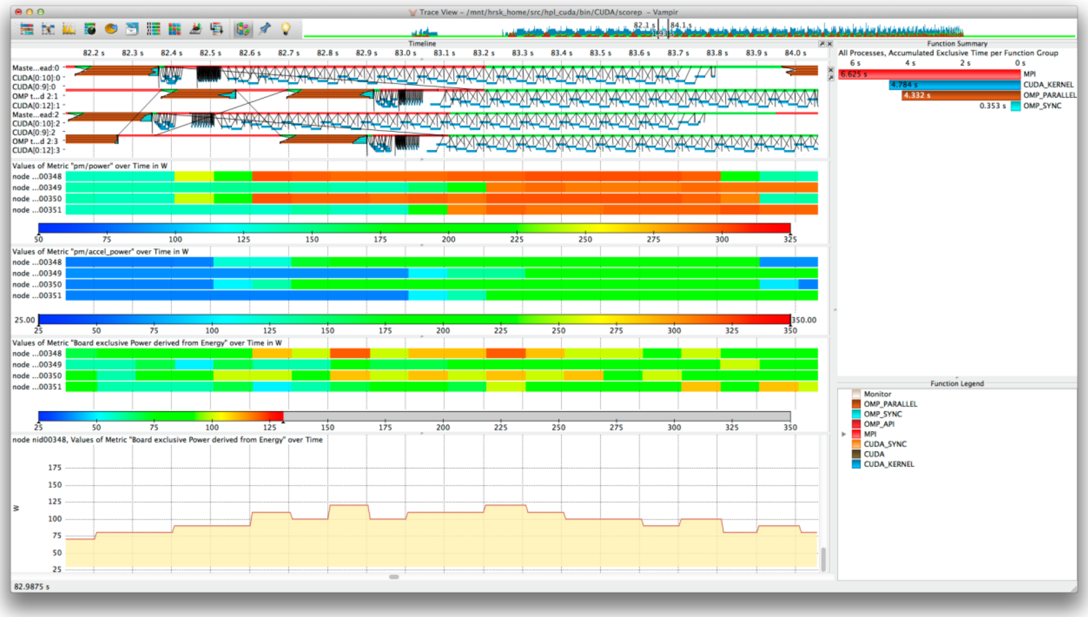

Trace View – /mnt/hrsk_home/src/hpl_cuda/bin/CUDA/scorep – Vampir
Timeline
Function Summary
All Processes, Accumulated Exclusive Time per Function Group
MPI
CUDA_KERNEL
OMP_PARALLEL
OMP_SYNC
Values of Metric "pm/power" over Time in W
Values of Metric "pm/accel_power" over Time in W
Values of Metric "Board exclusive Power derived from Energy" over Time in W
node nid00348, Values of Metric "Board exclusive Power derived from Energy" over Time
Function Legend
Monitor
OMP_PARALLEL
OMP_SYNC
OMP_API
MPI
CUDA_SYNC
CUDA
CUDA_KERNEL

Figure 20: Color-coded visualization of HPL CUDA with Vampir. The topmost timeline shows the behavior of the processes, threads, and CUDA streams over time for an interval of 2s. The second timeline displays the instantaneous node power over time. The third timeline displays the instantaneous graphic card power and the lowest timeline displays the board exclusive power without the graphic card derived from the energy.

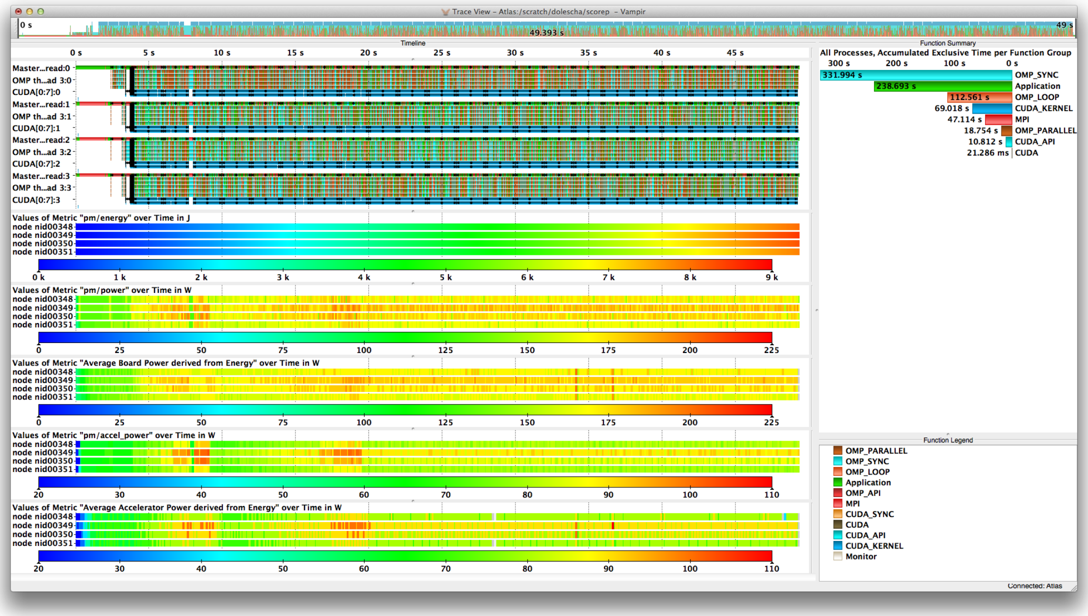

Trace View – Atlas:/scratch/dolescha/scorep – Vampir
Timeline
Function Summary
All Processes, Accumulated Exclusive Time per Function Group
331.994 s OMP_SYNC
238.693 s Application
112.561 s OMP_LOOP
69.018 s CUDA_KERNEL
47.114 s MPI
18.754 s OMP_PARALLEL
10.812 s CUDA_API
21.286 ms CUDA
Values of Metric "pm/energy" over Time in J
Values of Metric "pm/power" over Time in W
Values of Metric "Average Board Power derived from Energy" over Time in W
Values of Metric "pm/accel_power" over Time in W
Values of Metric "Average Accelerator Power derived from Energy" over Time in W
Function Legend
OMP_PARALLEL
OMP_SYNC
OMP_LOOP
Application
OMP_API
MPI
CUDA_SYNC
CUDA
CUDA_API
CUDA_KERNEL
Monitor
Connected: Atlas

Figure 21: 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) for an interval of 49.393s 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.

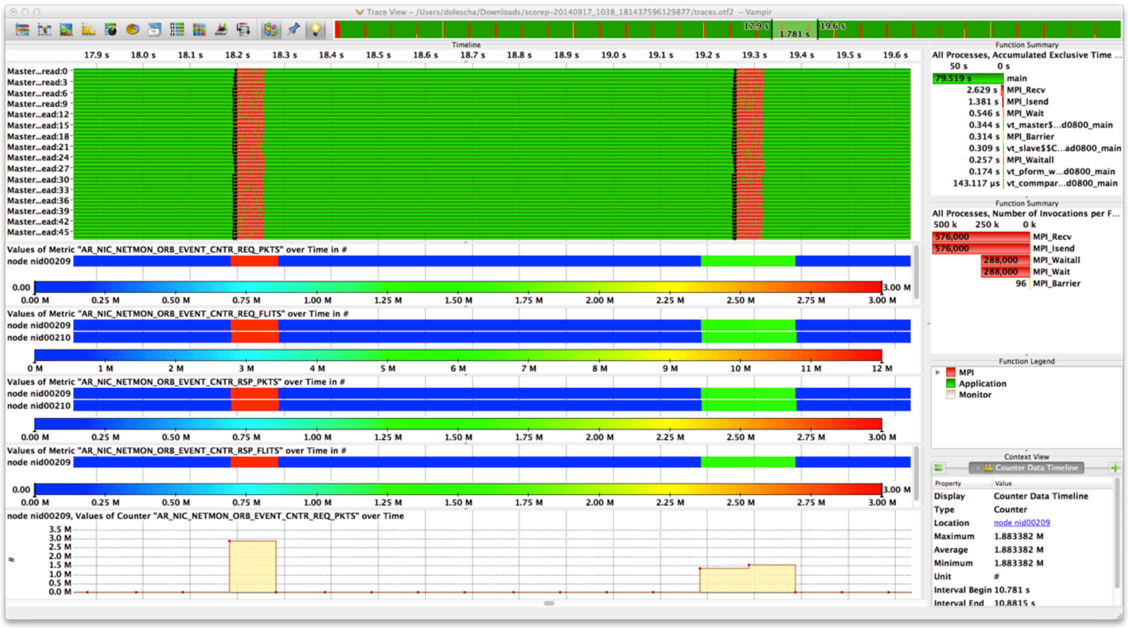
### 5.1.4 Tracing of Network Counters

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. In addition, the according environment variables must be set. However, the available counters may vary on each platform:

```
export SCOREP_METRIC_PLUGINS=APAPI
export \
    SCOREP_METRIC_APAPI="AR_NIC_NETMON_ORB_EVENT_CNTR_REQ_STALLED,\
    AR_NIC_NETMON_ORB_EVENT_CNTR_RSP_STALLED,\
    AR_NIC_NETMON_ORB_EVENT_CNTR_REQ_PKTS,\
    AR_NIC_NETMON_ORB_EVENT_CNTR_RSP_PKTS,\
    AR_NIC_NETMON_ORB_EVENT_CNTR_REQ_FLITS,\
    AR_NIC_NETMON_ORB_EVENT_CNTR_RSP_FLITS"
```

Figure 22 shows the correlation of application behavior and network information with Vampir.


Trace View – /Users/dolescha/Downloads/scorep-20140917_1038_181437596129877/traces.otf2 – Vampir
Timeline
Values of Metric "AR_NIC_NETMON_ORB_EVENT_CNTR_REQ_PKTS" over Time in #
Values of Metric "AR_NIC_NETMON_ORB_EVENT_CNTR_REQ_FLITS" over Time in #
Values of Metric "AR_NIC_NETMON_ORB_EVENT_CNTR_RSP_PKTS" over Time in #
Values of Metric "AR_NIC_NETMON_ORB_EVENT_CNTR_RSP_FLITS" over Time in #
node nid00209
node nid00210
node nid00209, Values of Counter "AR_NIC_NETMON_ORB_EVENT_CNTR_REQ_PKTS" over Time
Function Legend
🟥 MPI
🟩 Application
⬜ Monitor

Function Summary
All Processes, Accumulated Exclusive Time ...

| | |
|---|---|
| 79.519 s | main |
| 2.629 s | MPI_Recv |
| 1.381 s | MPI_Isend |
| 0.546 s | MPI_Wait |
| 0.344 s | vt_master$...d0800_main |
| 0.314 s | MPI_Barrier |
| 0.309 s | vt_slave$$C...ad0800_main |
| 0.257 s | MPI_Waitall |
| 0.174 s | vt_pform_w...d0800_main |
| 143.117 μs | vt_commpar...d0800_main |

Function Summary
All Processes, Number of Invocations per F...

| | |
|---|---|
| 576,000 | MPI_Recv |
| 576,000 | MPI_Isend |
| 288,000 | MPI_Waitall |
| 288,000 | MPI_Wait |
| 96 | MPI_Barrier |

Context View
Counter Data Timeline

| Property | Value |
|---|---|
| Display | Counter Data Timeline |
| Type | Counter |
| Location | node nid00209 |
| Maximum | 1.883382 M |
| Average | 1.883382 M |
| Minimum | 1.883382 M |
| Unit | # |
| Interval Begin | 10.781 s |
| Interval End | 10.8815 s |

**Figure 22: Vampir screenshot showing the correlated network activity.**

## 5.2 Selective Monitoring

Event tracing tools record each event of a parallel application in detail. Thus, it 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. We developed and evaluated three approaches to address the large amount of collected data, in particular, for massively parallel or long running applications. First, using different levels of detail by enabling or disabling certain parallel paradigms or prevent the instrumentation of functions that are usually inlined by the compiler. Second, applying a rewind within the record event stream to subsequently remove iterations that are not of interest and only keep those that represent deviating behavior. Third, remove highly frequent short-running functions calls that can overwhelm any recording memory buffer while in the same time contribute very less to the analysis and understanding of the overall application behavior (see [22]).

### 5.2.1 Monitoring of Different Levels of Details For Each Process

To compare different levels of details it is possible to build different instrumented versions of an application. For a multi-paradigm application like Gromacs this could be:

- Compiler instrumentation + MPI + OpenMP + CUDA,
- Compiler instrumentation with filters + MPI – OpenMP + CUDA,
- MPI + OpenMP + CUDA, or
- MPI + CUDA.

This can be achieved by setting the according instrumentation options in Score-P:

```
scorep --mpp=mpi --thread=omp:pomp_tpd
scorep --mpp=mpi --thread=omp:pomp_tpd --filter=<file>
scorep --mpp=mpi --thread=omp:pomp_tpd --nocompiler
scorep --mpp=mpi --thread=none --nocompiler
```

Currently the minimal instrumentation must contain MPI to get an entry point with MPI_Init and MPI_Finalize. In the future a wrapper that intercepts only MPI_Init and MPI_Finalize would reduce the minimal instrumentation further.

You can use aprun to launch the differently instrumented application in MPMD mode. Shell scripts can be used to set different environment for each version:

```
aprun -n pes [aprun_options] executable1 [args_ executable1] : \
      -n pes [aprun_options] executable2 [args_ executable2] : \
      -n pes [aprun_options] executable3 [args_ executable3]

aprun -n 12 ./app1 : -n 8 ./app2 : -n 32 ./app3
```

Figure 23 shows the visualization of classical monitoring. In comparison, Figure 24 shows the same application with different levels of detail for each node.

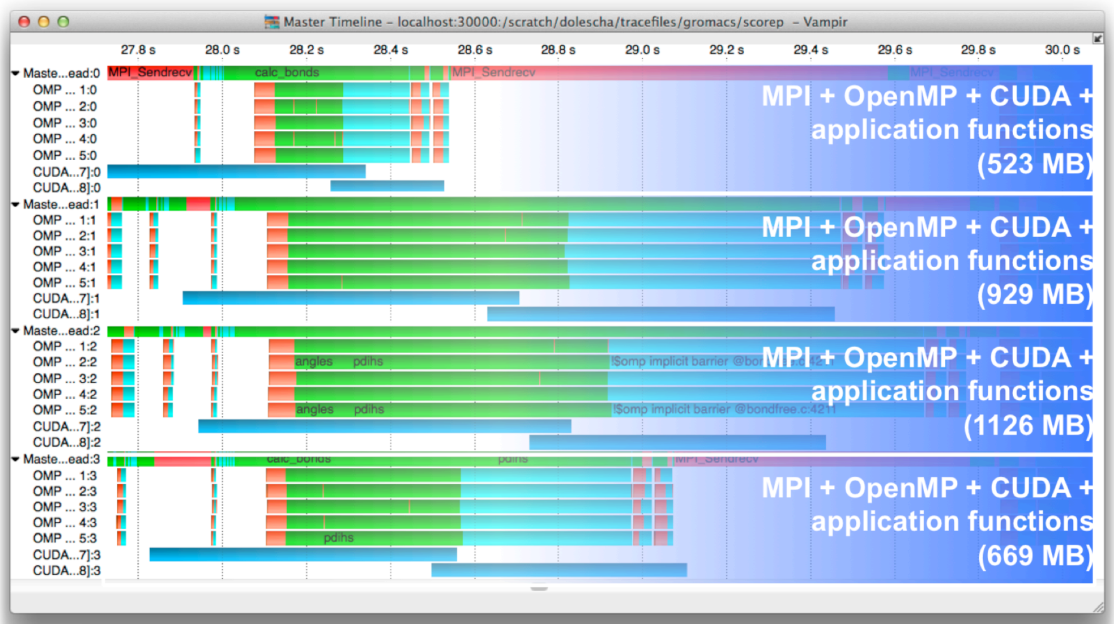


**Figure 23: Gromacs with four nodes each fully instrumented .**

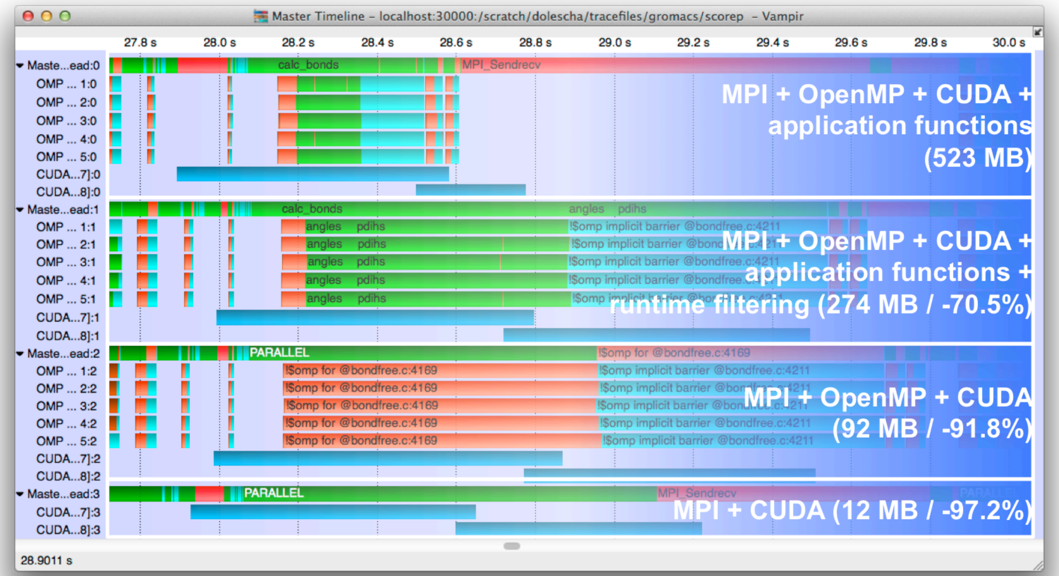

Master Timeline – localhost:30000:/scratch/dolescha/tracefiles/gromacs/scorep – Vampir
MPI + OpenMP + CUDA + application functions (523 MB)
MPI + OpenMP + CUDA + application functions + runtime filtering (274 MB / -70.5%)
MPI + OpenMP + CUDA (92 MB / -91.8%)
MPI + CUDA (12 MB / -97.2%)
calc_bonds
MPI_Sendrecv
PARALLEL
28.9011 s

**Figure 24: Gromacs with different levels of detail for each node. Reduction is relative to the according node in Figure 23.**

## 5.2.2 Selective Monitoring of Iterations

Selective monitoring is one approach to decrease the number of collected events. There are two main methods to select the recorded events: static and dynamic selection. For example, in iterative applications it is reasonable to avoid storing every single iteration, because most of them show more or less the same behavior. Therefore, 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 offset. 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.

Unfortunately, there are some analyses that will completely fail when even a single specific event is missing. One is the analysis of the communication behavior; especially for the Message Passing Interface (MPI). 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.

Thus, we developed a way to circumvent those restrictions: 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 selective monitoring techniques without sacrificing a detailed communication analysis.

To demonstrate the correct communication analysis, Figure 25 shows a screenshot of the visual analysis with Vampir. The fully monitored measurement can be seen on the upper half (white background); the measurement with selective monitoring on the lower half (blue background). The timeline view with the events over time on the horizontal axis and the processes on vertical axis is shown on the left side. Both views are zoomed to the size of approx. six iteration blocks, so the difference between both measurements can easily be seen. On the right side is a visualization of the average message data rate in a communication matrix. From the communication matrix it can be seen that the communication analysis was done correctly even with missing MPI events. This topic is also covered in more in detail in [23].

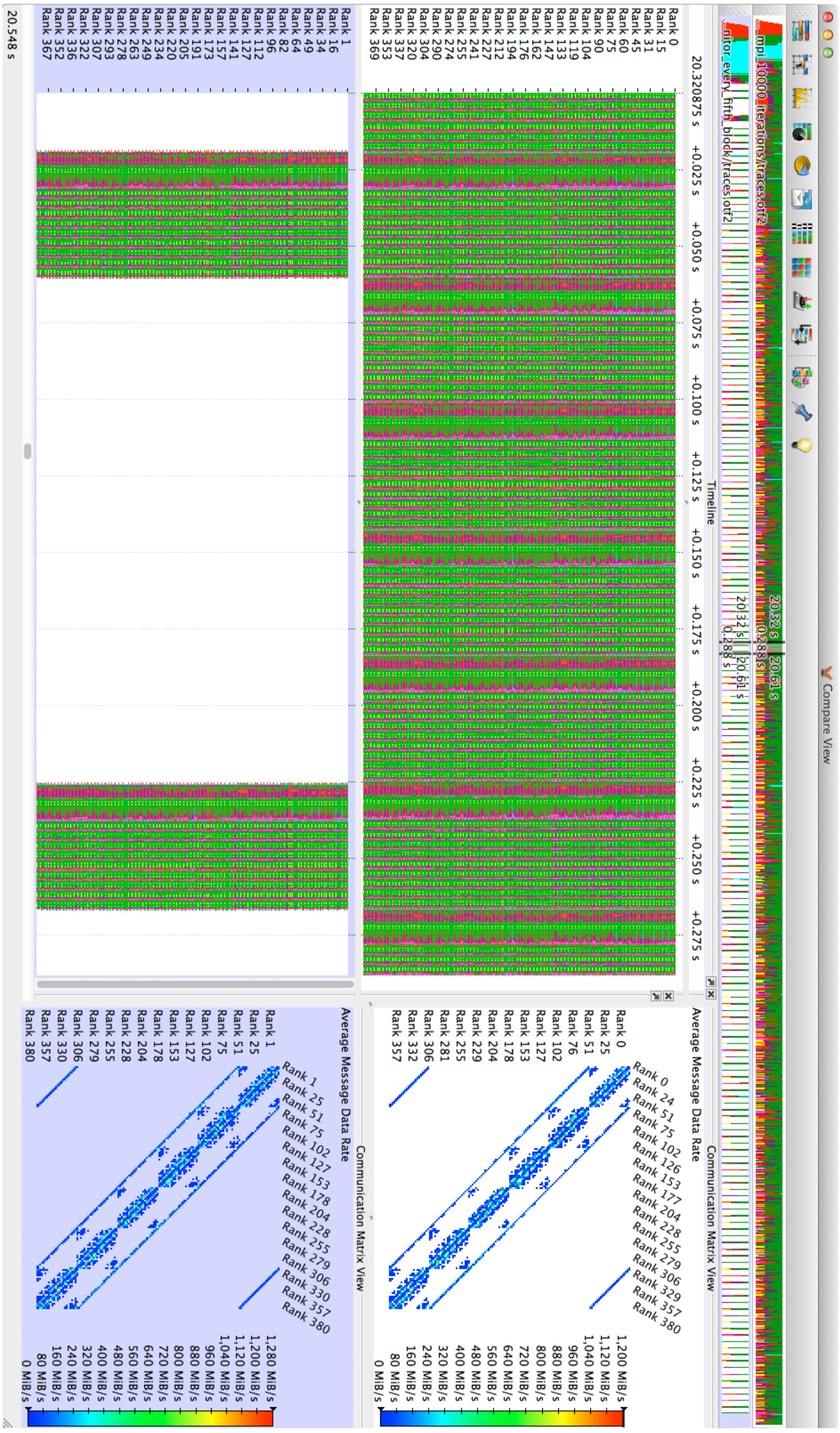


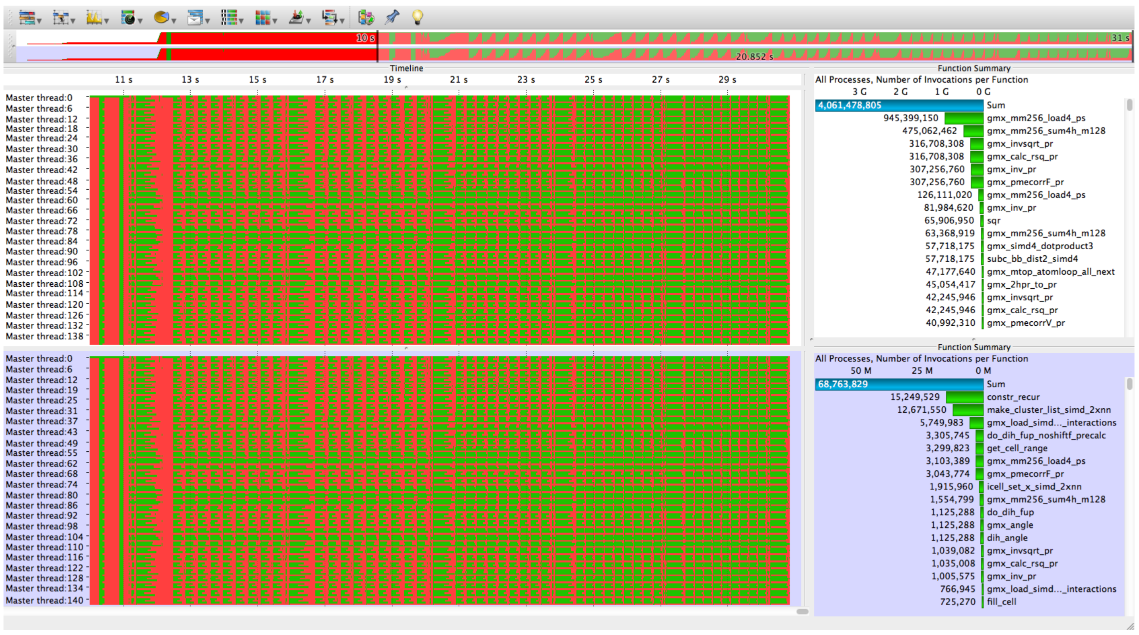
**Figure 25: Gromacs on 144 processes fully monitored (top) and selectively monitored (bottom) event trace visualized with Vampir zoomed in to about 6 iteration blocks.**

### 5.2.3 Selective Monitoring of Function Calls

Applying the same detail for each and every recorded event is prone to fail, especially, when tiny and often-used functions are monitored, e.g., inline functions and getter/setter class methods. Such highly frequent function calls can overwhelm any recording memory buffer while in the same time contribute very less to the analysis and understanding of the overall application behavior. We addressed the impact of high-frequency function calls and developed a method to minimize the amount of stored high-frequency functions while still keeping outliers that have an impact on the application behavior. We developed and applied a hierarchical memory buffer that is capable to discard recorded function calls when their duration is smaller than a pre-defined lower bound.

We used 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. Figure 26 and Figure 27 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). The timeline view with the events over time on the horizontal axis and the processes on vertical axis is shown on the left side. On the right side is a function summary showing the number of function invocations.

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 5 shows that the total number of function calls is reduced from about 4 billion to 68 million. Figure 6 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 [22].



Function Summary

All Processes, Number of Invocations per Function

```
3 G      2 G      1 G      0 G
4,061,478,805  Sum
  945,399,150  gmx_mm256_load4_ps
  475,062,462  gmx_mm256_sum4h_m128
  316,708,308  gmx_invsqrt_pr
  316,708,308  gmx_calc_rsq_pr
  307,256,760  gmx_inv_pr
  307,256,760  gmx_pmecorrF_pr
  126,111,020  gmx_mm256_load4_ps
   81,984,620  gmx_inv_pr
   65,906,950  sqr
   63,368,919  gmx_mm256_sum4h_m128
   57,718,175  gmx_simd4_dotproduct3
   57,718,175  subc_bb_dist2_simd4
   47,177,640  gmx_mtop_atomloop_all_next
   45,054,417  gmx_2hpr_to_pr
   42,245,946  gmx_invsqrt_pr
   42,245,946  gmx_calc_rsq_pr
   40,992,310  gmx_pmecorrV_pr
```

Function Summary

All Processes, Number of Invocations per Function

```
50 M      25 M      0 M
68,763,829  Sum
15,249,529  constr_recur
12,671,550  make_cluster_list_simd_2xnn
 5,749,983  gmx_load_simd..._interactions
 3,305,745  do_dih_fup_noshiftf_precalc
 3,299,823  get_cell_range
 3,103,389  gmx_mm256_load4_ps
 3,043,774  gmx_pmecorrF_pr
 1,915,960  icell_set_x_simd_2xnn
 1,554,799  gmx_mm256_sum4h_m128
 1,125,288  do_dih_fup
 1,125,288  gmx_angle
 1,125,288  dih_angle
 1,039,082  gmx_invsqrt_pr
 1,035,008  gmx_calc_rsq_pr
 1,005,575  gmx_inv_pr
   766,945  gmx_load_simd..._interactions
   725,270  fill_cell
```

**Figure 26: Gromacs on 144 processes fully monitored (top) and selectively monitored (bottom) event trace visualized with Vampir.**

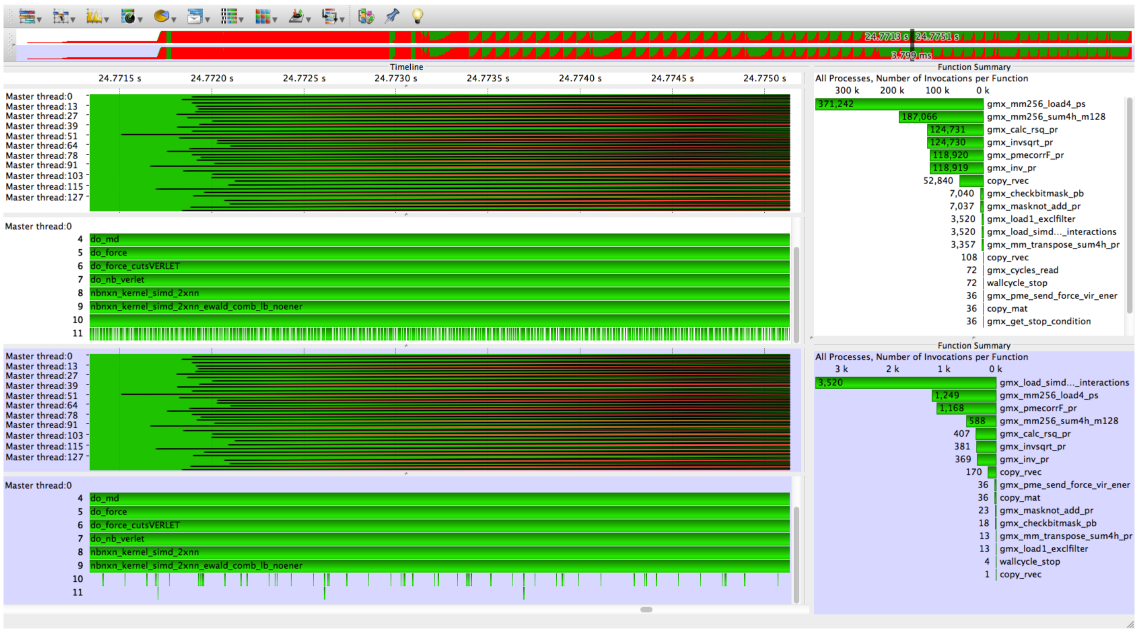


**Figure 27: Gromacs on 144 processes fully monitored (top) and selectively monitored (bottom) event trace visualized with Vampir zoomed to an application phase of about 3.8ms.**

## 5.3 Scalability

Event tracing delivers most detailed information allowing a profound post-mortem analysis of the parallel behavior. But, this comes with the cost of very large data volumes. Handling such a tremendous amount of data has always been a challenge in event tracing and is getting even more demanding with the rapid increase of processing elements. Since, the collected data is traditionally stored in one file per processing element, in particular, the rising number of resulting event trace files is one of the most urgent challenges. The limits of current parallel file systems allow handling only about ten or twenty thousand of parallel processes without any special treatment.

### 5.3.1 Using External Libraries

Writing one file per processing elements (e.g. check points or result files) does not scale to large systems since the sheer number of files overcharges the capabilities of today's file system meta-data servers. Two approaches that are dealing with the file system limitations and are applied to event tracing are SIONlib [25] and the I/O Forwarding Scalability Layer (IOFSL) [26]. 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 with 200.000 processes and Scalasca used SIONlib to record a full system run on the Jugeen system with almost 300.000 processes.

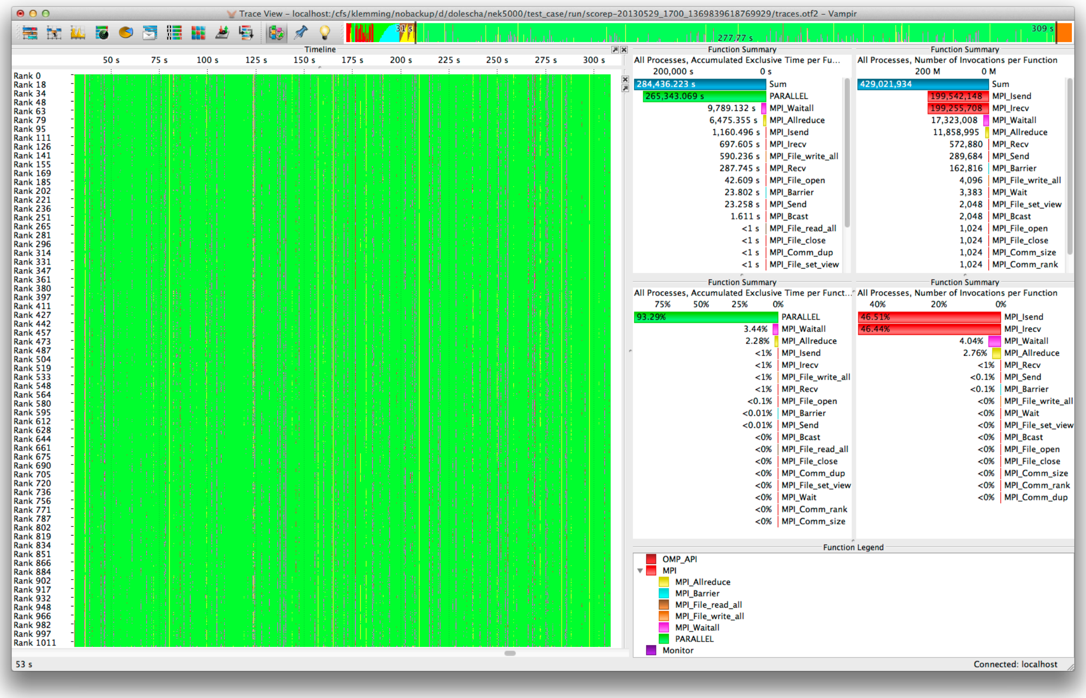
Since version 1.0 Score-P supports the usage of SIONlib but was restricted to pure MPI applications. With the upcoming release, Score-P 1.4 will support hybrid programs, as well.

## 5.4 Experiences with Target Applications

In general, we recommend starting to monitor the different applications with a coarse-grained approach, e.g., profiling or monitoring only the communication behavior. These approaches can be used to find program phases that contain potential bottlenecks or interesting behavior. These program phases can then be reviewed in detail with a complete analysis approach.

### 5.4.1 Nek5000

We monitored a MPI parallel version of Nek5000 with a jet data input set. The performance visualization with Vampir can be seen in Figure 28.



Trace View – localhost:/cfs/klemming/nobackup/d/dolescha/nek5000/test_case/run/scorep-20130529_1700_1369839618769929/traces.otf2 – Vampir

Function Summary

All Processes, Accumulated Exclusive Time per Fu...

| | |
|---|---|
| 284,436.223 s | Sum |
| 265,343.069 s | PARALLEL |
| 9,789.132 s | MPI_Waitall |
| 6,475.355 s | MPI_Allreduce |
| 1,160.496 s | MPI_Isend |
| 697.605 s | MPI_Irecv |
| 590.236 s | MPI_File_write_all |
| 287.745 s | MPI_Recv |
| 42.609 s | MPI_File_open |
| 23.802 s | MPI_Barrier |
| 23.258 s | MPI_Send |
| 1.611 s | MPI_Bcast |
| <1 s | MPI_File_read_all |
| <1 s | MPI_File_close |
| <1 s | MPI_Comm_dup |
| <1 s | MPI_File_set_view |

Function Summary

All Processes, Number of Invocations per Function

| | |
|---|---|
| 429,021,934 | Sum |
| 199,542,148 | MPI_Isend |
| 199,255,708 | MPI_Irecv |
| 17,323,008 | MPI_Waitall |
| 11,858,995 | MPI_Allreduce |
| 572,880 | MPI_Recv |
| 289,684 | MPI_Send |
| 162,816 | MPI_Barrier |
| 4,096 | MPI_File_write_all |
| 3,383 | MPI_Wait |
| 2,048 | MPI_File_set_view |
| 2,048 | MPI_Bcast |
| 1,024 | MPI_File_open |
| 1,024 | MPI_File_close |
| 1,024 | MPI_Comm_size |
| 1,024 | MPI_Comm_rank |

Function Summary

All Processes, Accumulated Exclusive Time per Funct...

| | |
|---|---|
| 93.29% | PARALLEL |
| 3.44% | MPI_Waitall |
| 2.28% | MPI_Allreduce |
| <1% | MPI_Isend |
| <1% | MPI_Irecv |
| <1% | MPI_File_write_all |
| <1% | MPI_Recv |
| <0.1% | MPI_File_open |
| <0.01% | MPI_Barrier |
| <0.01% | MPI_Send |
| <0% | MPI_Bcast |
| <0% | MPI_File_read_all |
| <0% | MPI_File_close |
| <0% | MPI_Comm_dup |
| <0% | MPI_File_set_view |
| <0% | MPI_Wait |
| <0% | MPI_Comm_rank |
| <0% | MPI_Comm_size |

Function Summary

All Processes, Number of Invocations per Function

| | |
|---|---|
| 46.51% | MPI_Isend |
| 46.44% | MPI_Irecv |
| 4.04% | MPI_Waitall |
| 2.76% | MPI_Allreduce |
| <1% | MPI_Recv |
| <0.1% | MPI_Send |
| <0.1% | MPI_Barrier |
| <0% | MPI_File_write_all |
| <0% | MPI_Wait |
| <0% | MPI_File_set_view |
| <0% | MPI_Bcast |
| <0% | MPI_File_open |
| <0% | MPI_File_close |
| <0% | MPI_Comm_size |
| <0% | MPI_Comm_rank |
| <0% | MPI_Comm_dup |

Function Legend

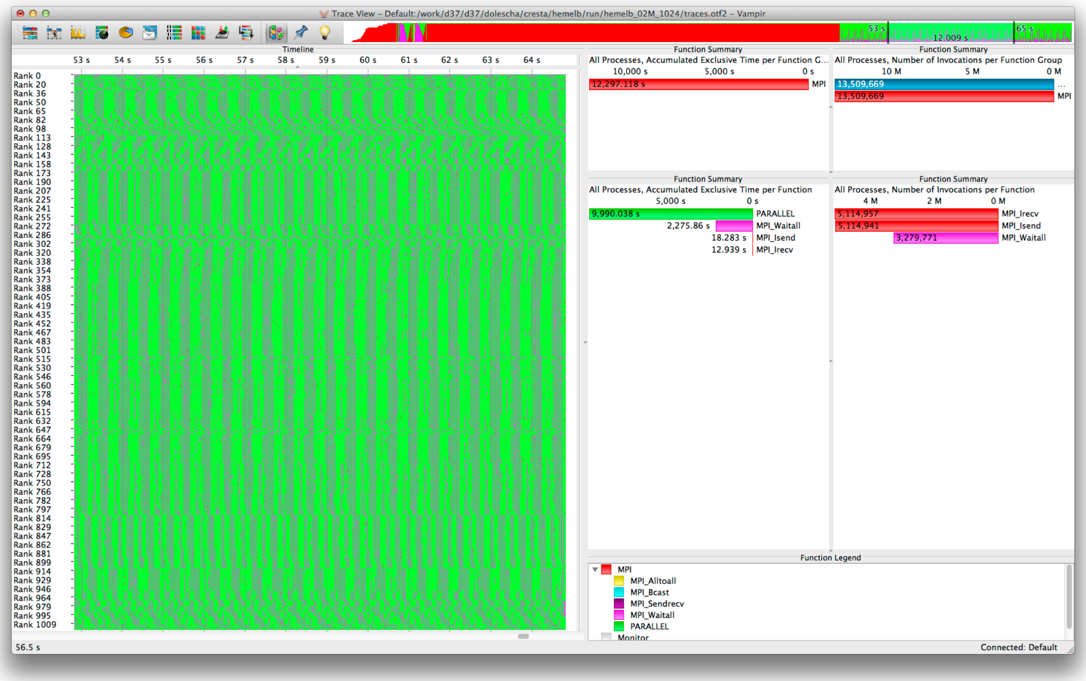
- 🟥 OMP_API
- 🟥 MPI
  - 🟨 MPI_Allreduce
  - 🟦 MPI_Barrier
  - 🟫 MPI_File_read_all
  - 🟧 MPI_File_write_all
  - 🟪 MPI_Waitall
  - 🟩 PARALLEL
- 🟪 Monitor

Connected: localhost

**Figure 28: Performance visualization of Nek5000 parallelized with MPI.**

### 5.4.2 HemeLB

We monitored a MPI parallel version of HemeLB. The performance visualization with Vampir can be seen in Figure 29.



Trace View – Default:/work/d37/d37/dolescha/cresta/hemelb/run/hemelb_02M_1024/traces.otf2 – Vampir

Function Summary

All Processes, Accumulated Exclusive Time per Function G...

| | |
|---|---|
| 12,297.118 s | MPI |

Function Summary

All Processes, Number of Invocations per Function Group

| | |
|---|---|
| 13,509,669 | ... |
| 13,509,669 | MPI |

Function Summary

All Processes, Accumulated Exclusive Time per Function

| | |
|---|---|
| 9,990.038 s | PARALLEL |
| 2,275.86 s | MPI_Waitall |
| 18.283 s | MPI_Isend |
| 12.939 s | MPI_Irecv |

Function Summary

All Processes, Number of Invocations per Function

| | |
|---|---|
| 5,114,957 | MPI_Irecv |
| 5,114,941 | MPI_Isend |
| 3,279,771 | MPI_Waitall |

Function Legend

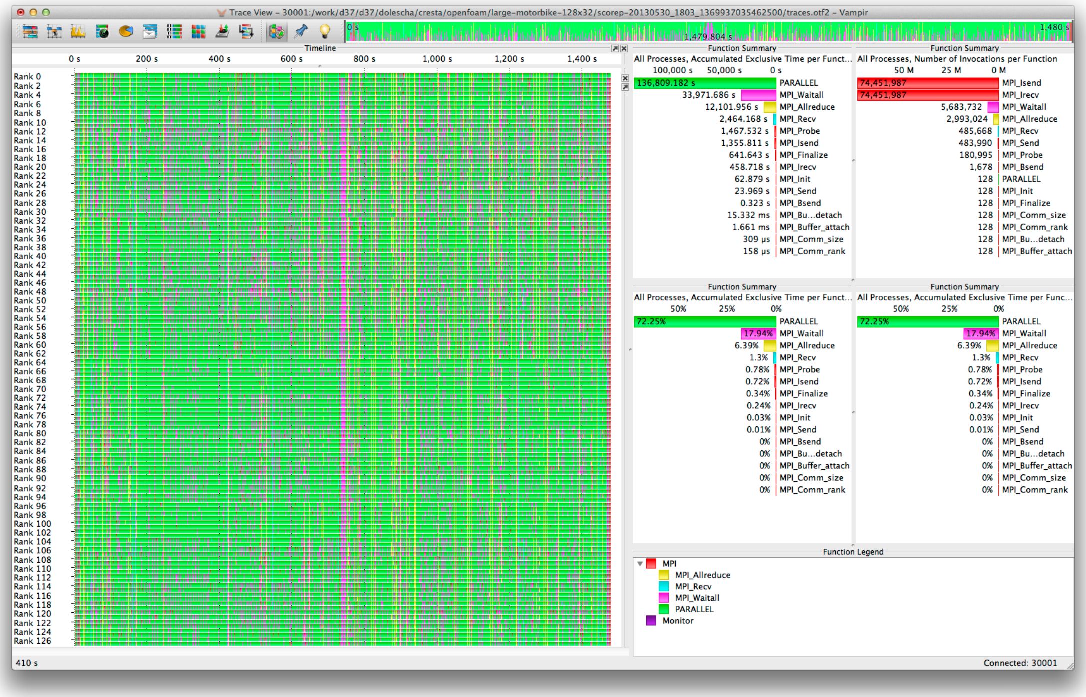
- 🟥 MPI
  - 🟨 MPI_Alltoall
  - 🟦 MPI_Bcast
  - 🟪 MPI_Sendrecv
  - 🟪 MPI_Waitall
  - 🟩 PARALLEL
- ⬜ Monitor

Connected: Default

**Figure 29: Performance visualization with Vampir of HemeLB parallelized with MPI.**

### 5.4.3 OpenFoam

We monitored an MPI parallel version of OpenFoam including the I/O interactions. The performance visualization with Vampir can be seen in Figure 30.



Trace View – 30001:/work/d37/d37/dolescha/cresta/openfoam/large-motorbike-128x32/scorep-20130530_1803_1369937035462500/traces.otf2 – Vampir

Function Summary — All Processes, Accumulated Exclusive Time per Funct…

| Time | Function |
|---|---|
| 136,809.182 s | PARALLEL |
| 33,971.686 s | MPI_Waitall |
| 12,101.956 s | MPI_Allreduce |
| 2,464.168 s | MPI_Recv |
| 1,467.532 s | MPI_Probe |
| 1,355.811 s | MPI_Isend |
| 641.643 s | MPI_Finalize |
| 458.718 s | MPI_Irecv |
| 62.879 s | MPI_Init |
| 23.969 s | MPI_Send |
| 0.323 s | MPI_Bsend |
| 15.332 ms | MPI_Bu…detach |
| 1.661 ms | MPI_Buffer_attach |
| 309 µs | MPI_Comm_size |
| 158 µs | MPI_Comm_rank |

Function Summary — All Processes, Number of Invocations per Function

| Invocations | Function |
|---|---|
| 74,451,987 | MPI_Isend |
| 74,451,987 | MPI_Irecv |
| 5,683,732 | MPI_Waitall |
| 2,993,024 | MPI_Allreduce |
| 485,668 | MPI_Recv |
| 483,990 | MPI_Send |
| 180,995 | MPI_Probe |
| 1,678 | MPI_Bsend |
| 128 | PARALLEL |
| 128 | MPI_Init |
| 128 | MPI_Finalize |
| 128 | MPI_Comm_size |
| 128 | MPI_Comm_rank |
| 128 | MPI_Bu…detach |
| 128 | MPI_Buffer_attach |

Function Summary — All Processes, Accumulated Exclusive Time per Funct…

| Share | Function |
|---|---|
| 72.25% | PARALLEL |
| 17.94% | MPI_Waitall |
| 6.39% | MPI_Allreduce |
| 1.3% | MPI_Recv |
| 0.78% | MPI_Probe |
| 0.72% | MPI_Isend |
| 0.34% | MPI_Finalize |
| 0.24% | MPI_Irecv |
| 0.03% | MPI_Init |
| 0.01% | MPI_Send |
| 0% | MPI_Bsend |
| 0% | MPI_Bu…detach |
| 0% | MPI_Buffer_attach |
| 0% | MPI_Comm_size |
| 0% | MPI_Comm_rank |

Function Summary — All Processes, Accumulated Exclusive Time per Func…

| Share | Function |
|---|---|
| 72.25% | PARALLEL |
| 17.94% | MPI_Waitall |
| 6.39% | MPI_Allreduce |
| 1.3% | MPI_Recv |
| 0.78% | MPI_Probe |
| 0.72% | MPI_Isend |
| 0.34% | MPI_Finalize |
| 0.24% | MPI_Irecv |
| 0.03% | MPI_Init |
| 0.01% | MPI_Send |
| 0% | MPI_Bsend |
| 0% | MPI_Bu…detach |
| 0% | MPI_Buffer_attach |
| 0% | MPI_Comm_size |
| 0% | MPI_Comm_rank |

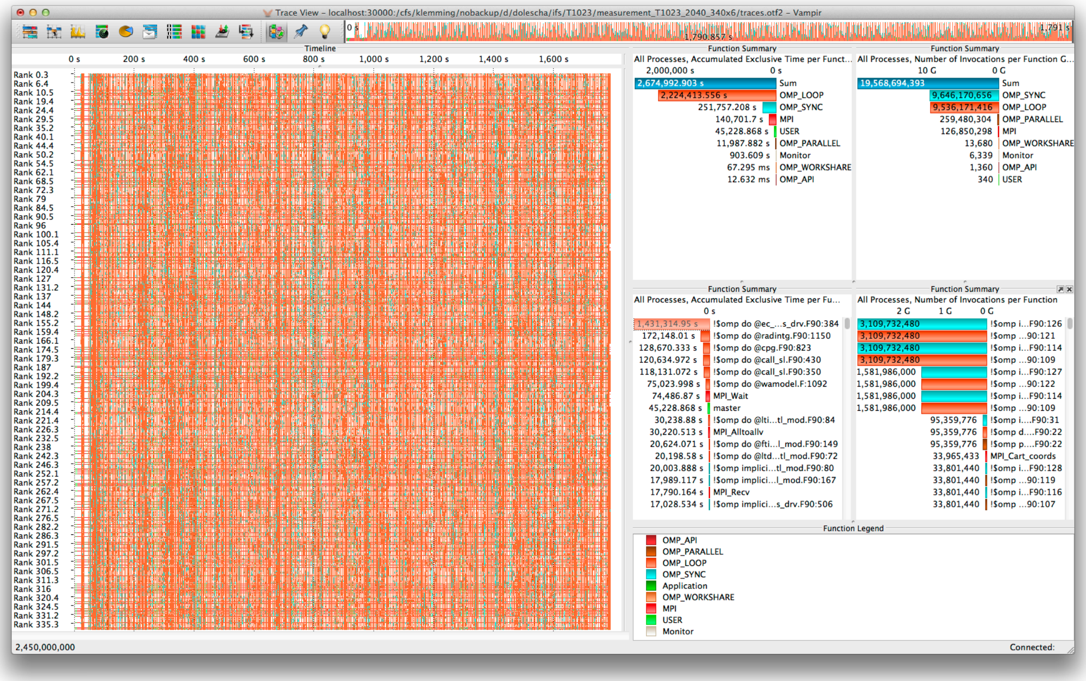
Function Legend: 🟥 MPI, 🟨 MPI_Allreduce, 🟦 MPI_Recv, 🟪 MPI_Waitall, 🟩 PARALLEL, 🟪 Monitor

Connected: 30001

**Figure 30: Performance visualization with Vampir of OpenFoam parallelized with MPI.**

### 5.4.4 IFS

We monitored a hybrid version of IFS parallelized with MPI and OpenMP running with dataset T1023. The performance visualization of the application behavior with Vampir can be seen in Figure 31. Figure 32 shows the analysis of an un-optimized communication pattern where all processes wait on rank zero (see also [24]).



Trace View – localhost:30000:/cfs/klemming/nobackup/d/dolescha/ifs/T1023/measurement_T1023_2040_340x6/traces.otf2 – Vampir

Function Summary — All Processes, Accumulated Exclusive Time per Funct…

| Time | Function |
|---|---|
| 2,674,992.903 s | Sum |
| 2,224,413.556 s | OMP_LOOP |
| 251,757.208 s | OMP_SYNC |
| 140,701.7 s | MPI |
| 45,228.868 s | USER |
| 11,987.882 s | OMP_PARALLEL |
| 903.609 s | Monitor |
| 67.295 ms | OMP_WORKSHARE |
| 12.632 ms | OMP_API |

Function Summary — All Processes, Number of Invocations per Function G…

| Invocations | Function |
|---|---|
| 19,568,694,393 | Sum |
| 9,646,170,656 | OMP_SYNC |
| 9,536,171,416 | OMP_LOOP |
| 259,480,304 | OMP_PARALLEL |
| 126,850,298 | MPI |
| 13,680 | OMP_WORKSHARE |
| 6,339 | Monitor |
| 1,360 | OMP_API |
| 340 | USER |

Function Summary — All Processes, Accumulated Exclusive Time per Fu…

| Time | Function |
|---|---|
| 1,431,314.95 s | !$omp do @ec_…s_drv.F90:384 |
| 172,148.01 s | !$omp do @radintg.F90:1150 |
| 128,670.333 s | !$omp do @cpg.F90:823 |
| 120,634.972 s | !$omp do @call_sl.F90:430 |
| 118,131.072 s | !$omp do @call_sl.F90:350 |
| 75,023.998 s | !$omp do @wamodel.F:1092 |
| 74,486.87 s | MPI_Wait |
| 45,228.868 s | master |
| 30,238.88 s | !$omp do @lti…tl_mod.F90:84 |
| 30,220.513 s | MPI_Alltoallv |
| 20,624.071 s | !$omp do @fti…l_mod.F90:149 |
| 20,198.58 s | !$omp do @ltd…tl_mod.F90:72 |
| 20,003.888 s | !$omp implici…tl_mod.F90:80 |
| 17,989.117 s | !$omp implici…l_mod.F90:167 |
| 17,790.164 s | MPI_Recv |
| 17,028.534 s | !$omp implici…s_drv.F90:506 |

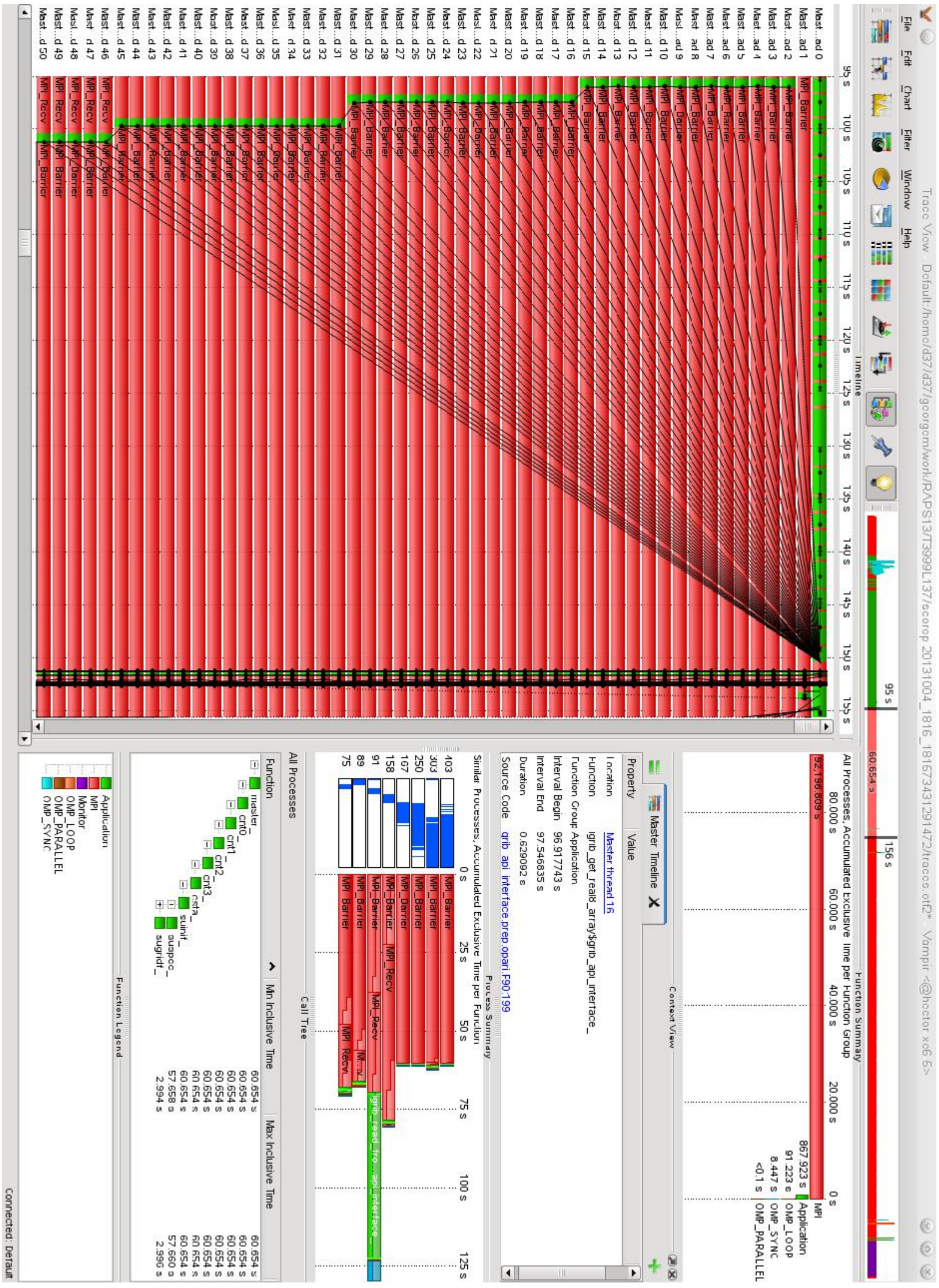
Function Summary — All Processes, Number of Invocations per Function

| Invocations | Function |
|---|---|
| 3,109,732,480 | !$omp i…F90:126 |
| 3,109,732,480 | !$omp …90:121 |
| 3,109,732,480 | !$omp i…F90:114 |
| 3,109,732,480 | !$omp …90:109 |
| 1,581,986,000 | !$omp i…F90:127 |
| 1,581,986,000 | !$omp …90:122 |
| 1,581,986,000 | !$omp i…F90:114 |
| 1,581,986,000 | !$omp …90:109 |
| 95,359,776 | !$omp i….F90:31 |
| 95,359,776 | !$omp d….F90:22 |
| 95,359,776 | !$omp p….F90:22 |
| 33,965,433 | MPI_Cart_coords |
| 33,801,440 | !$omp i…F90:128 |
| 33,801,440 | !$omp …90:119 |
| 33,801,440 | !$omp i…F90:116 |
| 33,801,440 | !$omp …90:107 |

Function Legend: 🟥 OMP_API, 🟫 OMP_PARALLEL, 🟧 OMP_LOOP, 🟦 OMP_SYNC, 🟩 Application, 🟧 OMP_WORKSHARE, 🟥 MPI, 🟩 USER, ⬜ Monitor

Connected:

**Figure 31: Performance visualization with Vampir of a hybrid IFS T1023 run.**


Trace View - Default:/home/d37/d37/gcorgcm/work/RAPS13/T399L137/scorep_20131004_1816_1815734313914172/traces.otf2* - Vampir <@hector xc6 6>
File Edit Chart Filter Window Help

95 s 100 s 105 s 110 s 115 s 120 s 125 s 130 s 135 s 140 s 145 s 150 s 155 s

All Processes, Accumulated Exclusive Time per Function Group
867.923 s MPI
91.223 s Application
8.447 s OMP_LOOP
<0.1 s OMP_SYNC
OMP_PARALLEL

Similar Processes, Accumulated Exclusive Time per Function

Application
MPI
Monitor
OMP_LOOP
OMP_PARALLEL
OMP_SYNC

Context View — Master Timeline

| Property | Value |
|---|---|
| Location | Master thread 16 |
| Function | ifgrib_get_real8_array$grib_api_interface_ |
| Function Group | Application |
| Interval Begin | 96.917743 s |
| Interval End | 97.546835 s |
| Duration | 0.629092 s |
| Source Code | grib_api_interface.prep.opari.F90:199 |

Call Tree — All Processes

| Function | Min Inclusive Time | Max Inclusive Time |
|---|---|---|
| master_ | 60.654 s | 60.654 s |
| cnt0_ | 60.654 s | 60.654 s |
| cnt1_ | 60.654 s | 60.654 s |
| cnt2_ | 60.654 s | 60.654 s |
| cnt3_ | 60.654 s | 60.654 s |
| cnt4_ | 60.654 s | 60.654 s |
| stepo_ | 60.654 s | 60.654 s |
| suinif_ | 57.658 s | 57.660 s |
| suspecg_ | 2.994 s | 2.996 s |

**Figure 32: Performance visualization with Vampir of IFS parallelized with MPI. The communication analysis uncovers an un-optimized communication pattern where all processes wait for rank zero.**

### 5.4.5 Gromacs

Within the coarse-grained analysis of Gromacs we detected that Gromacs uses a lot of tiny short-running functions such as getter/setter class methods and helper functions. While these tiny functions are usually automatically inlined by the compiler, the automatic instrumentation for tracing prevents the inlining. By itself, this provides event tracing tools the opportunity to record an application's behavior very detailed. However, if these functions are heavily used they might overwhelm the capacity of the recording memory buffers. While the recording of high-frequency functions enables a complete analysis, usually, they contribute very little to the analysis and understanding of the overall application behavior.

Next to the tremendous amount of tracing data that is generated by these tiny function calls, the overhead of monitoring these functions introduces even more bias. Within a monitoring overhead study for Gromacs, using MPI parallelization with 144 processes, built-in fftw, 10000 iterations, running on a Cray XC30, and monitored with Score-P 1.3b, we instrument Gromacs in three different ways and executed these versions to investigate the impact of instrumentation on runtime overhead. The first version (a) of Gromacs uses the native and common compiler function instrumentation, i.e., each function will be instrumented and function inlining is prevented. For a second version (b) of Gromacs, we intend to prevent instrumentation of all inline functions. Therefore, we compare the set of symbols of 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. For Gromacs the size of I is 1781. A third version (c) of Gromacs uses selective compiler instrumentation with an extension of filter (b) by the fifteen most frequently called functions.

The runtime and overhead results of these three different instrumented versions of Gromacs are presented in Table 2. As reference for this study we used the original and unmodified version of Gromacs. To approximate the overhead for entering and leaving of each instrumented function for the fully instrumented version we used a runtime filter that excludes all functions from recording. For this scenario we only reached 26.9 % of the original performance, i.e., even with our dynamic duration filtering technique on a fully compiler-instrumented application like Gromacs the achieved performance has only little significance for a later performance analysis. Recording all events either by a profiling or tracing approach for this fully instrumented application without any runtime filtering makes the situation even worse. In this case we only reach 1% of the original performance, the approximate size of the out coming trace file would be 18 T Byte, and we need at least a monitoring buffer of about 180GByte for each process to avoid any disturbing I/O operations. With these reference values we can conclude that the common-used default function-compiler instrumentation is the basic cause for a decreased performance and absolutely inappropriate for a detailed performance analysis. For the second version, the selective compiler instrumentation that prevents instrumentation of inline functions, we reach a performance of 92.9% by using a runtime filtering of each function (overhead for entering and leaving the instrumented functions), and respectively 67.8% of the original performance for the recording of all instrumented functions. With the last version, which uses the extended instrumentation filtering specification, we were able to increase the performance to 79% of the original performance while recording the instrumented functions in detail. The resulting size of the complete trace is still 37GByte with a total of 1,412,518,862 events. This topic is also covered in more detail in [22].

| Gromacs instrumentation version | Walltime | Gromacs' internal performance metric | Rel. performance |
|---|---:|---:|---:|
| Original unmodified version | 14.98s | 57.683ns/day | 100.0% |
| Fully-compiler-instrumented (a) with runtime filtering | 55.64s | 15.531ns/day | 26.9% |
| Fully-compiler instrumented (a) with profiling | 1483.1s | 0.583ns/day | 1.0% |
| Selective-compiler instrumentation (b) with runtime filtering | 16.12s | 53.610ns/day | 92.9% |
| Selective-compiler instrumentation (b) with tracing | 22.11s | 39.094ns/day | 67.8% |
| Extended selective-compiler instrumentation (c) with tracing | 18.95s | 45.598ns/day | 79.0% |

**Table 2: Monitoring overhead study for different instrumented versions of Gromacs.**

This study led to the selective monitoring approaches demonstrated in Section 5.2. In addition, Gromacs was recorded with multiple paradigms such as MPI, OpenMP, CUDA, and energy counters simultaneously to capture the complete application behavior (See Section 5.2.1).

## 5.5 Outstanding Issues and Future Work

This section lists current restrictions and outstanding features that will be covered in future releases:

The approach to wrap the DMAPP library to capture UPC behavior via the libpgas library (see Section 5.1.1) is realized as a prototype within VampirTrace. This feature is scheduled for a Score-P version greater than 1.4.

The approach to capture the OpenACC usage within an application is currently realized with an LD_PRELOAD mechanism and the CUPTI interface since the OpenACC tool interface is not standardized yet. The PGI compiler from version 14.9 provides a preliminary interface. Its usage is currently implemented in a branch version of Score-P and scheduled for a Score-P release version greater than 1.4.

For the monitoring of energy and power consumption (see Section 5.1.3) an additional thread is forked to run on the CPU set of process zero. Thus, the sampling frequency of the energy and power sources is important. However, these sources usually have a refresh rate of 10Hz or lower. Thus, the sampling frequency of these sources can be kept low as well.

The monitoring of different levels of detail for different processes (see Section 5.2.1) currently requires a lot of manual work. The future idea is to build and run the different versions automatically and use a profiling run to determine optimal instrumentation. This feature is not scheduled for Score-P so far.

While the rewind feature for a selective monitoring of iterations (see Section 5.2.2) is implemented in Score-P the dynamic runtime criteria are not implemented yet. This feature is not scheduled for Score-P so far.

The selective monitoring of function calls (see Section 5.2.3) relies on the prototype implementation of the OTFX tracing library. A release version of OTFX and an according integration into Score-P is not scheduled so far.

# 6 Debuggers

During the CRESTA project, each of the applications was invited to submit their impressions of the needs of debugging, for current usage and for their future usage – including which platforms and programming models would be of evolving interest. The results of this were summarized in Year 1 deliverables.

During Year 2 and Year 3, more active co-design activities were pursued with ECMWF and UCL.

In particular, the IFS experimentation with Coarray Fortran was assisted by debugging of Cray CAF by Allinea tools. Feedback on scalability, usability and integration with the bespoke user workflow at ECWMF was received and helped to direct modifications and new workflow oriented tool perspectives.

The second major focus, UCL HemeLB ultimately provided a significant opportunity for the tools to prove their value as the HemeLB hit an unexpected roadblock which was resolved by tools.

## 6.1 Allinea Tools

Allinea Software develops software tools HPC developers – including Allinea DDT, the parallel debugger, and Allinea MAP, the parallel profiler.

Both tools use a scalable tree for command of tool daemons – this has been used for debugging 700,000 core jobs, and frequently sees use at over 100,000 cores.

Within HPC software, progress can stumble due to two unpredictable interruptions: defects and performance. Both are beyond ordinary comprehension at scale. How can a bug be fixed if that bug only arises at (say) 100,000 cores? How can the performance of an application be understood when the behaviour at 100 cores is no indicator of the behaviour at 100,000?  Old tricks such as print statement debugging, or timer-printing profiling, do not help.

The challenges to solve these problems are (1) to provide low overhead tools – enabling an application to run within typical resources at the typical problem size; and (2) to convey the problem that the application has, even at extreme scale. For debugging, how can the differences (needles in haystacks) be identified; for profiling, how can typical issues such as poor balance, or bad I/O be best shown?

### 6.1.1 Allinea DDT

Allinea DDT is the scalable parallel debugger used by 70% of the top HPC centers – and present on the larger systems within the CRESTA project. The tool was made available to CRESTA participants for the duration of the project at maximum scale.

It uses its scalable control tree to handle the largest applications, stepping, or setting breakpoints in fractions of a second even on the largest machines. One major innovation in the tool is that it presents data to the user that helps to highlight the differences.

### 6.1.2 Allinea MAP

Complementing Allinea DDT,is the Allinea MAP tool – which we also chose to apply to co-design codes.

Allinea MAP is a sampling based profiler – and thus tackles performance in a different manner to tracing oriented profiling as seen with Vampir. Both approaches provide valuable insight.

The profiler is able to execute applications without requiring instrumentation or recompilation. It aims to present information that can help with the majority of performance problems.

Allinea MAP is an adaptive sampling profiler, adapting the frequency of samples over time, which keeps perturbation to a minimum. Sampling records the process stack, counters of communication, time, memory usage, I/O and the CPU instruction types. This enables source line correlation of information such as the achieved level of vectorization within a code. Codes typically experience considerably lower than 5% performance impact. The key to scalability is to realize that a profiler does not need to save everything - only what is necessary to understand the problem. Samples from each process are merged through the tree at the end of the job, retaining stacks, and min-max, standard deviation and mean of MPI, I/O and CPU metrics. Our contribution is in scalable visualization techniques – for example, a timeline in Allinea MAP shows the min, max, and mean – with the shading of the metric line indicating the standard deviation – enabling balance across processes to be understood.

## 6.2 HemeLB with Allinea Tools

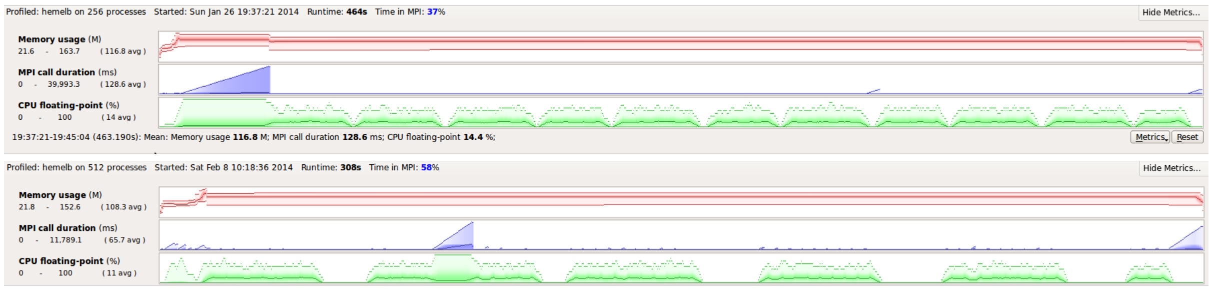
Over 100 of the systems in the Top 500 list in November 2013 had greater than 30,000 cores – and hence codes that wish to be "scalable" must scale to multiple thousands – or multiple tens of thousands of cores.

The reality of many large systems is that very few applications strive to achieve this scalability. HemeLB was an exception: the goal of reaching scale has a direct impact on the simulation that can be undertaken. In order to simulate a most significant area of the brain, the Circle of Willis, 50,000 cores and higher were highly desirable.

We will focus on one case during this quest for scalability – and a significant outcome.

### 6.2.1 Initial Performance Profiling with Allinea MAP

When we first deployed the application, Allinea MAP detected low percentages of vectorization – visible immediately on our timeline. Our user error is common with codes shared across the community - it is important to have quick methods to identify simple mistakes. After resolving this through reconfiguration, a second pattern was identified.

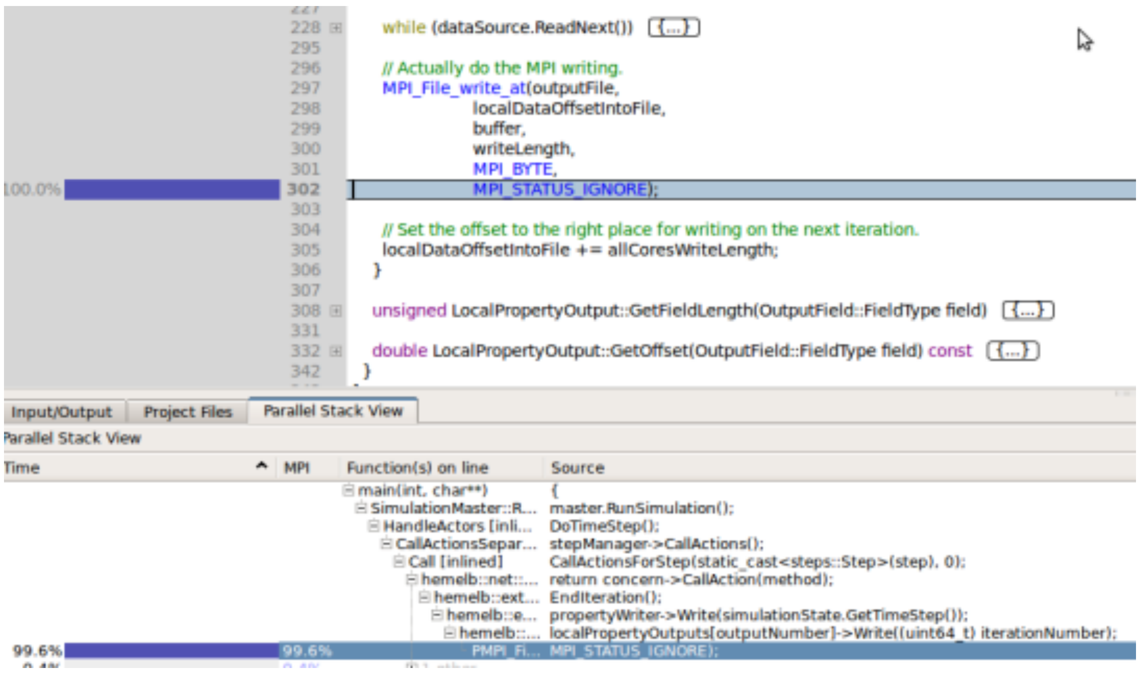


```
Profiled: hemelb on 256 processes   Started: Sun Jan 26 19:37:21 2014   Runtime: 464s   Time in MPI: 37%
Memory usage (M)        21.6  -  163.7     ( 116.8 avg )
MPI call duration (ms)  0  -  39,993.3     ( 128.6 avg )
CPU floating-point (%)  0  -  100          ( 14 avg )
19:37:21-19:45:04 (463.190s): Mean: Memory usage 116.8 M; MPI call duration 128.6 ms; CPU floating-point 14.4 %;

Profiled: hemelb on 512 processes   Started: Sat Feb 8 10:18:36 2014   Runtime: 308s   Time in MPI: 58%
Memory usage (M)        21.8  -  152.6     ( 108.3 avg )
MPI call duration (ms)  0  -  11,789.1     ( 65.7 avg )
CPU floating-point (%)  0  -  100          ( 11 avg )
```

**Figure 33: The increased flat-lining/troughs in the CPU floating point between 256 and 512 process cases**

In Figure 34, note the lower green timelines – representing amount of CPU floating point over time (with min,max and mean across all processes) - the runs at 256 processes and 512 processes are superimposed vertically. The pattern of interest is the "troughs" of CPU floating point operations (the zero-height points on the green line) – which have increased in significance at 512 processes, and represent greater share of time. The two glitches in MPI for each case (blue sawtooth edges) – are also being investigated.



```
            while (dataSource.ReadNext()) {...}

            // Actually do the MPI writing.
            MPI_File_write_at(outputFile,
                              localDataOffsetIntoFile,
                              buffer,
                              writeLength,
                              MPI_BYTE,
100.0%                        MPI_STATUS_IGNORE);

            // Set the offset to the right place for writing on the next iteration.
            localDataOffsetIntoFile += allCoresWriteLength;
          }

          unsigned LocalPropertyOutput::GetFieldLength(OutputField::FieldType field) {...}

          double LocalPropertyOutput::GetOffset(OutputField::FieldType field) const {...}
        }
```

```
Time             MPI     Function(s) on line        Source
                         main(int, char**)          {
                          SimulationMaster::R...    master.RunSimulation();
                           HandleActors [inli...    DoTimeStep();
                            CallActionsSepar...     stepManager->CallActions();
                             Call [inlined]         CallActionsForStep(static_cast<steps::Step>(step), 0);
                              hemelb::net::...      return concern->CallAction(method);
                               hemelb::ext...       EndIteration();
                                hemelb::e...        propertyWriter->Write(simulationState.GetTimeStep());
                                 hemelb::...        localPropertyOutputs[outputNumber]->Write((uint64_t) iterationNumber);
99.6%            99.6%             PMPI_Fi...       MPI_STATUS_IGNORE);
```

Figure 34: Zooming into the source code in a trough

The inline source code and the parallel stack display revealed the issue to be I/O. Reducing the frequency of periodic stores restored performance and enhanced scalability. This is leading the team at UCL to have a better understanding of the impact of I/O on the code on larger systems.

In this case, fixes were obvious and did not require deeper insight, but in other cases this will help to detect where problems lie and enable the use of targeted insight tools within the CRESTA framework such as TU Dresden's Vampir to explore specific MPI usage patterns within a smaller more tractable part of the application.

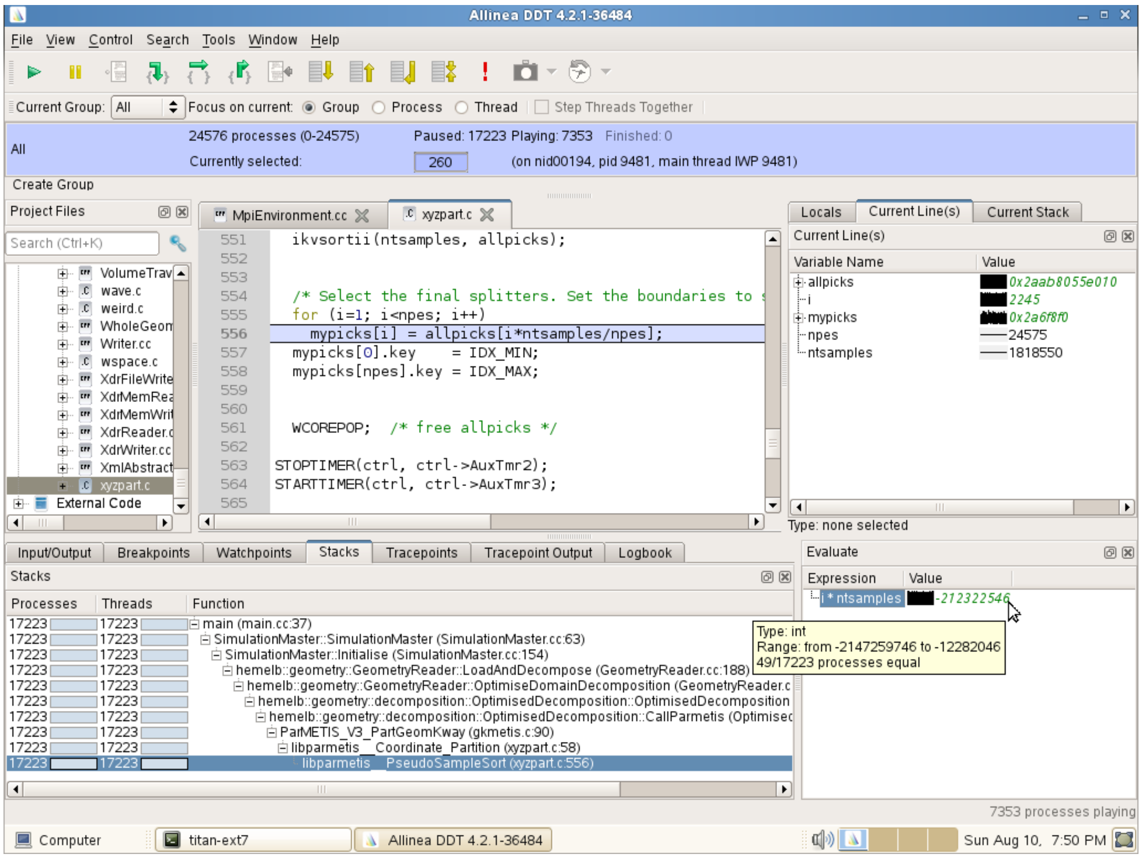
### 6.2.2 Solving a 50,000 Cores Crash

After having made performance improvements, the developers then attempted to take HemeLB through to high-scale – running at 3k, 6k, 12k, 24k cores successfully. The next attempt, to run the application 49,152 cores crashed repeatedly – and the application was unable to progress the science as a result.

At this point, Allinea and UCL explored the issue together. It required access to debugging that could handle this scale – Allinea DDT was installed and ready to use on the system.

We were subsequently able to reproduce the issue on a 24,576 core run which we use here for illustration of the method, only.

Initially it was not clear how far through the application the crash was, but it was believed to be an error within the 3rd party ParMETIS partitioner. We ran the application through with DDT once, to confirm this scenario, and then recompiled this core library to enable debug information to be provided by the compiler.



All    24576 processes (0-24575)    Paused: 17223 Playing: 7353 Finished: 0
Currently selected: 260 (on nid00194, pid 9481, main thread IWP 9481)

```
551    ikvsortii(ntsamples, allpicks);
552
553
554    /* Select the final splitters. Set the boundaries to s
555    for (i=1; i<npes; i++)
556      mypicks[i] = allpicks[i*ntsamples/npes];
557    mypicks[0].key    = IDX_MIN;
558    mypicks[npes].key = IDX_MAX;
559
560
561    WCOREPOP;  /* free allpicks */
562
563  STOPTIMER(ctrl, ctrl->AuxTmr2);
564  STARTTIMER(ctrl, ctrl->AuxTmr3);
565
```

Current Line(s)

| Variable Name | Value |
|---|---|
| allpicks | 0x2aab8055e010 |
| i | 2245 |
| mypicks | 0x2a6f8f0 |
| npes | 24575 |
| ntsamples | 1818550 |

Type: none selected

Stacks

```
Processes  Threads  Function
17223      17223    main (main.cc:37)
17223      17223     SimulationMaster::SimulationMaster (SimulationMaster.cc:63)
17223      17223      SimulationMaster::Initialise (SimulationMaster.cc:154)
17223      17223       hemelb::geometry::GeometryReader::LoadAndDecompose (GeometryReader.cc:188)
17223      17223        hemelb::geometry::GeometryReader::OptimiseDomainDecomposition (GeometryReader.c
17223      17223         hemelb::geometry::decomposition::OptimisedDecomposition::OptimisedDecomposition
17223      17223          hemelb::geometry::decomposition::OptimisedDecomposition::CallParmetis (Optimised
17223      17223           ParMETIS_V3_PartGeomKway (gkmetis.c:90)
17223      17223            libparmetis__Coordinate_Partition (xyzpart.c:58)
17223      17223             libparmetis__PseudoSampleSort (xyzpart.c:556)
```

Evaluate

| Expression | Value |
|---|---|
| i * ntsamples | -212322546 |

Type: int
Range: from -2147259746 to -12282046
49/17223 processes equal

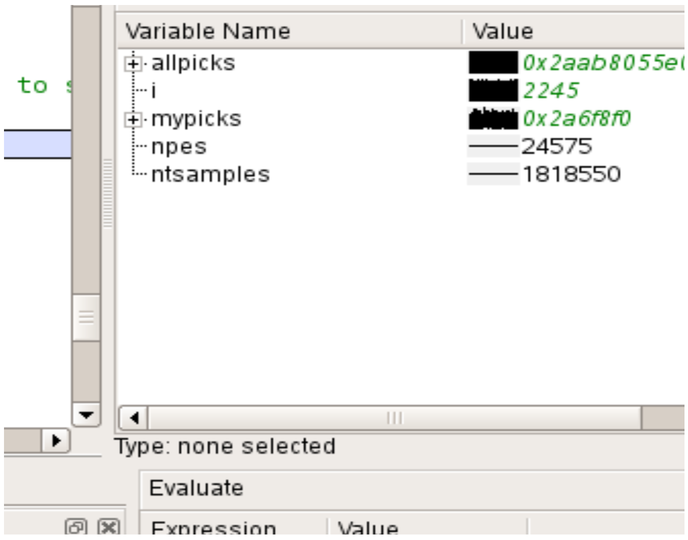
7353 processes playing

**Figure 35: Allinea DDT window showing 80% pf processes crashing at the same ParMETIS line**

This initial view identified almost 80% of the processes as crashing at the same identical line of ParMETIS (illustrated above with the single blue line through the source code). In Figure 36, 17,223 processes have crashed at xyzpart.c line 556. This should be a “well proven” line of code in this well used library.

This particular line of code suggested only a few potential options – invalid arrays, invalid array access, or general data corruption as a side effect of earlier errors.

Examining the array indexing into allpicks on the right hand side of the expression. We could see that the data was sensible and consistent (given by the straight horizontal line graph of ntsamples and npes):

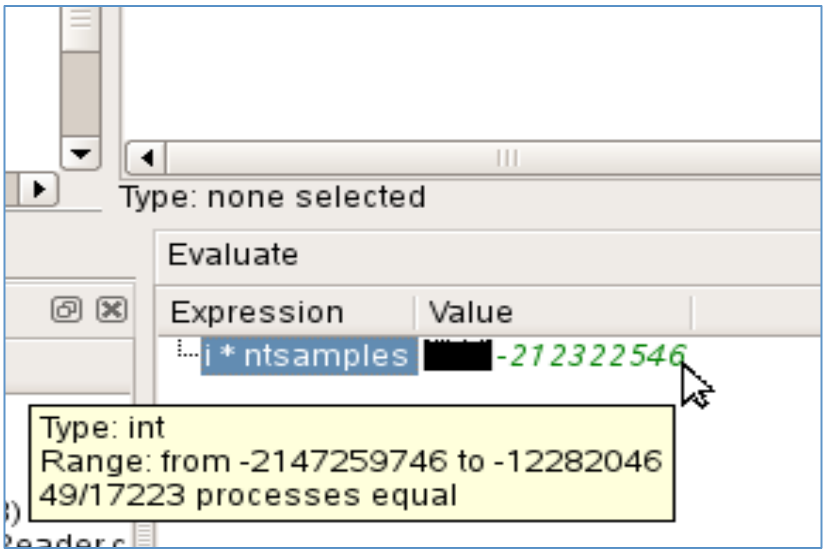


| Variable Name | Value |
|---|---|
| allpicks | 0x2aab8055e( |
| i | 2245 |
| mypicks | 0x2a6f8f0 |
| npes | 24575 |
| ntsamples | 1818550 |

Type: none selected

**Figure 36: Allinea DDT window showing that  the data was sensible and consistent**

We then examined the expression used in the indexing and evaluated this within the debugger:



```
i * ntsamples   -212322546
Type: int
Range: from -2147259746 to -12282046
49/17223 processes equal
```

Figure 37: Allinea DDT Window showing that indexing is overflowing

As could be clearly seen: the indexing was overflowing.

We were able to identify that indexes were by default 32-bit – which (given 15 bits are used by the processor count alone) is not sufficient for such common indexing / sampling multiplications in Petascale applications.

Having exactly identified the cause, the application was recompiled to use 64 bit indexing (a difficult to find configuration option to ParMETIS) – and then successfully completed the largest simulation ever achieved.

This problem was solved extremely quickly – and contrasts with what was expected to be a near impossible task. Interactive, debugging was as effective and fast as debugging only a handful of processes.

The bug and fix have been fed back to the ParMETIS team.

A case study on this result has been published on the CRESTA website, and widely coverage by the industry press.

## 6.3 MPI Correctness Check of HemeLB With MUST

Discussions with the developers of the CRESTA co-design applications highlighted HemeLB as an interesting and challenging test case for MUST. The application makes heavy use of MPI derived datatypes. This includes a continuous creation and destruction of datatypes at runtime, whereas most applications create their datatypes once in an initialization function. This behavior distinguishes HemeLB from existing test cases for MUST. Particularly, the use of the *struct* datatype that HemeLB is employing is known to decrease type matching performance in MUST, and the continuous creation of datatypes stresses a component of MUST that received no scalability services yet. As a result, our experiments serve to:

- Check whether MUST's checks operate correctly,
- Check whether the adaptive communication in HemeLB exhibits no hidden MPI usage errors, and
- Analyze scalability of MUST.

Initial runs with MUST yielded two deviations from expected behavior: First, the correctness logs of MUST could become lengthy. Second, MUST reported suspicious type matching errors or deadlocks. An investigation of the log file revealed a defect in the filtering and aggregation rules that condense correctness reports of multiple processes. We applied a correction to MUST to remove this defect. Afterwards, we could trace the suspicious correctness errors to a second defect in MUST's datatype handling. This defect resulted from an incorrect handling of reused identifiers. A second correction then provided correct behavior in MUST.

The remaining items in MUST's correctness log highlight the use of MPI_Waitall directives with a count value of 0. This is correct application behavior, but suspicious, which motivates us to highlight it as a warning with MUST. Additionally, we observe a missing MPI_Type_free directive for a datatype created with MPI_Type_create_resized. We still investigate the source of this report and whether it highlights a small improvement option for HemeLB's cleanup, or whether is a spurious message.

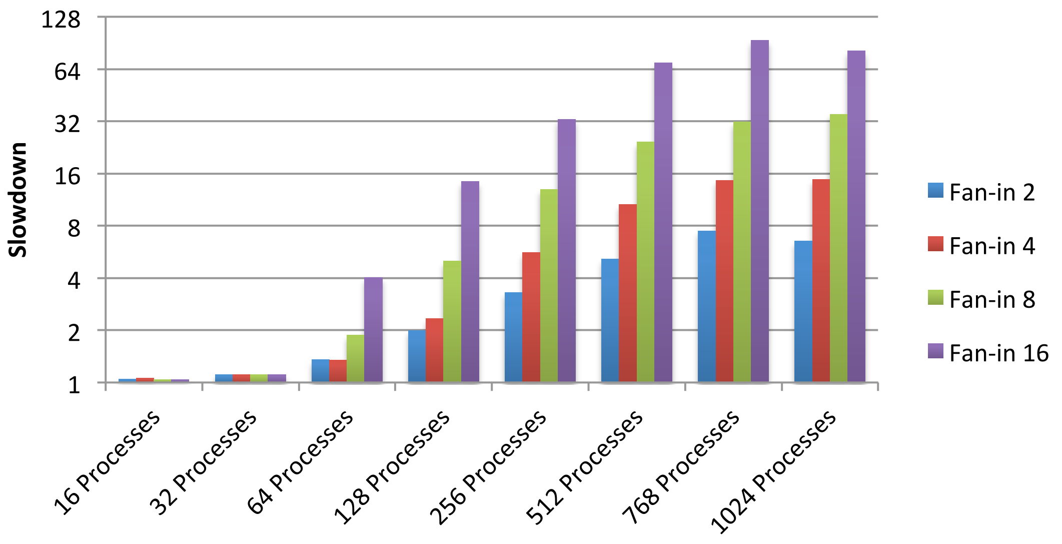


**Figure 38: Runtime with MUST divided by runtime of a reference run as "Slowdown" highlights the impact of the fan-in.**

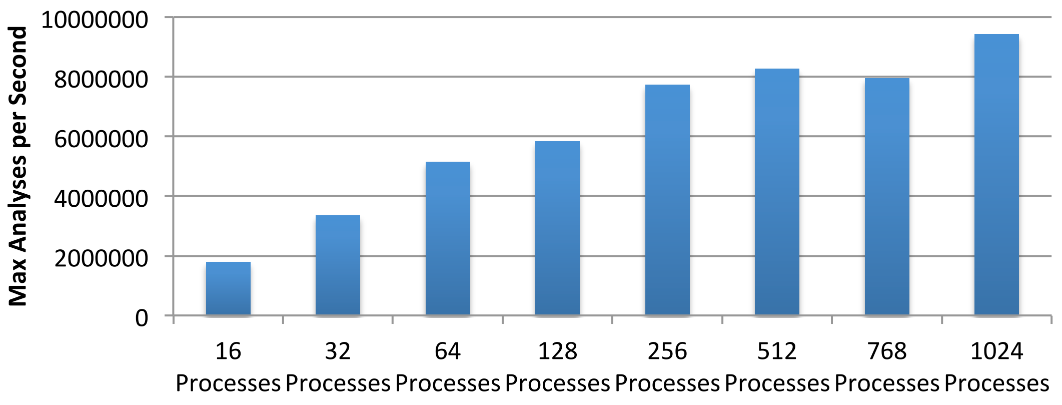


**Figure 39: Increasing event rates (from maximum number of analyses on a MUST tool process) highlight the cause for increasing slowdowns.**

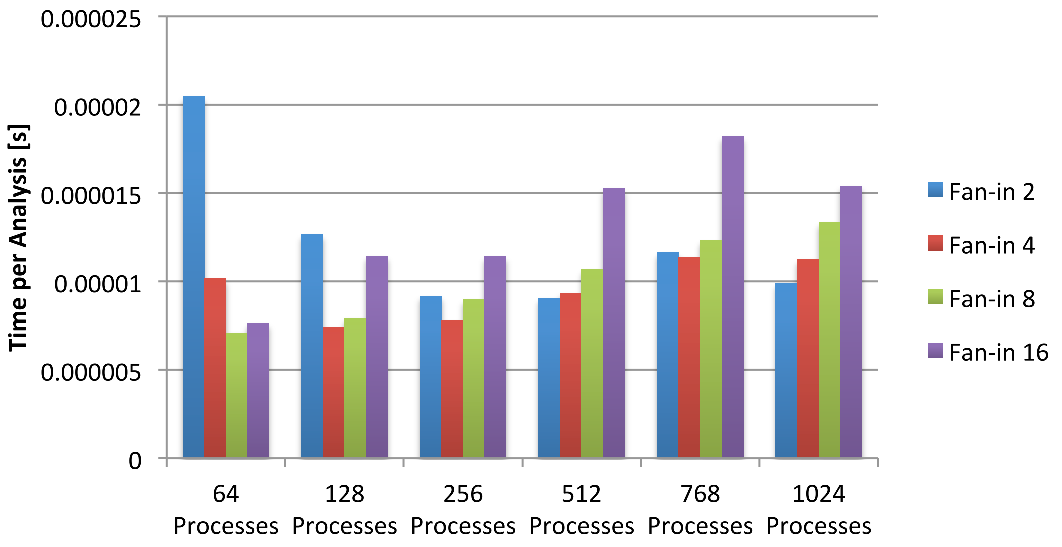


**Figure 40: Time (in seconds) per event analysis highlights no scalability problems.**

To investigate the overhead of MUST for this challenging application, we use a simple bifurcation dataset that is known to scale well to 768 MPI processes. We run our tests on the Sierra Cluster at the Lawrence Livermore National Laboratory. With a reduced feature set of MUST that focuses on point-to-point analysis and its associated type matching checks we run experiments at increased scales. This feature set serves the primary operation type in HemeLB well. Figure 41 presents MUST overheads for increasing scale for this test case. We use different fan-ins for the MUST configuration to control how many tool processes of MUST serve one application process. The lower the fan-in, the lower the tool overhead (at the expense of extra compute resources). The measurements highlight that even with the increased datatype matching costs for the complex derived datatypes we can handle HemeLB at increased scale. Note that the experiment setup is a strong scaling test, i.e., the rate at which MPI processes of HemeLB issue MPI operations increases with scale (Figure 40). Thus, MUST overhead increases with scale, since tool processes handle increasing loads. The slowdown for 768 and 1024 processes already highlights that the slowdown becomes about constant when the application reaches its saturation point, i.e., when it cannot increase its MPI operation rate anymore. Figure 41 highlights this notion by computing the runtime that MUST consumes per event analysis. This value is about constant across scale and highlights no linear increase that would denote a scalability problem (at lower scale it is higher due to idle times on the tool processes).

## 6.4 Outstanding Issues and Future Work

Development of Allinea tools continues as commercially supported products – and the roadmap has taken on board feedback from experience within CRESTA. The R&D team continues to work on next generation architectures and programming models. Opportunities for R&D that were not ready to be addressed during CRESTA such as fault-tolerant tools and checkpoint ready tools, or domain specific language support are still in early days, which support our decision to not include these at the time, but may see new R&D projects in the near future.

With the test cases of MUST, its benchmark experiments, and our co-design experience, we see good scalability for MUST. We detail benchmark experience in D3.7 “Frameworks for Exascale Applications” separately. The behavior of HemeLB to continuously create derived MPI datatypes highlights one option for improvement in MUST. To better support this scenario MUST could provide scalability services for its management of user defined MPI resources (communicators, process groups, requests, windows, datatypes). Another notion is that MUST can exhibit noticeable slowdowns even with low fan-ins, e.g., about 8 for the experiments with HemeLB and a fan-in of 2. Depending on the application use case, especially for long running applications, this can decrease the applicability of MUST. Performance improvements for the event handling and processing in MUST and GTI could further decrease slowdowns.

# 7 Conclusions

In this deliverable, we described the experiences gained with applying the methods and tools developed in WP3 to benchmarks and co-design CRESTA applications. We presented the experience with benchmarks and application for each WP3 task. For each framework developed in WP3, a critical review of outstanding issues was performed and future research directions were outlined.

We presented first the experiences gained with the PGAS programming model by developing a Coarray Fortran benchmark suite, using the Coarray Fortran in the IFS application to calculate Legendre Transforms and implementing Fast Fourier Transforms in UPC. We reported the first results using the targetDP programming framework in Ludwig, a lattice Boltzmann application.

We investigated the use of compiler support for GPU programming by porting the Nek5000 code to multi-GPU systems and present the performance results. We described the use of OpenACC in the GROMACS application. We used the first implementation of an auto-tuning system for OpenACC codes to tune the OpenACC version of the Nek5000 code. The co-design work, involving the development of the adaptive runtime system and Nek5000, was described, and the use of different components of the runtime systems in benchmarks was presented.

The new features of Score-P and Vampir (support for new programming systems and new hardware counters, selective monitoring and enhanced scalability) have been used in CRESTA applications: Nek5000, OpenFOAM, IFS, HemeLB, Gromacs.

The Allinea DDT and MAP tools and MUST correctness checker have been used in HemeLB CRESTA application to detect and analyze software errors and correctness on large scale HemeLB simulations.

## 8 References

[1] Frameworks for Exascale Applications, CRESTA Project Deliverable D3.7

[2] T. El-Ghazawi, W. Carlson, and J. Draper, *UPC Manual v1.2*, June, 2005, https://upc-lang.org/upc-documentation

[3] J. Reid, *Coarrays in the next Fortran Standard*, April 21, 2010. ftp.nag.co.uk/sc22wg5/N1801-N1850/N1824.pdf

[4] D. Henty, *A Parallel Benchmark Suite for Fortran Coarrays*, in Applications, Tools and Techniques on the Road to Exascale Computing (IOS Press, 2012), pp. 281-288.

[5] D. Henty, *EPCC Fortran Coarray micro-benchmark suite* (v1.0 at 01/11/2014), https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-co-array-fortran-micro

[6] D. Henty, *Performance of Fortran Coarrays on the Cray XE6*, in Proceedings of Cray User Group 2012, cug.org/proceedings/attendee_program_cug2012/includes/files/pap181.pdf

[7] D. Henty, *Fortran Coarrays: PGAS Performance on Cray XE6 and Cray XC30 Platforms*, hot topic talk presented at EASC 2014, the Second Exascale Applications and Software Conference, 2-3 April 2014 Stockholm, Sweden.

[8] J. Reid, *Additional coarray features in Fortran*, in Proceedings of the 7th International Conference on PGAS Programming Models 3-4 Oct 2013, Edinburgh, www.pgas2013.org.uk/sites/default/files/pgas2013proceedings.pdf

[9] Mozdzynski, George, et al. "A PGAS implementation by co-design of the ECMWF Integrated Forecasting System (IFS)." *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012.

[10] Järleberg, Erik, et al. "Communication and Computation Overlapping Parallel Fast Fourier Transform in Cray UPC." *7th International Conference on PGAS Programming Models*.

[11] Fürlinger, Karl, and David Skinner. "Capturing and visualizing event flow graphs of MPI applications." *Euro-Par 2009–Parallel Processing Workshops*. Springer Berlin Heidelberg, 2010.

[12] Aguilar, Xavier, Karl Fürlinger, and Erwin Laure. "MPI Trace Compression Using Event Flow Graphs." *Euro-Par 2014 Parallel Processing*. Springer International Publishing, 2014. 1-12.

[13] NERSC-8/Trinity Benchmarks. http://www.nersc.gov/systems/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/

[14] Alcouffe, Ray E., et al. "PARTISN: A time-dependent, parallel neutral particle transport code system." *Los Alamos National Laboratory, LA-UR-05-3925 (May 2005)* (2005).

[15] MPICH wiki : http://wiki.mpich.org/mpich/images/1/17/Wave2d.cpp.txt.

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

[17] Tufo, H.M., Fischer, P.F.: Fast Parallel Direct Solvers For Coarse Grid Problems. In: J. Par. & Dist. Comput., 61, p. 151--177 (2001).

[18] Fox, G.C. et.al.: Solving Problems on Concurrent Processors: General techniques and regular problems. Prentice Hall, Englewood Cliffs NJ (1988).

[19] Michael Schliephake and Erwin Laure. Performance Analysis of Irregular Collective Communication with the Crystal Router Algorithm. *EASC 2014 Exascale Applications and Software Conference.* Stockholm, 2-3 April 2014.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

# Annex A. GROMACS/OpenACC Porting

This Annex contains a report on the GROMACS/OpenACC porting work prepared by the Cray Programming Environment compiler development team.

## A.1 Summary of Cray GROMACS OpenACC Work

During the summer of 2013, Cray, Inc. employed a summer intern to attempt a port of GROMACS 4.6.2-dev to use OpenACC in place of CUDA. This effort continued intermittently into the early part of 2014.

Our intern worked the problem from two angles. He attempted to write OpenACC from scratch starting with the CPU version while at the same time taking the low-level CUDA version and replacing specific CUDA functions with equivalent OpenACC. Instead of compiling the function with the Nvidia nvcc CUDA compiler, it was compiled with the Cray compiler (CCE). We had a "working" version of the latter method first and dropped the rewrite idea. Replacing specific CUDA with OpenACC had the advantage of reusing the CUDA optimizations.

We got the OpenACC version to the point where performance was within 10-15% of the CUDA version but the more we worked on it, the more we considered it a prototype rather than implementation. The CUDA version through macros and C++ methods, has 24 versions of the compute intensive kernel called "nbnxn". At runtime, depending on the input data file, various versions of the kernel are called, some of which short-circuit large amounts of work. Our OpenACC kernel was only one of the 24-kernels in the CUDA version, therefore not close to an actual working version of Gromacs. In addition, it was difficult to verify if accuracy of outputs were sufficient and some were clearly wrong. For debugging, we used only one input file: rf.tpr.

The prototype can be used as an example of how to replace CUDA code with OpenACC code, but replacing the entire CUDA version will require significant rewrite. An OpenACC version has the advantage of being more independent of specific GPU architectures, requiring less maintenance than the current low-level CUDA version. CCE did add significant capability specifically for Gromacs in 8.3 such as:

- compile-time setting of max regs
- access to CUDA functions
- access to fast, low precision rsqrt instructions
- access to atomics
- kernel launch taking into account use or non-use of shared-memory setting optimal cache-configuration.

The best performing versions of CUDA and OpenACC made use of OpenMP threading in the CPU portion of the Gromacs.

## A.2 Details

The compute intensive part of GROMACS kernel nbnxn contains 24 CUDA versions, and one is chosen at runtime by this call:

```
nb_kernel = select_nbnxn_kernel(cu_nb->kernel_ver, nbp->eeltype, bCalcEner,
                                 plist->bDoPrune || always_prune);
```

which choses the specific kernel from a 6x2x2 dimensioned decision table. The first dimension has 6 entries (0=k_nbnxn_cutoff, 1=k_nbnxn_rf, 2=k_nbnxn_ewald_tab, 3=k_nbnxn_ewald_tab_twin, 4=k_nbnxn_ewald, 5=k_nbnxn_ewald_tab_twin). The second dimension indicates whether energy is computed (0=energy not computed, 1=energy computed), and the third dimension indicates whether pruning is used (0=no pruning, 1=pruning).

The actual CUDA kernels executed at runtime are driven by the input data set. The energy version of each kernel appears to be invoked every 100-iterations (beginning with 1).

The kernel decision table is below:

```
nb_default_kfunc_ptr[eelCuNR][nEnergyKernelTypes][nPruneKernelTypes] =
{
    { { k_nbnxn_cutoff,                       k_nbnxn_cutoff_prune },
      { k_nbnxn_cutoff_ener,                  k_nbnxn_cutoff_ener_prune } },
    { { k_nbnxn_rf,                           k_nbnxn_rf_prune },
      { k_nbnxn_rf_ener,                      k_nbnxn_rf_ener_prune } },
    { { k_nbnxn_ewald_tab,                    k_nbnxn_ewald_tab_prune },
      { k_nbnxn_ewald_tab_ener,               k_nbnxn_ewald_tab_ener_prune } },
    { { k_nbnxn_ewald_tab_twin,               k_nbnxn_ewald_tab_twin_prune },
      {  k_nbnxn_ewald_tab_twin_ener,           k_nbnxn_ewald_twin_ener_prune  }
},
    { { k_nbnxn_ewald,                        k_nbnxn_ewald_prune },
      { k_nbnxn_ewald_ener,                   k_nbnxn_ewald_ener_prune } },
    { { k_nbnxn_ewald_twin,                   k_nbnxn_ewald_twin_prune },
      {  k_nbnxn_ewald_twin_ener,               k_nbnxn_ewald_twin_ener_prune  }
},
};
```

The actual call to the CUDA kernel is:

```
nb_kernel<<<dim_grid,  dim_block,  shmem,  stream>>>(*adat,  *nbp,  *plist,
bCalcFshift);
```

where "nb_kernel" is dynamically set to one of the 24 actual CUDA kernels.

The OpenACC version call is:

```
acc_test(aatomdata,anbparam,aplist,acc_bcalcfshift,nblock,CL_SIZE*CL_SIZE,shm
em,plist->bDoPrune||always_prune,bCalcEner,d_tmp,bcalc,tshift);
```

where "acc_test" is static. All data movement in the OpenACC version is still done with CUDA though this could easily be converted to use OpenACC.

For input data file rf.tpr, 2000-iterations of "rf" are called with the energy version called every 100 trips (beginning with 1).

The developed OpenACC kernel used the CUDA kernel "rf" as a model because input data file rf.tpr was used. However, the "energy" version of "rf" was not implemented, and therefore "energy" results are not complete.

It was felt much optimization work had already been done so there was advantage to reusing this optimization work. The alternative would have been to begin with the CPU version and add OpenACC directives.

Because there is only one of 24 CUDA kernels developed using OpenACC, the Cray effort is not a complete solution, but could be used as a model going forward. Source changes would be required to mimic the CUDA kernel selection that chooses the specific kernel. Alternatively, the kernel checks could be done at runtime though with probably performance cost. Because CUDA is low level, maintenance costs may be cheaper transitioning between GPU generations with OpenACC compared to CUDA. Other functions currently done in CUDA, such as explicit data movement between CPU and GPU, could be transitioned to OpenACC.

## A.3 Modified Files

The following files were modified for this OpenACC experiment (all relative to the base directory):

- OpenACC call to synchronize CUDA and OpenACC streams. Note this file has file extension of "cu" and therefore is processed by nvcc, not a C compiler such as craycc (part of CCE). Because of this, the usual OpenACC interface file (openacc.h), could not be referenced but instead a direct file path to openacc.h was used as a workaround:
  `src/mdlib/nbnxn_cuda/nbnxn_cuda_data_mgmt.cu`
- Device is initialized with OpenACC:
  `src/kernel/mdrun.c`
- Invokes the OpenACC kernel multiple times:
  `src/mdlib/nbnxn_cuda/nbnxn_cuda.cu`
- The actual OpenACC kernel code:
  `src/mdlib/nbnxn_kernels/nbnxn_kernel_gpu_ref.c`
- OpenACC types added:
  `src/mdlib/nbnxn_kernels/nbnxn_kernel_gpu_ref.h`

## A.4 OpenACC Performance Issues

When functional, performance was poor relative to CUDA. Analysis showed performance suffered relative to the CUDA version for the following reasons:

- shared memory was not used to contain arrays defined and referenced by different threads within a block, causing long latency global memory references
- poor occupancy caused by more than 128-registers per thread
- register spilling for same reason
- use of double precision reciprocal sqrt functions along with casts between single and double.
- poor occupancy caused by amount of shared-memory usage (after eventual usage of shared-memory)
- The CUDA version uses a block of 8x8x1 whereas OpenACC has no official way to indicate block geometry. The OpenACC code converts a 64x1x1 block shape to 8x8x1 through extra code.

## A.5 Optimizations

The following optimizations were made:

- To get similar arithmetic performance as the nvcc option "-use_fast_math", CCE compiler option -hfp4 is used allowing rsqrt instruction instead of the much slower discrete sqrt and divide operations. Option -hfp4 also allows for general use of rounded versions of arithmetic instructions equivalent to nvcc options:
  `-ftz=true -prec-div=false -prec-sqrt=false`
- Use of the fast, low precision rsqrt instruction requires defensive coding style to handle denormals explained in: https://developer.nvidia.com/content/cuda-pro-tip-flush-denormals-confidence
- CCE compiler option `-Wx,"--maxrregcount=64"` (new with CCE 8.3) should be used to set GPU kernel max registers to 64 which increases occupancy to 0.5 (up from 0.25) with CCE default 128 max registers. This new 8.3 CCE capability allows options to be passed to ptxas from the CCE invocation.
- Similar to the CUDA version, shared memory is used to contain arrays defined and referenced by different threads within a block.
- The CUDA version uses a block of 8x8x1 whereas OpenACC has no official way to indicate block geometry. The OpenACC code converts a 64x1x1 block shape to 8x8x1 through extra code.

- At kernel launch, the OpenACC runtime uses amount of kernel shared-memory to choose the optimal L1/shared-memory ratio configuration among:
  1. 16KB L1 and 48KB shared-memory
  2. 48KB L1 and 16KB shared-memory
  3. 32KB L1 and 32KB shared-memory
- The single large CUDA kernel was split into 3 OpenACC kernels lessening the max register count for the remaining dominate kernel.
- Use of OpenMP threading in addition to GPU threading improved performance. Wall-clock speedup was approximately 2x when using 8 OpenMP threads as measured on both Cray XK7 and Cray XC30 systems.

## A.6 Next Steps

The routine containing the OpenACC kernel "acc_test" in file:

`src/mdlib/nbnxn_kernels/nbnxn_kernel_gpu_ref.c`

requires integration into the Gromacs framework.

Further OpenACC work is required to cover the 24 possible CUDA kernels including the 6 eeltype's, with and without and pruning.