D3.1 – State of the art and gap analysis

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 Exascale computing, software and simulation

Due date:	M5
Submission date:	29/02/2012
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	KTH
Version:	1.0
Status	Final
Author(s):	Xavier Aguilar (KTH), Jens Doleschal (TUD), Alan Gray (UEDIN), Alistair Hart (CRAY UK), David Henty (UEDIN), Tobias Hilbrich (TUD), David Lecomber (ASL), Stefano Markidis (KTH), Harvey Richardson (CRAY UK), Michael Schliephake (KTH)
Reviewer(s)	James Hetherington (UCL), Jeremy Nowell (UEDIN)

Dissemination level	
<PU/PP/RE/CO>	*PU – Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	24/01/2012	First version of the deliverable	Xavier Aguilar (KTH), Jens Doleschal (TUD), Alan Gray (UEDIN), Alistair Hart (CRAY UK), David Henty (UEDIN), Tobias Hilbrich (TUD), David Lecomber (ASL), Stefano Markidis (KTH), Harvey Richardson (CRAY UK), Michael Schliephake (KTH)
0.2	30/01/2012	- First draft of executive summary, introduction, and conclusion sections. - Updated chapter 3. Added new sections and subsections. Added references. - Updated section 4.3. Included the references. - Reorganised chapter 5. Completed subsection 5.2.5. Included additional references. - Reorganised chapter 6. - Fixed reference style.	Xavier Aguilar (KTH), Jens Doleschal (TUD), Alan Gray (UEDIN), Alistair Hart (CRAY UK), David Henty (UEDIN), Tobias Hilbrich (TUD), David Lecomber (ASL), Stefano Markidis (KTH), Harvey Richardson (CRAY UK), Michael Schliephake (KTH)
1.0	10/2/2012	- Corrections made to document based on author feedback.	Xavi Aguilar (KTH), Jens Doleschal (TUD), Alan Gray (UEDIN), Alistair Hart (CRAY UK), David Henty (UEDIN), Tobias Hilbrich (TUD), David Lecomber (ASL), Stefano Markidis (KTH), Harvey Richardson (CRAY UK), Michael Schliephake (KTH)

2.0		- Corrections made to document based on reviewer feedback.	Xavi Aguilar (KTH), Jens Doleschal (TUD), Alan Gray (UEDIN), Alistair Hart (CRAY UK), David Henty (UEDIN), Tobias Hilbrich (TUD), David Lecomber (ASL), Stefano Markidis (KTH), Harvey Richardson (CRAY UK), Michael Schliephake (KTH)
3.0		- Final corrections made to document based on author and reviewer feedback.	Xavi Aguilar (KTH), Jens Doleschal (TUD), Alan Gray (UEDIN), Alistair Hart (CRAY UK), David Henty (UEDIN), Tobias Hilbrich (TUD), David Lecomber (ASL), Stefano Markidis (KTH), Harvey Richardson (CRAY UK), Michael Schliephake (KTH)

Table of Contents

Index of Figures

1 Executive Summary

Development environments provide the tools to ease the implementation of scientific algorithms in computer codes, enable applications to run efficiently on parallel supercomputers, allow fast and non-invasive performance monitoring and analysis, and permit prompt detection of code errors. This document presents the state of the art for scientific programming development environments, discusses requirements for exascale computing, and outlines the future work of the CRESTA development environment work-package to enable the CRESTA co-design applications to achieve exascale performance.

Traditional programming languages (Fortran, C, C++) and parallelisation methods are reviewed. A PRACE survey among 55 applications shows that majority of applications running on European supercomputers are written in Fortran, C and C++. Fortran (Fortran 90/95, Fortran 77) is the most popular. MPI is the most common method to achieve parallelism, while a few applications use a hybrid solution with MPI and OpenMP. Emerging programming models, such as the PGAS languages/libraries (UPC, Co-array Fortran, Chapel, OpenSHMEM), are introduced also. It is very likely that application developers will prefer an incremental approach to a full rewrite in a new language. The programming of GPUs with CUDA, OpenCL, and compiler directives, such as OpenACC, is described. Accelerator directives provide a mechanism for bridging the gap between applications written for the CPU and those ported explicitly to the GPU. The main challenge still remains to deliver an acceptable level of performance using this high-level approach. The current OpenACC directives are unlikely to provide the full functionality to exploit a heterogeneous node. An important task during the CRESTA project is to ensure that the nascent standards, e.g. as part of OpenMP, evolve in the most productive way for HPC users of heterogeneous architectures.

The state of the art of compilers, automatic tuning tools, and runtime systems is presented. New compiler optimisation technique, targeting exascale computers, are discussed, and different autotuning frameworks, such as Active Harmony and CHiLL, and runtime systems, such as StarPU, StarSS, ForestGOMP, Charm++, and HPX, are discussed. It is shown in this deliverable that there has been very promising work in the area of compilers, autotuning and runtime systems. The challenge for the CRESTA project is to extend these techniques to large-scale distributed memory applications such as those in the CRESTA benchmark suite. This study needs to be started on the current generation of petascale machines to identify the most successful approaches that may prove to work on exascale computer platforms. With current automatic tuning tools and runtime systems, there is no single approach that meets all the different challenges. We propose to define a consistent approach that can target mark-up of choices at all the development stages (algorithm choice, source, compilation and launch). This will be done by developing a domain-specific language (DSL) that enables the expression of knowledge, hints and decisions for an efficient execution of an application on exascale supercomputers. Its development will be informed by both the CRESTA co-design applications and open source autotuning projects. If the DSL we develop is general enough then it can be used to wrap specific autotuners without excessive effort. Other runtime services are directed towards the support of dynamic load balancing of application runs on exascale systems as well as using hybrid parallelisation techniques for dynamic adaption of the program to heterogeneous systems.

Existing techniques to monitor and analyse application performance on current petascale computer systems are presented. The Vampir and Score-P performance monitoring and analysis tools have been proven to scale on current petascale supercomputers. However, to raise performance monitoring and analysis tools from petaflop to exaflop scale requires us to develop new methods for monitoring and analysis of information or to combine existing methods to gain a better insight into the system and application by the lowest possible intrusion. To reach exascale

performance, the goal is to develop scalable strategies to selectively monitor systems and applications and to use analysis techniques to identify outliers and provide sufficient insights into application and system behaviour. Data mining and reduction techniques will be necessary in exascale computing in order to perform on-the-fly information reduction that will be a requirement for a scalable, automated online performance analysis.

The current state of parallel debuggers and correctness checking tools is presented. The Allinea DDT debugger proved to be scalable on the current petascale computing systems. A survey among CRESTA application experts was completed to understand the current usage of debuggers and error checking tools. This includes information on common implementation errors that application developers make and desirable new features in debugger environments. The survey points out that scalability of debuggers on exascale supercomputers is an important priority for application developers. However, methods to present program state of an exascale application such that developers can understand and pinpoint bugs will also be crucial. Additional main points are the debugger support for new programming models, such as PGAS languages, and the integration of debuggers and correctness checking tools in a unique framework. TU Dresden will work with Allinea to enable the MUST MPI checker to work within the Allinea DDT debugger platform, while also extending MUST's scalability in order to cope with more than just smaller scale test cases.

2 Introduction

As we look forward to the exascale era it is clear that we must face new challenges, not least for the software stack required to support exascale applications. Each component of the development environment (programming models, compilers, runtime systems, performance monitoring and analysis tools and debuggers) needs to fully and efficiently exploit future exascale computing platforms, characterised by two main features:

- Extreme parallelism of the order of hundreds of millions of compute units.
- Heterogeneous computer architecture, where the compute units are a mix of CPUs and accelerators.

Programming languages, parallelisation methods, compilers, autotuning software, performance monitoring and analysis tools, and debuggers should fully exploit concurrency available in future exascale supercomputers. All these tools are expected to run efficiently, and scale reasonably while increasing the number of compute units. In addition, the development tools should support heterogeneous architectures.

The challenges of developing software for future exascale computers have been analysed previously by the IESP, ESSI and PRACE initiatives. This deliverable starts from the results presented in their roadmap, white papers and reports, and specifically focuses on the development environment for the CRESTA co-vehicle applications.

This document is organised as follows. Section 3 examines the traditional programming models and parallelisation methods, presenting their current usage in PRACE network and CRESTA co-design vehicle applications. The different levels of hardware parallelism in the current and future computer architecture are then examined. Novel programming models and the programming of GPGPUs with CUDA, OpenCL and compiler directives are discussed. Section 4 presents the challenges of developing compilers to produce an optimised executable code for exascale supercomputers, addresses automatic tuning tools, and discusses the state of the art of current runtime systems. Section 5 presents the performance monitoring and analysis tools available and the new developments in the field. Section 6 presents the state of the art of debuggers and correctness checking tools, their support for programming models and hardware. The usage of debuggers and correctness checking tools in the CRESTA project is analysed. Section 7 concludes the deliverable, analysing the gaps to be filled towards the implementation of development environments for exascale computing and summarising the main results.

2.1 Purpose

The purposes of this deliverable are as follows:

- Present the current status and emerging trends in programming models and parallelisation methods. Emphasis is given to how programming models need to adapt to massive parallelism on different hardware levels and on heterogeneous platforms.
- Present the state of the art and new developments in compiler optimisation techniques and automatic tuning tools.
- Present the state of the art of the runtime systems.
- Present the state of the art in performance monitoring and analysis tools and the challenges of monitoring and analysing the performance of applications running on exascale computer platforms.
- Discuss the current debugger and correctness checking tool performance on petascale computing platform and indicate the new directions in developing scalable debuggers on exascale supercomputers.
- Present the development environment current limitations and gaps that may affect successful implementation for exascale computing platforms.

- Provide directions and guidance for the future work in the CRESTA development environment work package.

2.2 Glossary of Acronyms

AGAS	Active Global Address Space
API	Application Programming Interface
ARB	Architecture Review Board
BLAS	Basic Linear Algebra Subprograms
CAF	Co-Array Fortran
CUDA	Compute Unified Device Architecture
DRAM	Dynamic Random-Access Memory
DSL	Domain Specific-Language
DSM	Distributed Shared Memory
DSP	Digital Signal Processor
EC	European Commission
EESI	European Exascale Software Initiative
FFT	Fast Fourier Transform
FIFO	First In, First Out
FLOP	FLoating point OPerations
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
HPC	High Performance Computing
IESP	International Exascale Software Project
MPI	Message Passage Interface
NUMA	Non-Uniform Memory Access
OO	Object Oriented
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
ORNL	Oak Ridge National Laboratory
PGAS	Partitioned Global Address Space
PRACE	PaRtnership for Advanced Computing in Europe
RDMA	Remote Direct Memory Access
SM	Stream Multiprocessors
SPMD	Single Program Multiple Data
UPC	Unified Parallel C
WP	Work Package

3 Programming Models

3.1 Traditional Programming Methods

3.1.1 Basic Languages

Many hundreds of programming languages and (to a lesser extent) communications models have been developed during the last decades, but it is fair to say that High Performance Computing is dominated by an extremely small subset of well-established programming models (as we demonstrate in Section 3.2). In terms of languages, almost all applications seen on typical HPC systems today are written in Fortran, C or C++, sometimes mixing these. Here we review these languages.

3.1.1.1 Fortran

With its development starting in 1953, Fortran was one of the first high-level programming languages and has been in constant use for nearly sixty years.

The language was developed with a view to efficiency and hence the structure of programs is relatively constrained, making it easier for the compiler to produce efficient code. In established compiler suites, there is a long history of performance optimisation for Fortran, which adds to the performance advantages.

There was an early decision (dating back to FORTRAN66) to standardise the language using internationally recognised standards bodies, ANSI (for FORTRANs 66 and 77 and for Fortran 90) and ISO/IEC (for Fortrans 95, 2003 and 2008). This made the language attractive to HPC users looking for portability of applications between compiler and hardware vendors.

Early support for double precision and complex intrinsic numeric data types (added in FORTRAN II in 1958) also cemented the language's appeal for scientific computing. Good support for multidimensional arrays and enhancements like modules enhanced the appeal of Fortran for many scientific applications. The maintenance of backward compatibility between Fortran versions has also facilitated application longevity.

A number of compilers now advertise full compliance with the Fortran 2003 standard. Although none yet have full support for the Fortran 2008 standard, the Cray Compilation Environment (version 8.0, released December 2011), for instance, has partial support including for submodules and co-arrays [1].

The main strength of Fortran is that the language is deliberately restrictive to the programmer, which gives the compiler better scope to either produce highly optimised code or to carry out strict runtime checks of code correctness (e.g. bounds checking).

The main disadvantages of Fortran are that the language is traditionally poor at interacting with the Operating System, does not handle string manipulations well and that it is difficult to manipulate unformatted data files. These problems have been at least partially addressed in Fortran 2003, but the usual solution has been to mix in some C routines. Despite the language changes in Fortran90 onwards, Fortran is often viewed as an old-fashioned programming language and most universities no longer teach it as standard.

3.1.1.2 C

The C programming language was originally developed between 1969 and 1973 to help realise the Unix operating system.

The language was not formally standardised until 1989 (by ANSI, then adopted by ISO in 1990 and known interchangeably as C89 or C90), but prior to this the book by Kernighan and Ritchie [2] widely acted as a *de facto* definition of the language. Since then, a C99 standard has been released. As with Fortran, backward compatibility was largely maintained during these revisions.

Historically, performance of C applications was hampered by the extensive use of pointers to reference data arrays. This makes it difficult for the compiler to guarantee

independence of data operations, which is needed when optimising (and also when parallelising) compiled code. Many compiler vendors have introduced ways to indicate independence (e.g. through pragmas), but this is not guaranteed to be portable. C99 has improved this somewhat, e.g. by introducing the `restrict` qualifier to indicate independence.

C has two particular strengths in HPC. On systems running a Unix or Linux-based operating system, the programmer has good access to the underlying OS, including environment variables and command line arguments. This is particularly useful when profiling a code and understanding the relationship between application performance and the system state (memory usage etc.). The other strength is the ability to handle raw data files produced, for instance, by third party applications on different computing architectures.

C also has good string handling properties. Sociologically, it is often viewed as being a more modern programming language than Fortran, although, given its age, this is perhaps more because C++ and Java are based on C, rather than C itself being modern. Again, C is only taught in a minority of universities now.

In C, the user has far more control over memory management and this can make it harder for the compiler to produce highly optimised code or to verify code correctness. With modern compilers, however, a lot of these problems have been overcome and for many scientific codes the performance difference between Fortran and C is negligible. In addition, modern Fortran versions also now have improved interfacing abilities with the OS.

3.1.1.3 C++

C++ was developed between 1979 and 1983 by Bjarne Stroustrup. Starting as an extension to C, the language was not standardised until 1998 (by ISO), with a technical corrigendum in 2003 and a new standard known as C++11 in 2011. C++ has only recently been used in HPC, largely because the complexity of the language and the shortage of highly optimising compilers.

Fortran and C are largely procedural languages, with the program organised as a set of distinct subprograms that are explicitly called in the code. This leads to a calltree-based structure for an application profile.

C++ can be used in this way (being a superset of C), but it also offers great scope for the programmer to use a variety of software engineering methods to produce well-organised, understandable, maintainable and extensible code. These methods include: object oriented (OO) programming methods and data-hiding, where data is packaged into objects that also include methods for accessing and modifying the information; template metaprogramming and const correctness.

The classic problem with using C++ in HPC is that the more high-level features are used in the code, the harder it becomes for the compiler to produce optimised code. The encapsulation of methods in objects makes it hard to predict the independence of statements in the code and the packaging of data often leads to poor layouts of data in memory that lead to poor cache utilisation in repeated operations such as loop iterations.

These problems can be reduced by C++ programming techniques such as templating, but the large amount of knowledge needed to start using these is often a barrier to uptake.

The advantage of C++ is that, correctly used, it can abstract the user from the underlying architecture and choice of communications model(s). This must be balanced against the careful programming needed to allow efficient code execution and the possible lack of compiler portability that comes from using such advanced programming techniques. C++ is also seen as being a modern language, with many universities teaching it in the view that this provides a useful employment skill for undergraduates.

3.1.1.4 Fortran/C/C++ interoperability

Given the complementary strengths of Fortran and C, there is a long tradition of mixed language compilation. Different parts of the application may be written in either Fortran or C, as appropriate, and separately compiled to object files that are then linked into a single executable.

In Fortran, data objects may be passed to subprograms simply by reference that matches the mechanism used by C when a pointer is passed. Arrays of data can therefore be presented in the two languages as contiguous chunks of memory. Given that Fortran and C array elements have different orderings in memory (in Fortran, the leftmost index has unit stride in memory, whereas in C it is the rightmost), care must be taken to access data in a consistent manner in the two languages.

Prior to the Fortran 2003 standard, however, it was difficult to mix Fortran and C in a completely portable fashion. Clearly the same levels of precision (i.e. number of bytes per data item) must be used, but this cannot be guaranteed. For instance, whilst it is almost universally true that `INTEGER` in Fortran and `int` in C both use 4 bytes, this is not specified in any language standard. Neither was there any way to check for any incompatibility when cross-calling between languages, either within the program or through compile-time type checking. There is also no guarantee of runtime errors, although most mismatches rapidly lead to floating point exceptions or similarly trappable errors.

Linking of the two languages was also complicated by compiler-dependent additions of trailing underscores to, or even capitalisation of, subprogram names in object files. Whilst some compilers offer options to control this, some investigation usually involving trial-and-error or running the Unix `nm` command (often filtered using `grep`) on the object files was required. This exercise would then be repeated each time the Fortran or C compiler was changed or a new computer architecture was available.

Fortran 2003 represented a milestone in compatibility, providing a standardised mechanism for interoperating with C. This is done through an intrinsic module named `iso_c_binding` that contains information about the type parameter values for intrinsic types. There are also mechanisms for ensuring the compatibility of memory address pointers and of global and passed data. Finally, there is also a consistent way to call Fortran procedures from C and vice versa with Fortran `INTERFACE`s offering compile-time type-checking for passed arguments.

Whilst this was not formally standardised until Fortran 2003, many Fortran95 compilers have supported this interoperability for a number of years. Nonetheless, many applications use the “traditional” non-portable linking, partly due to inertia and partly because of the perceived (but necessary) complexity of the `iso_c_binding` module.

C++ is a much more complicated language than either C or Fortran, so interoperability can be more difficult. If the object oriented programming techniques are sparingly used, C++ is directly compatible with C and, by extension, with Fortran. If, however, the more advanced features of C++ are exploited, interoperability is harder to arrange. Currently, most applications do not attempt to mix C++ with C or Fortran as part of the user code. Libraries written in either C or Fortran are called from within C++ codes, but this is usually achieved by passing pointers to contiguous memory segments that hold the relevant data. As such, this mandates the user to abandon, at least temporarily, the OO principles of data hiding and encapsulation.

With no type-checking of passed data, care must also be taken that the same levels of precision are used in both Fortran and C. Care must be taken to arrange that passed arrays of data are accessed by the two languages in a consistent manner.

3.1.1.5 Python

Python and, to a much lesser extent, other scripting languages are being increasingly used in HPC applications. In contrast to the previous languages, python is not typically used to directly develop key kernels in HPC applications. More usually, it is used either as a wrapper to control job launch parameters or as a runtime framework to tie together

kernels written in compiled languages. The latter is made possible through the integration of, for instance, MPI in python.

The motivation for this is that python has a reputation (largely justified) for providing a way to rapidly and flexibly develop large amounts of code, building on a wide range of existing libraries (known as modules). Given the oft-cited rule of thumb that 80% of the runtime of a typical scientific application is spent in 20% of the code, python is a suitable choice for coding the remaining, non-performance-critical 80% of the application.

Although there is undoubtedly a small performance sacrifice in doing this, the main disadvantage to using python in this way is the current lack of efficient parallel runtime support. As an example, the startup time of the application can become enormous when scaling to large numbers of MPI ranks as each python process separately attempts to load the same module files. These problems are surmountable, but as only a minority of python users (and developers) are using it in an HPC environment, it is not yet clear how this will be achieved.

3.1.2 Parallelisation Methods

Here we introduce the most common methods of achieving parallelism in HPC applications.

3.1.2.1 OpenMP

The OpenMP API [3] is arguably the simplest model for adapting a serial application to allow it to utilise multiple compute cores in parallel, with the key restriction that these cores must operate on the same memory address space: it can be used to parallelise code within multi-core workstations or single nodes of larger system. To be effective for large-scale problems it must be combined with other "distributed-memory" parallelisation techniques.

Before the emergence of the OpenMP standard, there were several hardware vendors offering shared-memory systems but each provided a different compiler with different sets of directives to allow use of multiple threads or vector units, making it hard to write portable code. The OpenMP forum, consisting of major vendors plus academic organisations, released the first OpenMP standard (supporting Fortran only) in October 1997, followed by a revision supporting C/C++ in 1998. Further revisions since then have improved, clarified and expanded the standard.

The model is based on the concept of "threads", which are like processes, except that they can share memory with each other (as well as having private memory). While serial applications only follow a single thread of execution (and hence only utilise a single core), OpenMP allows the use of multiple threads, with the idea that the work in the application can be distributed onto multiple cores. OpenMP provides a set of extensions to Fortran, C and C++, consisting of compiler directives, runtime library routines and environment variables.

The parallel region is the basic parallel construct in OpenMP, which defines the section of the program that is to be executed in parallel. The program begins execution on a single thread (the master thread), and when the first parallel region is encountered, the master thread creates a team of threads. Every thread executes the statements inside the parallel region. At the end of the parallel region, the master thread waits for the other threads to finish, and continues executing the next statements. OpenMP has directives that indicate that work should be divided up between threads. Since loops are the main source of parallelism in many applications, OpenMP has extensive support for parallelising loops. A Fortran example is as follows:

```
!$OMP PARALLEL
!$OMP DO
do i=1,n
  b(i) = (a(i)-a(i-1))*0.5
```

```
end do
!$OMP END DO
!$OMP END PARALLEL
```

Without the directives inserted, the loop would just execute on a single core. The combination of the “parallel” and “do” directives instruct the compiler to split the loop into multiple sub-loops, to be executed by multiple threads in parallel. The number of threads to be used can be controlled through an environment variable or runtime routine, and there are a number of options to control which loop iterations are executed by which threads. It is up to the programmer to ensure that the iterations of a parallel loop are independent. Only loops where the iteration count can be computed before the execution of the loop begins can be parallelised in this way.

Recent versions of OpenMP also support the concept of tasks, which provide a mechanism for parallel execution of code regions where loop-based decompositions are not suitable. The concept of "tasks" was added to the OpenMP standard when version 3.0 was released in 2009. OpenMP tasks allow the parallelisation of certain types of algorithms that are not amenable to straightforward use of loop directives (for example the traversal of linked lists or trees, with operations on each element). The task construct defines a section of code to be packaged up for execution. Multiple tasks can then be executed in parallel by the multiple threads in a parallel region. A new "task wait" directive provides a mechanism to instruct the program to wait until all tasks have been completed.

The concept of threads is wider than OpenMP, and it is possible to incorporate threaded programming into sequential programs using lower level APIs such as POSIX threads (pthreads) (on UNIX platforms) but such APIs are typically involve more complex programming, are not fully cross-platform and cross-language (in particular use within Fortran is not straightforward). OpenMP is a portable higher-level abstraction (which, depending on the implementation, may utilise such lower-level APIs internally).

The OpenMP approach can be attractive as a way to add incremental parallelism to an application without significant changes.

3.1.2.2 Message Passing Interface

It is usually only practical to use OpenMP to parallelise within a single node of a system. To utilise multiple nodes, each with its distinct memory address space, a distributed–memory parallelisation technique is required. Such methods must include mechanisms for data to be transferred between nodes: i.e. for messages to be passed. The effective message-passing standard is the Message Passing Interface (MPI) [4].

In the 1980s and early 1990s there emerged several message-passing systems with differing syntax but similar goals. MPI was the first effort to produce a message-passing interface standard across the whole parallel processing community. Sixty people representing forty different organisations collectively formed the “MPI Forum” in 1992. A two-year process of proposals, meetings and review resulted in a document specifying a standard Message Passing Interface (MPI).

The message-passing model is based on the notion of processes. One can think of a process as an instance of a running program, together with the program’s data. Parallelism is achieved by having many processes co-operate on the same task. Each process has access only to its own data – i.e. all variables are private. Most message passing programs use the Single-Program-Multiple-Data (SPMD) model, where all processes run the same program and each process has a separate copy of the data. To make this useful, each process has a unique identifier and processes can follow different control paths through the program, depending on their identifier (called the “rank”).

Processes communicate with each other by sending and receiving messages through library calls from the conventional sequential language such as Fortran, C or C++. In the simplest case one process would call the `MPI_Send` routine to send data while

another calls `MPI_Recv` to receive data. These calls must correspond, leading to the description of MPI as a "double-sided" communication model. The MPI library functions take as arguments, pointers or references to the data structures as well as other information such as the amount of data and the destination/source rank. There are variants to these library calls to give more control and flexibility to tune performance and manage more complex patterns. There also exists a range of functionality for other operations, such as the initialisation/finalisation of parallel executions, synchronisation, and "collective communications" involving groups of processes (e.g. one process broadcasting data to all others).

MPI is very powerful and flexible: it gives the programmer intricate control to develop a complex parallel program spanning many nodes, and this is why it has become the most popular technique on large-scale supercomputers. This is at the expense of complexity: since it is up to the user to fully manage the distinct memory spaces and task decompositions, MPI programs can become very involved. Also moving an existing application to MPI often requires significant code changes (in comparison to adding OpenMP).

3.1.2.3 Hybrid Programming and Interoperability of OpenMP and MPI

Modern systems are comprised of multiple nodes, each with multiple cores and shared resources such as memory. OpenMP programs are usually restricted to a single such node, while MPI programs can span multiple nodes. The simplest way to utilise all cores using MPI is to run with the same number of MPI processes as cores, i.e. treat each core, and associated memory, as a separate "partition" of each node, and not to differentiate between intra- and inter-node communications at the program level (although a well-written MPI library might well do so behind the scenes). Alternatively, one can run with a "hybrid" program containing a combination of MPI and OpenMP. In this case there would be less MPI processes than physical cores, and each MPI process would consist of multiple threads, to ensure utilisation of all cores. A typical configuration is one MPI rank per node or CPU.

Performance can benefit from this approach in a number of ways [5]. By using fewer MPI ranks per node, the memory requirements of the application can be reduced, allowing larger local sub-problems in the parallel-decomposed problem and making the performance less reliant on the memory bandwidth. The increased local problem size can increase the efficiency of the local computation. It can also allow further strong scaling of applications that had reached the limits of their parallel decomposition. Using fewer MPI ranks per node also reduces network traffic, which may have particular benefits in collective operations (depending on how the MPI library is written).

The advantages are system and implementation dependent, but are expected to become increasingly important as the number of cores per node increases (as is the trend; see CRESTA Deliverable D2.1.1 "Architectural developments towards Exascale" for further discussion).

3.2 Current Programming Model Usage

3.2.1 PRACE Survey

The Partnership for Advanced Computing in Europe (PRACE) project is tasked with implementing a pan-European High Performance Computing service and the necessary infrastructure [6]. A PRACE deliverable was published in February 2011 that contained the results of surveys designed to gather information on the active HPC systems in Europe, usage profiles and details of the applications using these systems [7].

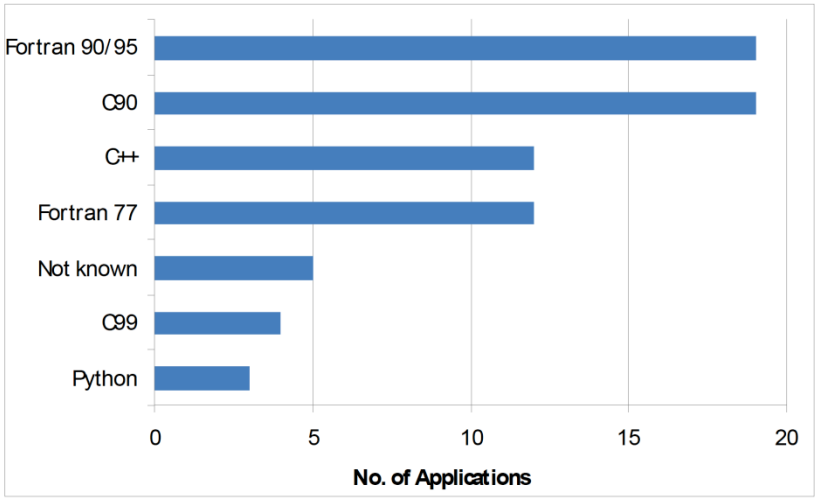

Figure 1: Application base languages. Reproduced from [7].

Each of the major European HPC service providers was surveyed on applications accounting for greater than 5% of system utilisation. Information was gathered relating to a total of 57 distinct applications. Figure 1 shows base language utilisation (noting that the total number is higher than 57 since some applications use more than one base language). In can be seen that Fortran, C and C++ account for the vast majority of total usage, with Fortran (Fortran 90/95, Fortran 77) being the most popular, followed by C (C90 + C99) and then by C++. The only other reported language is Python, used in a few applications.

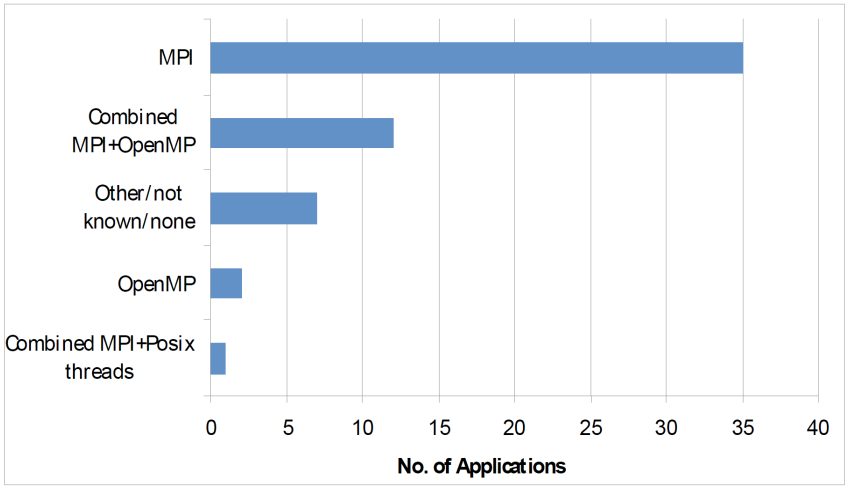

Figure 2: Application parallelisation methods. Reproduced from [7].

Figure 2 shows the breakdown by parallelisation method. It can be seen that the vast majority of applications used MPI: some of these in combination with OpenMP. Sole OpenMP usage was small (which is not surprising since the systems involved are typically used for relatively large parallel jobs, and OpenMP is suitable for intra-node parallelisation only). The only other reported parallelisation method was that one application used Posix threads (combined with MPI).

A comparison with 2008 PRACE survey shows that there has been an increase in the proportion of the applications using C or C++ compared to those using Fortran. The proportion of applications using hybrid MPI and shared memory has increased also compared to the 2008 PRACE survey. The longevity of parallel HPC simulation codes makes it unlikely that there will be major shifts in these patterns over the next five to ten years.

Therefore, the results of this survey indicate that the vast majority of applications use the traditional programming methods and models described in the preceding section.

3.2.2 CRESTA Co-Design Applications

In this section, we briefly summarise the languages and parallelisation methods used in the CRESTA co-design applications. As for the PRACE applications, all use the "traditional" programming models already described. Further details on this may be found in the report accompanying CRESTA Deliverable D2.6.1 "CRESTA benchmark suite".

3.2.2.1 GROMACS

GROMACS is written in C and C++, with optional inline x86 assembly code and/or CUDA. Parallelism is a hybrid of MPI and OpenMP.

3.2.2.2 ELMFIRE

ELMFIRE is mainly written using Fortran90, with some C used for auxiliary functions. The code is single-threaded, with pure MPI parallelism.

3.2.2.3 HemeLB

HemeLB is written in C++ with parallelism via MPI. A hybrid version, mixing OpenMP with MPI, is expected in the early part of the CRESTA project.

3.2.2.4 IFS

IFS combines Fortran (Fortran90 and Fortran95) with C. The parallelism is implemented using a hybrid of MPI and OpenMP.

3.2.2.5 OpenFOAM

OpenFOAM is implemented using C++ with parallelism via MPI only, although some work has been done on hydridising certain solvers using OpenMP.

3.2.2.6 Nek5000

Nek5000 is written using FORTRAN77 and C. Parallelism is via MPI only.

3.3 Suitability for Future Architectures

3.3.1 Increasing Numbers of Cores

The PRACE survey discussed in Section 3.2.1 contained another interesting finding.

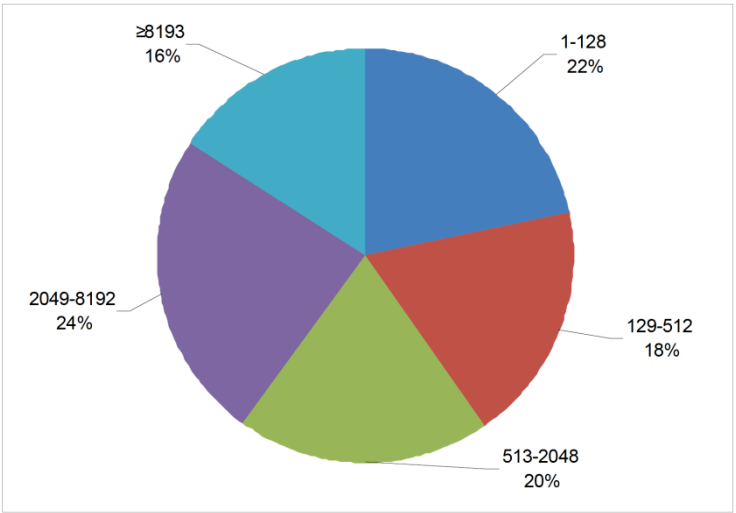

Figure 3: Distribution of total utilisation, in terms of number of cores used per application. Reproduced from [7].

Figure 3 shows a profile of application sizes, in terms of numbers of cores used; 84% of the applications use less than 8192 cores. The peak performance of today's

processors is of the order of 10 GFlops/core. Therefore, in order to scale to an exaflop using today's CPUs, applications would have to scale to at least 100 million cores. This highlights the problems we face going forward: even if it were possible to build exascale systems with standard CPU technology, there is an enormous gap in terms of how well applications can scale today and how well they would have to scale to utilise an exaflop.

3.3.2 Vector Hardware

To maximise performance on current and future computing systems, applications must fully exploit each microprocessor in the system, whether it is using a desktop PC or the largest of supercomputers. This means exploiting the parallelism within the chip not just at the "core" level (i.e. running across all cores in parallel) but also, crucially, at the "vector" level (taking advantage of the fact that each "core" can simultaneously process vectors containing multiple data elements). The latest CPUs from Intel and AMD can typically process (double precision data type) vectors of length 4 concurrently. This will increase with future products; the upcoming Intel MIC architecture will debut with vector length of 8. Current Graphics Processing Units (see following section) already process larger vectors (albeit described with different terminology): the latest NVIDIA GPUs operate on "warps" of size 32. Therefore, it is increasingly important that applications are able to actually exploit such vector-level parallelism. Traditional languages such as C and Fortran do not have any mechanisms for exposing this vector-level parallelism; instead there is reliance on compiler auto-vectorisation. However, without extra information and assurances from the programmer compilers often find this a difficult task unless code is written in a very specific way.

3.3.3 Accelerated Hardware

Some early HPC architectures treated special-purpose hardware as an attached resource (for example the AMT DAP) that could be used for parts of an application that could be mapped to the specialist hardware. Increasing miniaturisation has resulted in a family of small integrated products (FPGAs, DSPs and GPUs) that can be integrated on the processor board or attached via a PCI expansion card. One of the most promising is the Graphics Processing Unit (GPU, sometimes also called a General Purpose GPU, or GPGPU) which is an attractive proposition as a target to offload computationally demanding sections of an application running on a standard CPU.

GPUs were originally designed to display computer graphics, but they have developed into extremely powerful chips capable of handling demanding, more general-purpose calculations. The GPU architecture is somewhat different to that of the more familiar CPU. Much more space on the silicon is devoted to computation: each GPU possesses hundreds of cores of a much simpler design than a full-featured CPU. Also, GPUs use "Graphics Memory", which is much faster at loading data than the traditional memory used by CPUs; memory performance being a very important aspect for many scientific applications. These differences offer great opportunities for boosting the performance of scientific applications, but at a cost in terms of software development effort; GPUs cannot be programmed with traditional computing languages, and delicate tuning is needed to fully exploit GPUs. This is because the simple GPU cores lack the CPU's considerable logic devoted to managing out-of-order execution. As a consequence, code must be tailored to fit a narrow hardware performance window, or suffer a very long execution time. This is true of many other accelerators as well, including Intel's MIC family.

A particular challenge (at least with current GPUs) is caused by accelerators having their own memory space, so the programmer must manage the memory transfers to and from the accelerator; a task that is not supported in traditional programming languages. In future products, there will probably be closer integration between CPU and GPU hardware, perhaps alleviating this issue, although it is an open question as to whether cache-coherency will be achievable for the CPU and GPU.

3.3.4 Node Level Computational Performance

The increasing performance of future systems is likely to come largely from increased performance at the node level, rather than dramatic increases in the number of nodes. In the short term we will see increasing number of CPU cores per node, possibly combined with accelerators consisting of high numbers of simpler, GPU-like cores. In the longer term we may well see closer integration, or a blurring of the distinction between CPUs and accelerators, but nevertheless each node will likely contain small numbers of sophisticated CPU cores plus high numbers on simplistic number-crunching cores. The memory per node will increase, but not at the speed of the computational increase, i.e. the memory per core will decrease. Further information on these trends can be found in CRESTA Deliverable D2.1.1 “Architectural developments towards Exascale”.

The commonly-used method of using a separate message-passing task per core, and subsequently partitioning the memory, is unlikely to be appropriate or feasible on very high numbers of simple cores due to memory, computational and/or communication overheads. A more natural way to abstract the hardware is to use a hybrid model with small numbers of message-passing tasks on each node comprising larger numbers of some sort of lightweight threads.

3.4 Novel Programming Methods

3.4.1 Partitioned Global Address Space (PGAS) Languages

PGAS languages differ from the “traditional” models described above by introducing the concept of a global memory space that is partitioned between the participating processes (like ranks in MPI), with each process able to access both local and remote memory. Access to local memory is via standard sequential program mechanisms, and access to remote memory is directly supported by the new features of the language and is usually done in a “single-sided” manner (unlike the double-sided, matched send/receive API of, for instance, MPI). This not only enables (in principle) more productive parallel programming, it also allows the compiler to perform type checking and to optimise communications, since the compiler is aware of both the computation and communication requirements of the source. There are no penalties for local memory access. The single-sided programming model is more natural than the message-passing alternative for some algorithms, and it is also a good match for modern networks that support Remote Direct Memory Access (RDMA), potentially offering performance advantages over message-passing techniques.

3.4.1.1 Co-Array Fortran

Fortran Co-Arrays are an example of a PGAS model and a relatively new mechanism for performing communications in parallel Fortran applications. The Co-Array Fortran model [8] was introduced in its current form in 1998 as a simple extension to the Fortran 95 language, and a subset of core features was formally integrated into the Fortran 2008 standard (where the spelling was changed to “coarray”). The expectation is that additional features will be published in a Technical Specification in due course. At this time, there is limited availability of Fortran implementations that support coarrays, although since they are now part of the standard the situation is expected to improve.

Co-array Fortran programs follow a Single Program, Multiple Data (SPMD) model: like MPI, a single program is replicated a fixed number of times. Each replication is referred to as an “image”, and images are executed asynchronously. The execution path may differ from image to image: each image has a unique identifier that can be used in control statements. A new, *co-dimension* syntax is used in addition to the standard array dimension syntax. For example,

```
real :: x(10)[*]
```

declares a co-array that has a (standard) dimension of size 10 on each image. The square brackets define the codimension, with * allowing the number of images to be

specified when the job is launched. There is a separate copy of `x` on each image which can be locally accessed in the normal way (e.g. `x(4)` refers to the 4th element of the local `x` array). The co-dimension syntax allows remote memory accesses, e.g. `x(4)[j]` will access the fourth element of `x` on the jth image, which will be a remote access for all images except image j. There are a number of additional features, e.g. regarding synchronisation of images and the retrieving of information about images.

3.4.1.2 Unified Parallel C

Unified Parallel C (UPC) [9] is analogous to Co-array Fortran in the sense that it extends the C programming language to provide support for explicit parallelism and the ability to access remote, as well as local, memory. However, there are differences in the way that this is done. Furthermore UPC is not part of the C standard, but instead an extension to the language. Both commercial and open source compilers are available.

Similarly to CAF, UPC is a PGAS model and programs operate in Single Program, Multiple Data (SPMD) fashion: multiple processes execute the same program, but the execution paths can differ. Whereas CAF uses the term image to refer to one process, UPC uses the term *thread* (noting this has a different meaning to, e.g. OpenMP threads). To allow threads to access both local and remote memory, UPC provides at the program level the concept of two memory spaces: private and shared. Objects declared in private memory space use regular C declarations, e.g.

```
int x; // private variable
```

and are only accessible by a single thread. Objects declared in shared memory space using the "shared" identifier, e.g.

```
shared int y; // shared variable
```

are accessible by all threads. The shared memory space is used to communicate information between threads. All threads can directly access shared data, even if it resides in a remote location. UPC creates a logical partitioning of the shared memory space, and shared objects have affinity to a specific thread. Shared arrays can be declared with the affinity distributed between threads (and additional syntax allows control over the specific decomposition). This affinity corresponds to physical locality: better performance will be realised when a thread accesses data to which it has affinity (i.e. it resides on the local node) compared to when it accesses data which has affinity with another thread (which resides on a node on which that thread is running). It is up to the programmer to always keep data locality and affinity in mind when designing programs that perform well.

Computation on shared distributed arrays can be shared among a set of threads. UPC has built-in mechanisms for explicitly distributing and sharing work, such as a `forall` loop, similar to a standard `for` loop in C but with the loop iterations distributed. There are a number of additional features, e.g. regarding synchronisation.

3.4.1.3 OpenSHMEM

Whilst it is not exactly a PGAS language, there is increasing interest in the SHMEM communications model, which can be called from any of the traditional programming models via library API calls. SHMEM provides a SPMD execution model (like CAF and MPI), with a single-sided model of communications. The recently formed OpenSHMEM standards group [10] is currently finalising version 1.0 of an OpenSHMEM standard, unifying a number of vendor-specific variants.

3.4.1.4 Interoperability

On many HPC systems there is considerable advantage in using PGAS languages. Given that CAF and UPC are extensions to existing, interoperable languages it is reasonable to expect some degree of interoperability between the parallel programming models, especially as (on a given platform) they probably share a common underlying software stack. Interoperability would allow users to incrementally

port their application, assessing performance and productivity advantages as they do this. Having to completely rewrite an application in a relatively untested (at least from most users' point of view) programming model is a significant barrier to widespread adoption.

The main problem with such interoperability is that each of the languages and parallelisation models is standardised in isolation, and no standard formalises this interaction with other programming models.

In most (if not all) implementations, CAF interoperates with MPI. The image number corresponds directly with the MPI rank. This makes it simple to replace MPI send operations with CAF puts. This allows incremental porting of a Fortran MPI code, taking advantage of the PGAS features where appropriate. In practice, however, the interoperability can be complicated by the use of MPI features like subcommunicators and datatypes.

The same statements hold for the interoperability of SHMEM with MPI and with CAF.

It is possible for there to be some degree of interoperability between UPC and MPI and/or SHMEM, but this is not well explored at present.

3.4.1.5 Other PGAS Languages

Co-Array Fortran and UPC can be regarded as PGAS extensions to the traditional Fortran and C languages respectively. There also exist other, more radical, PGAS languages that are currently less mature and popular. Three of these emerged from the DARPA-led High Productivity Computing Systems programme: X10, Fortress and Chapel. X10 [11] is a parallel object-oriented language originally developed by IBM. The syntax is very close to Java and C++, but the language provides parallel programming concepts such as "places" which provide an abstraction of some subset of the hardware and "activities" that execute efficiently on that subset. Fortress [12], originally created by Sun Microsystems, also provides a global-view of the parallel program, and it uses the terminology of a "location" to refer to a specific memory space and the threads operating on that memory space. The Fortress syntax, though, is quite different to X10: it is an evolution of Fortran that aims to closely resemble mathematical operations, and hence be a productive tool for scientists. If developers can program using high-level mathematical concepts then underlying parallel implementation can be handled implicitly.

Chapel [13] aims to improve the programmability of parallel computers by allowing a higher level of algorithmic expression and by improving the separation of this from the details of how data is distributed across the system. A particular feature of Chapel is "multi-resolution": users initially write very abstract code and then incrementally add more detail until they are as close to the machine as their (performance) needs require.

Whilst Chapel has an imperative block structure (as in Fortran, C and C++), it currently lacks interoperability with other programming models, as do the rest of these novel languages, which is a considerable barrier to widespread uptake and exploitation.

3.4.2 GPU Programming Models

Systems in which the traditional CPUs are augmented by Graphics Processing Units (GPUs) are becoming more common. The GPU acts as an "accelerator" to the CPU: most lines of application source code are executed on the CPU (using the standard serial computing model of computation) and key computational kernels are executed on the GPU (using the stream computing model of computation), taking advantage of the large number of cores and high graphics memory bandwidth on offer, with the aim that the code as a whole performs better than if the CPU was used alone.

In the November 2011 edition of the Top 500 Supercomputer rankings [14], 37 of the 500 systems were of this nature. Of these, 35 contained GPUs manufactured by NVIDIA while the remaining 2 contained GPUs manufactured by AMD.

3.4.2.1 NVIDIA CUDA

Compute Unified Device Architecture (CUDA) [15] is the proprietary interface to the NVIDIA architecture, and consists of extensions to C/C++ that allow interfacing to the GPU hardware. An analogous Fortran version of CUDA ("CUDA Fortran") is available as a commercial third-party product from the Portland Group compiler vendor [16].

GPUs operate under the "stream computing" model of computation, where the data set is decomposed into a stream of elements. A single computational function (or kernel) operates on each element: multiple cores can process multiple elements in parallel. This model is obviously only suitable for data-parallel problems. The NVIDIA GPU is partitioned into Streaming Multiprocessors (SMs), with multiple "CUDA cores" per SM, which operate in a vector fashion. There are less scheduling units than cores on each SM and threads are scheduled in "warps" of size 32. Threads within a warp always execute the same instruction in lock-step (on different data elements). In CUDA, this hardware is abstracted as "Grid" of "Thread Blocks". The multiple blocks in a grid map onto the multiple SMs. Each block in a grid contains multiple "threads", mapping onto the cores in an SM. CUDA extends C with the required new syntax for specifying grid/thread block decompositions, defining and launching kernels, and managing data transfers between the separate CPU and GPU memory spaces.

We illustrate by way of simple example. Consider the following sequential vector addition loop:

```
for (i=0;i<N;i++){
  c[i] = a[i] + b[i];
}
```

To run this on the GPU using CUDA, we first need to use CUDA API calls to allocate memory on the GPU, and copy the `a` and `b` arrays to the GPU memory. Then we launch a kernel on the GPU:

```
vectorAdd<<<N/256, 256>>>(a, b, c);
```

and finally, again using API calls, copy the `c` array back to the CPU and free GPU memory. The `<<<…>>>` notation is used to define the decomposition of loop iterations over the parallel threads. Here, we have specified a 1D decomposition (where 2D and 3D decompositions are also possible), with 256 threads per block, and N/256 blocks. The `vectorAdd` kernel is defined as a function with the `__global__` qualifier:

```
__global__ void vectorAdd(float *a_gpu, float *b_gpu, float
*c_gpu)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  c_gpu[i] = a_gpu[i] + b_gpu[i];
}
```

where the internal variables `threadIdx.x` and `blockId.x` are unique thread/block identifiers, and `blockDim.x` is the number of threads per block. The kernel is therefore executed by each of the N threads, with each thread responsible for a single element of the vector.

As this example demonstrates, CUDA is very powerful in providing functionality to flexibly utilise the GPU hardware, but has the serious caveat that the resulting code is much more complex than the equivalent CPU code.

3.4.2.2 OpenCL

Open Compute Language (OpenCL) [17] is a cross-platform API that can be used to program not just GPU-accelerated systems, but also other heterogeneous and/or multicore architectures. When used to program GPU accelerated systems, the model is

very similar to that of CUDA. There are similar abstractions and basic functionality, with differing terminology, e.g. a “Thread” in CUDA is expressed as a “Work Item” in OpenCL. The key advantage of OpenCL over CUDA is that it is portable to other systems, such as those utilising AMD GPUs, although adaptations still may be required for performance. At this point in time CUDA remains more mature and well documented for use on NVIDIA systems. OpenCL addresses the hardware at a slightly lower level, resulting in more work for the programmer. For example, OpenCL does not add any new language syntax so the launching of each kernel involves a series of library calls; in CUDA this is hidden from the user by use of the `<<<…>>>` syntax (see the previous section).

3.4.2.3 Accelerator Directives

Directive-based programming of GPUs and other accelerating co-processors is a relatively new, high-level programming model that provides an alternative to rewriting applications in GPU-specific languages (such as CUDA or OpenCL). As has been discussed above, the vast majority of large-scale, parallel scientific codes are written in the traditional HPC languages Fortran, C and C++. Many of the algorithms used are, or can be made to be, extremely suitable for running on the vector-like architectures of GPUs. Rather than rewriting, a better approach (from a developer productivity viewpoint) is to provide a mechanism for compilers to generate executables that can run on the GPU from the original source code.

Most accelerators have a radically different architecture to a typical CPU. In particular, they have a narrow performance range, and kernels with the wrong loop structure or pattern of memory access will run with extremely low efficiency. It is unlikely, then, that any compiler would be able to automatically generate efficient executables for the accelerator without guidance from the application developer. Accelerator directives provide a mechanism for the developer to add such guidance for the compiler, by means of non-executable statements (specially-constructed comments) in the code. A non-accelerating compiler that targets a CPU would ignore these, but an accelerating compiler can interpret these as prescriptive or suggestive information that control the creation of kernels that will execute on the accelerator.

Directive-based programming models are not new and OpenMP provides a good example that such models can be successful, both in terms of allowing the compiler to generate efficient code and in being widely adopted by the user communities.

The first widely available accelerator directive programming models were vendor-specific, from PGI [18] and CAPS [19]. Whilst these both have some use, widespread uptake requires vendors to feel that the programming model(s) have some permanency via an external standardisation process. There are currently two such efforts on-going.

The first is via the established OpenMP Architecture Review Board (ARB) standards committee [3]. A subcommittee was established to develop an extension to the existing OpenMP 3.0 standard that would target a wide class of possible accelerators. This would include GPUs, but also address other accelerators e.g. digital signal processors (DSPs). The work of this committee is on going, with a draft standard being discussed in weekly or biweekly conference calls. The basis for this draft standard was the PGI directive model, but almost all large-scale compiler vendors are represented.

More recently, it was recognised that while the OpenMP specification evolved, there was a need for a minimal, interim standard to serve early adopters of the directive-programming model. This would provide a published document describing the model (as opposed to the fluid draft OpenMP proposal) and would allow portability of code between multiple compiler vendors.

To this end, the OpenACC standard [20] launched in November 2011, with support from NVIDIA (as GPU hardware manufacturer) and compiler developers Cray, PGI and CAPS.

For most simple examples, the differences between all these programming models are largely semantic. By way of example, we focus on using OpenACC directives to

accelerate a two-dimensional stencil calculation followed by calculation of a residual (ignoring the fact that really only one loopnest is required for this simplistic example):

```
!$acc data copyin(b) copyout(a)
!$acc parallel loop
DO i = 2,N-1
  DO j = 2,M-1
    a(i,j) = b(i+1,j) + b(i-1,j) + b(i,j+1) + b(i,j-1)
  ENDDO
ENDDO
!$acc end parallel loop
residual = 0
!$acc parallel loop reduction(+:residual)
DO i = 2,N-1
  DO j = 2,M-1
    residual = residual + a(i,j)
  ENDDO
ENDDO
!$acc end parallel loop
!$acc end data
```

Each loopnest is compiled as a kernel on the accelerator. The compiler chooses a default loop schedule (division of loop iterations across the computational cores of the accelerator). Optional clauses on the directives can be used to influence this choice. As in traditional OpenMP, reduction variables are identified using an appropriate clause.

Given the separate memory spaces of the CPU and GPU, good performance can only be gained by minimising data transfers. The enclosing data region instructs the compiler to minimise data traffic by only moving the data arrays before or after the two kernels (rather than for each one), and movements are tuned to match the read-only and write-only use of the arrays in the loopnests. If a variable (scalar or array) is not mentioned in a data region, data movement occurs at the boundaries of the parallel loop region.

With this level of control, the compiler has the potential to deliver code that performs well on the GPU as well as on the CPU. The developers of the Cray compiler, for instance, expect accelerator directive-generated code to typically deliver 90% or more of the performance of an equivalent code hand-written in CUDA.

Being based on the original source code, interoperability of the generated kernels with the host, CPU code is assured. More importantly, there is increasing work to support interoperability of the directive-based model with more established GPU-targeted programming models like CUDA.

4 Compilation and Runtime Environments

4.1 Compilers for Exascale Computing

The fundamental job of a serial compiler is to take the programmer's high-level, generic, human-readable source-code description of some algorithm or operation and translate it into low-level machine code that is specific to a particular CPU architecture. In order to be as general-purpose as possible, modern compilers defer the architecture-specific aspects to a very late stage, performing code transformations using some architecture-neutral intermediate representation of the source code. Compilers also deal with multiple high-level languages by converting all of them to the same intermediate representation.

Modern compilers are all generally good at improving code performance through “classical” optimisation techniques such as removal of redundant code, storing temporary values in registers rather than memory, function inlining, loop unrolling etc. However, the ever increasing gap between the rate at which CPUs can issue instructions and the speed of DRAM means that the limiting factor in many codes is the way in which data is read from and written to memory, rather than the rate at which floating-point instructions are issued. This means that the most important optimisations concern how user data is laid out in memory and how it is accessed within loops.

Most compilers can already address some of these issues. For nested loops, the loop ordering may be changed so that memory access is done efficiently (e.g. to match Fortran row-major array storage order). Loops may be split into many separate chunks (a technique called “tiling”) to improve data re-use and make better use of the CPU's caches. Prefetching may be inserted to load data from memory well before it is required to try and hide memory latencies. However, all these classical techniques suffer from three major limitations.

Firstly, the executable is generated based purely on the information in the source code. In reality the optimal set of loop transformations depends critically on the dimensions of the arrays being processed and the extents of the loops being executed. In many cases these are not known at compile time and will depend on the data set. Since a single application is typically run against multiple data sets this implies that separate executables may be required for different runs of the same code. It is possible to generate a single executable containing multiple versions of a routine and choose the most appropriate one at runtime based on the specific input parameters. However, this technique cannot cope with the huge number of possible options for real codes.

Secondly, the compiler is optimising against some theoretical model of the CPU architecture, e.g. it may assume that a certain amount of cache is available per core and tile the loops to fit this size. This may have been effective in the past with simple memory architectures, but modern multicore processors are becoming so complicated that it is almost impossible to make accurate predictions about achieved performance from simple theoretical models.

Thirdly, all the optimisations performed are local, i.e. confined to the operations performed in a particular function, set of loops or basic code block. Since memory access patterns have such a large impact on performance, global choices such as the order in which array dimensions are defined can also be extremely important. Although a compiler can perhaps change the order of loop execution over the various dimensions of an array, it cannot change the order in which these dimensions are declared. In many cases the loop order may be fixed due to the requirements of the algorithm. In such cases the only way to improve memory access patterns is to change the array declarations, but currently this can only be done by the programmer as it has global consequences for the correctness of the entire program. These layout issues are becoming even more relevant as the optimal loop ordering for GPU accelerators (to enable the memory coalescing that is essential to achieving good performance) is often the opposite of that required for cache-based microprocessors (to enable stride-1 access).

A possible solution to the first two problems is to change the way in which the compiler chooses its optimisation strategy. Rather than trying to predict the performance of various optimisations based solely on the source code, an empirical method is adopted where an executable is generated and the performance measured by running it on the target CPU with a real data set. Code optimisation then becomes a classical discrete optimisation problem, searching for the optimal value of some objective function (e.g. minimum runtime) against a set of integer variables (e.g. the size of each loop tile, the unrolling factor of each loop etc.). Note that this approach can easily be applied to any other experimentally measurable metric, e.g. minimum energy consumption or minimum memory footprint.

This approach has been investigated in a previous study - crucial to its feasibility is that it “requires a compiler to be able to generate different codes rapidly during the search by adjusting parameter values, without costly compiler reanalysis. It also demands that the compiler have a clean interface to a separate parameter search engine" [24]. The CHiLL framework is used which represents loops from real codes in a symbolic form that is amenable to various transformations. These transformations are specified in a separate script and might include swapping the order of some loops, tiling them with a specified tile size and unrolling others by some given length. Given this script, CHiLL can perform source-to-source translation of a program that is then compiled with a standard compiler (where any inbuilt compiler options for loop transformation are disabled).

This must then be coupled with some efficient optimisation framework as exhaustive parameter searches are prohibitively expensive. In [24] this is done by the Active Harmony system that couples an efficient simplex-based optimisation algorithm with the ability to execute many trial runs in parallel on multi-CPU systems. Encouraging results have been obtained for common HPC kernels. However, it does not appear that this process is entirely automated. Although the system automatically optimises the variables that parameterise a CHiLL script, it appears that the script itself must be supplied by the programmer. For example, although the optimal tile size might be found automatically, it is up to the programmer to decide which loop(s) should be tiled.

Although it produces efficient code, there is a large overhead to this approach due to the need to execute the code in question many times. In [44] an alternative approach is taken which initially performs an equivalent empirical optimisation on a training suite of selected test codes. The results of these runs are all fed into a machine learning system. When compiling a user code (as opposed to one from the training suite), the code is compared to the results from the training suite and the best optimisation approach is chosen based on the similarity between the user and training codes. The role of the machine learning system is to perform this matching based on certain code characteristics. The choice of these characteristics is crucial to the success of this approach: both static and dynamic features were used, the latter including data from hardware performance counters collected during actual runs of the unoptimised code.

For thread-based programming models this approach can be extended to optimisation of parallel codes [44]. In directive-based models such as OpenMP (shared-memory) and the new OpenACC API (GPU accelerators), the parallelisation statements include various integer parameters specifying the way that the loop should be split up amongst threads. Examples include the scheduling type and chunk size for OpenMP or the vector length in OpenACC. These parallel parameters can be optimised in an analogous way to the serial ones discussed above. Sophisticated techniques also exist to predict the general scaling behaviour from runs on a restricted range of threads using adaptive neural networks.

This approach relies on the parallelisation already having been done by the user. In [45] this is extended so that the compiler automatically identifies potential parallelism in the code in addition to optimising its implementation. This is extremely hard to do from a simple static analysis, so the approach here is to actively involve the user. If the compiler discovers that parallelisation might be possible, but cannot be certain, it prompts the user for input rather than simply reverting to a serial implementation. The

user often has additional information (e.g. knowledge of the input data set or the range of possible values of certain variables) that can allow for parallelism in the specific cases of interest to the user even if it is not possible in general.

To move from applying these techniques to small kernels up to full-scale applications requires additional tools. In [46], standard profilers are used to identify performance critical regions, and these code sections are separated out into stand-alone functions by the "outlining" tool ROSE. This outlining procedure could in principle be done manually, but might become too laborious in practice if the performance profile is very flat across the application. Overall performance improvements of more than a factor of two are achieved for the SMG2000 benchmark application. Although SMG2000 is a reasonable sizeable piece of code (with over 50K source lines), it spends more than half its time in a single nest of four loops containing a single line of code. This makes it absolutely ideal for optimisation, and it is therefore not clear how well this approach will scale to larger, more complicated applications.

The most elegant way to attack the global problem of optimising the data layouts is for the application to employ abstract data types, where the user only specifies the data operations and the details of their implementation (e.g. the layout of data in memory) is opaque. If a code is written in this way then the optimisation techniques described above could be used on the small sections of code that implement the operations. This strict distinction between form and function is, however, not very common in most real HPC codes where large arrays are declared explicitly in the computation routines and accessed directly. In this case, special tools are needed to transform data layouts. FTRANS [47] is an example, which does source-to-source translation, requiring the user to annotate all array declarations in the original code. Although FTRANS can transform array declarations, loop transformations require an additional tool.

All this work has been done for shared-memory parallel models, whereas any exascale machine will have to have a distributed memory architecture to scale beyond a few hundreds of cores. The problem for distributed memory is much harder. For example, message-passing MPI calls simply appear as generic external functions to a serial compiler and there is little that can be done to optimise them: the compiler has no idea at all what they do. Although a user could in principle annotate the MPI calls to give additional information to the compiler (e.g. specifying when the data required for a send operation is ready and when it can safely be overwritten), or be prompted for this information by the compiler, this approach does not appear to have been investigated in practice.

PGAS languages should be easier to address as communications are part of the syntax and so the compiler has much more information to work with. However, it is still a very hard problem and may require additional information (again via directives or direct user interaction) to be useful in practice. Such directives could state that a particular array is not accessed remotely within a certain section of code, or is only read remotely and not written. Although this might be obvious to the programmer, it may be difficult or impossible to ascertain purely from the source code. As compilers always aim to produce correct code, they have to make pessimistic assumptions whenever there is any doubt. A directive of this type is already implemented in the Cray compiler for Fortran coarrays [48], stating that a parallel operation does not need to complete immediately but can be delayed until the next synchronisation point. Although it might appear at first sight that this directive is not needed, a strict interpretation of the language standard means that the compiler cannot make what would seem to be obvious assumptions in the absence of this additional information.

In summary, there has been quite a substantial amount of promising work in the area of compilers for serial and small-scale parallel kernels. The challenge is in extending these techniques to large-scale distributed memory applications such as those in the CRESTA benchmark suite. This study needs to be done on the current generation of petascale machines in order to select the approaches that may prove successful at the exascale.

4.2 Autotuning

In the context of this project, autotuning is the process by which an application may be optimised for a target platform by making automated optimal choices of how the application is built and deployed. Tuning choices can be made that target algorithms, source, compilation and application launch. The complexity of today's software stacks and hardware architecture means that the inherent system complexity is outstripping the capability and feasibility of the individual programmer to make the optimum choices. Automated tuning for a particular architecture makes it much more practical to obtain portable performance across a range of target architectures. Furthermore some applications have lifecycles beyond a generation of hardware architecture and hence the ability to retune an application for a new hardware target would be advantageous even if initial deployment was for one particular architecture. This is a very challenging problem, as we will see, and typically autotuning has been limited to domain-specific libraries, serial optimisations or specific parallel application patterns. We want to approach the problem of whole application autotuning where the application is likely to be a parallel and possibly hybrid implementation running on an exascale platform.

Before looking at existing autotuning projects we should consider the areas that are amenable to tuning. These can be divided in various ways but we can look at different aspects in turn. The first consideration centres on choices made by the application writer:

- Choices in algorithm made by developers
- Algorithm implementation and optimisation choices
- Scientific Library choices (API to use and library to use)

For each set of choices made we have a version of source that should be compiled and run optimally on the target architecture. Ignoring any parallel aspects for now we can use compiler autotuning to optimise the code generation and library autotuning to optimise standard libraries in use by the application. Note in particular that the optimum tuning choice may depend on the problem size and target machine attributes. The build itself is amenable to optimisation with choice of compiler and compiler options.

Moving into the parallel domain we add decomposition choices, stencil operations, consideration of choices in how to perform communication patterns etc. At runtime there are launch choices and possibly decomposition choices for a hybrid (for example MPI/OpenMP application).

4.2.1 Novel Autotuning Tools

The most successful attempts at autotuning have been for domain-specific numerical scientific libraries. Notable examples are ATLAS [20] (BLAS), FFT [22] (FFTs) and SPIRAL [24] (DSP Algorithms). For library implementations the autotuning boils down to making an optimum choice from a set of kernels for a set of problem sizes. The library is built using experience from running many experiments to distil what is learned into making the correct choice for a given problem size presented to the library.

Library autotuning is also used to develop commercial products. For example the Cray LibSci product includes autotuned implementations of dense and sparse matrix operations and FFTs.

Public domain autotuning implementations are available, for example a combination of the Active Harmony framework and the CHILL compiler transformation framework can target compiler-based application tuning [24]. Another interesting project is the Collective Tuning project [25] which facilitates a machine learning approach to compiler optimisation using a database of program features with known performance characteristics.

Specialised hardware (DSPs and GPUs) have spawned an interest in autotuning, not least because various runtime parameters related to kernel launch and memory access choices are crucial in obtaining best performance. Some researchers are performing rudimentary exhaustive search to find the best parameters and require better techniques. The OpenMPC [26] project is an example targeting GPU optimisation.

Other approaches are very high level, for example the PetaBricks [27] approach of incorporating algorithm choice into the language. It is also possible to use domain-specific abstractions which limit the tuning to the crucial aspects of a given problem domain – for example OP2 [27] can usefully use autotuning for the GPU backend.

4.2.2 Autotuning in CRESTA

Given all this activity there is no one approach that covers all aspects of application tuning, there is no consistent way to mark up choices and expose those to an autotuning framework and autotuning is not something in the mind of the average HPC application developer.

Our goal is to define such a consistent approach that can target mark-up of choices at all the stages mentioned (algorithm choice, source, compilation and launch) and hopefully in a way that is not too onerous far the application developer. (This means that we need to be able to interoperate with any automated build or metaprogramming tools to the extent that proves possible.) In particular we want to pay attention to important parallel tuning aspects (stencils, communication patterns etc.). We would also support definition of independent tuning spaces, plugin tuners for specific aspects and feedback mechanisms required for those tuning frameworks that can optimise based on execution metrics (for example performance counter metrics) beyond the primary optimisation metric. The ideal situation of the future would be one where the application developer thinks of exposing such choices as part of application development. If the DSL we develop is general enough then it can be used to wrap specific autotuners without excessive effort.

We expect to spend some time studying the status of various autotuning projects to determine how useful they are to the HPC community in general and to the CRESTA application set in particular. This will also inform the development of the DSL (which will be more general than the currently available tools and should also allow us to do some integration at a later stage in the project).

4.3 Runtime Systems

The goal of runtime systems is to execute applications having certain features efficiently on a specific system eliminating the complexity of managing low-level or system-dependent operations such as task scheduling and resource management. Runtime systems are expected to play an important role in exploiting exascale architectures successfully for two main reasons. In the first place, exascale-computing platforms will likely be a heterogeneous mix of CPU and accelerators. Therefore, runtime systems are expected to assist programming libraries and compilers to exploit a heterogeneous computing platform. Secondly, exascale systems will demonstrate high variability in performance and availability of the components. The runtime systems will have the important role of scheduling tasks and redistributing data as well as adapting dynamically to a changing system status or a varying application performance. In the following subsections, some runtime systems aiming at tackling these challenges are presented. This overview shows a common denominator in using threads heavily. Furthermore, all systems provide some service to transfer data between the processing elements. All runtime systems provide beyond the varied use of threads different means to address specific aspects of their application field. These are for example implementations for the data transfer between nodes or a software layer providing virtually shared memory spanning over heterogeneous processors with local memories.

In addition, using runtime systems to decrease power and energy consumption while still retaining performance will be essential in exascale supercomputers. Two energy-aware approaches are described.

4.3.1 Novel Runtime Systems

4.3.1.1 StarPU

StarPU is directed towards computers with heterogeneous multicore designs and special hardware such as coprocessors and accelerators [29]. The programming model defines tasks called here “codelet”. They can encapsulate existing functions. Furthermore, codelets can have multiple implementations for different processors. A scheduler tries to run the codelets as efficient as possible. Also a data management library is part of StarPU. It provides transparently the needed memory for a codelet on the scheduled resource.

The specification of the computational tasks of StarPU’s programming model requires their implementations possibly for several processing elements as well as a definition of the input and output data. Data will be moved between different memories of the heterogeneous computer system based on this definition if needed. Tasks are submitted asynchronously and their termination is given to notice by callbacks.

StarPU implements a software shared memory system for the heterogeneous processing elements. This memory system transports required data to processing elements in time before the start of the execution of the codelets. It is possible to have multiple copies of certain data in the memories of different processing elements. Another important task of the shared memory implementation is the bookkeeping of the memory status. For this StarPU applies a basic cache-coherence protocol, the MSI protocol (Modified-Shared-Invalid).

Furthermore, StarPU provides filters that support the subdivision of data for block- and tile-based algorithms. These filters make it possible to reduce the data movements between processing elements according to the schedules during the runtime of a program. The data transfer of the StarPU runtime system is implemented asynchronously, and matches in that way the widely available features of modern accelerator devices.

Tasks are submitted with push operations from the application to the scheduler. It is possible to describe the expected performance with additional programming effort. The scheduler pops tasks to the processing elements using different scheduling algorithms. These algorithms can range from a simple FIFO queue to sophisticated algorithms considering resource usage policies and cost-models can make scheduling decisions.

StarPU uses a history-based performance prediction model. It traces the execution time of tasks and uses it in subsequent schedule computations. For this purpose it is assumed that tasks possess similar characteristic features and complexity if they are called again with the same parameters in a given application.

4.3.1.2 StarSs

StarSs is a programming environment for parallel applications based on task-level parallelism that is managed on the node-level [30]. Tasks are realised as functions and managed by the runtime system. The task specification is done by compiler pragmas.

Functions that are annotated with compiler pragmas form tasks that are managed by the runtime system. The annotations specify input and output parameters and variables that are used for data reductions.

By these means the following features of the runtime system are obtained: data dependency control, data dependency reduction, and workload distribution.

Data are produced in tasks and travel from task to task during the program execution. StarSs analyses the input and output parameters of the annotated tasks with respect to the dependencies ‘read-after-write’, ‘write-after-write’, and ‘write-after-read’. The dependency analysis allows the calculation of a possible parallel execution order as a

graph that respects the required serial executions coming from the data dependencies. The task execution is managed accordingly to this graph by a multithreaded runtime system that takes the next executable tasks and starts them.

The outcome of the dependency analysis is also used to use register renaming. Register renaming is a technique used in compiler construction to remove data dependencies at the cost of larger amounts of memory. Its application allows the removal of read-after-write and write-after-write dependencies.

The scheduling algorithm takes several measures to ensure good performance with respect to the workload distribution. It tries to run tasks that are dependent on each other in the same thread.

The main thread runs the application, creates tasks and assigns them to the queues of worker threads. It is possible to use work stealing between processors.

4.3.1.3 ForestGOMP

ForestGOMP is a runtime system consisting of a multi-level thread scheduler and a NUMA-aware memory manager [31]. It aims at dynamic load distribution under consideration of the application structure and the hardware topology. The implementation of ForestGOMP showed an improvement to applications with nested, massive parallelism.

ForestGOMP is an extension of GNU's OpenMP implementation on top of a flexible scheduling framework called "BubbleSched". This framework in turn is implemented on top of a thread library called "Marcel". The library Marcel again uses a library named "hwloc" to handle the hardware topology and implements MAMI (Marcel Memory Interface). This is a NUMA-aware memory manager.

The library hwloc was first developed at first as part of the scheduling framework. It has been recently externalised and can now be used as a standalone component to manage the hardware affinity of threads on current multicore systems.

Hwloc is used in ForestGOMP to provide a description of the hardware during the runtime to BubbleSched. BubbleSched builds an analogously structured hierarchical set of queues. The hierarchy levels in the queue set correspond to the levels found in the hardware: machine (node), NUMA nodes, and cores. The scheduler places every thread into a certain queue according to a scheduling policy or specifications of where it should run. So-called "bubbles" are used as a grouping mechanism expressing certain affinities. BubbleSched is responsible for the execution of hierarchical organised bubbles on the hardware used.

MAMI, the NUMA-aware memory interface, gets a description of the available memory hierarchy during the runtime, from the hwloc library. The interface supports memory binding and interleaving. Additionally, it provides possibilities to migrate data. Such a move can be executed synchronously upon request of an application or by application of a next-touch strategy. Furthermore, it provides some information about the costs of reading, writing and migrating data to the application.

The functionality of memory allocation with a next-touch strategy complements the first-touch strategy that is typically well supported by operating systems already. The next-touch strategy allows the marking of a memory page or region to have affinity with a certain NUMA node during the next use. It is important to be able to do this because a node not using the data most intensive could allocate the memory due to the first-touch strategy.

ForestGOMP is able to place data and execute threads on preferred NUMA nodes based on these building blocks. The scheduling policy is implemented within a memory bubble scheduler. This scheduler makes an initial distribution of data and threads as well as executes work stealing if cores become available for other work.

The distribution of data and threads is based on so-called memory hints. They are provided by MAMI through the BubbleSched statistics interface and contain information about the data and their usage intensity by threads. MAMI calculates the data location

and summarises the information about bubbles that allows BubbleSched to make reasonable scheduling decisions.

It is possible to implement different work-stealing algorithms for each different bubble scheduler. A scheduling algorithm used within a NUMA node respects the memory hierarchy and cache structure by using only the local cores of the node. Another stealing algorithm is directed to perform the stealing from remote nodes and to migrate the assigned memory. This again is done under consideration of the hardware topology provided by hwloc and the amount of data used by a thread to optimise the costs of migration.

4.3.1.4 Charm++

Charm++ is a parallel programming system focusing on the enhancement of programming productivity, as well as on the requirement to maintain scalability [32]. The application NAMD, for biomedical computation, is a well-known example that has been implemented using it. Charm++ uses overdecomposition, processor virtualisation and migratable objects to provide load-balancing and dynamic resource management.

Charm++ programs consist of C++ objects that are grouped into collections and use asynchronous method invocations for communication. These objects, so-called *chares*, can be migrated to other processors by the runtime system. The base mechanism of program execution works in the way that the scheduler invokes methods of objects that are selected from the queue of the processor where the runtime system resides. Other possible execution styles are ‘threaded’ and ‘sync’. Threaded methods can block and return the control to the scheduler during the execution while sync methods block the caller until their termination. Messages are also delivered to the correct destination objects after migrations.

The migration of objects as well as of threads provides a dynamic load-balancing capability. The runtime system instruments the communication and workload for the purposes of dynamic monitoring. Different schedulers can distribute the work based on this performance monitoring as well as on information about the topology of the interconnect network.

Charm++ programs typically have a much higher number of chares than processors. Furthermore, the normal method invocation is done asynchronously. Both features can be used to hide communication latencies and to overlap communication and computation. Additionally, the runtime system allows the addition or removal of processors dynamically during the runtime.

Additional tools are available beyond the basic Charm++ runtime system. Amongst them is Adaptive MPI (AMPI) that is an implementation of the MPI standard on top of the Charm++ runtime system. It makes it possible to provide automatic load balancing to MPI applications. This is done in that way that AMPI introduces virtual processors that are implemented as user-level threads bound to Charm++ objects. Such threads can be assigned to a physical processor and migrated if necessary for load-balancing reasons. Message passing is implemented as communication between Charm++ objects and handled by the runtime system. Furthermore, it is possible to shrink or to expand the number of MPI processes utilised by Charm++.

4.3.1.5 High-Performance ParalleX (HPX)

ParalleX [33] provides a runtime system addressing numerical simulations showing scalability problems when using classical technologies such as MPI. Examples are numerical simulations of relativistic phenomena in astronomy. They use algorithms based on adaptive mesh refinement methods that apply computational resources in such a way that comparatively more calculations are done in parts of the domain where changes are happening faster than elsewhere. ParalleX tries to provide an efficient runtime system for such algorithms and has been developed with consideration of recently introduced technologies that are expected to be essential ingredients in exascale computing, such as multi-core processors and accelerators.

ParalleX fosters the use of multithreaded processes with flexible synchronisation mechanisms. Its current implementation provides a means to use the concepts of an Active Global Address Space (AGAS), parcel transport and management, threads and thread management, and parallel processes.

The active global address space is used for the global unique naming of entities that can be moved between different localities and must be placed in the local address space. Localities could be arbitrary system partitions. The current implementation manages nodes as localities and uses a centralised server-client architecture to manage the identities of the global objects. Scalability problems of this service are avoided by caching.

The parcel transport and management provides active messages that can be sent to destinations in the global address space. The messages define an action to perform and its arguments plus a continuation – a list of local control objects that must be notified at the end of the action. The distributed control flow can therefore be adapted dynamically to the resources.

Threads also own a global unique name and can be managed from other localities. They serve as destinations for parcels that assign work to them. A user level scheduler, which implements cooperative multithreading to avoid the overhead of context switches, controls the thread execution.

Local control objects provide a variety of options to synchronise parallel activities. The options comprise, amongst others, mutexes and semaphores. Local control objects are also used to realise dataflows. Local control objects can be used to connect the outputs and inputs of subsequent calculations. It therefore becomes possible to avoid global barriers; calculations can start as soon as all prerequisites are fulfilled.

4.3.2 Suitability for Future Architectures

Future architectures will be characterised by more complex hardware than before. Another expectation is that more diversified hardware will be used. These trends put up the requirement that software needs to address more system specific aspects to gain performance. More complicated machine models have to be considered where previously simpler and more general assumptions were sufficient. It becomes more important to provide abstract interfaces to programmers that allow users to implement algorithms without too much need for hardware specific formulations. Furthermore, automated support for dynamic adaptation between hardware and software should be provided wherever possible.

The given analysis of the state of the art shows that there are many ongoing activities focusing on this issue. They provide valuable approaches and knowledge for future solutions. A general and comprehensive solution has not been developed so far. One reason is that many factors influence the execution of programs and partly conflicting measures must be optimised with respect to application and programmer efficiency.

4.3.3 Energy-Aware Approaches

As we move towards the exascale era, the energy-aware execution of applications is becoming a key consideration in HPC. The increasing size of machines and infrastructures are getting to a point where they will become unsustainable from an energy consumption point of view. Furthermore, this increase in power consumption leads to an increase in temperature, reducing the reliability of the system components. Therefore, runtime systems will be responsible not only for the efficient execution of applications from an execution time point of view but from their energy aspect as well.

There are two main approaches for power-aware techniques in HPC, Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Concurrency Throttling (DCT). On one hand, DVFS tries to adapt the CPU voltage and frequency in order to reduce power consumption while maintaining performance. On the other hand, DCT controls the number of threads being executed in multi-threaded codes to save energy and improve performance at the same time. For instance, if a code runs with four threads as fast as

with eight, the number of threads is reduced to four, saving then the energy of four cores.

DVFS is used by several state-of-the-art algorithms for power management control in order to reduce CPU frequency under waiting phases of the application [34][34][35][36]. For example, detecting critical CPU paths in an MPI program and slowing-down the CPU frequency under the non-critical ones. In that way, the computing phases are extended under MPI latency times, reducing the power consumed but without losing any performance. Other solutions use, for example, intra-node bottlenecks like memory latency to reduce the CPU frequency.

DCT algorithms [37][38], for instance, exploit performance limitations due to data-synchronisation and off-chip bus bandwidth to reduce the number of threads at runtime. All these techniques used in DVFS and DCT are based in heuristics and prediction models trained with benchmarks or with profiles from previous runs of the application.

These power-aware control techniques have only been implemented in some experimental runtimes for research purposes using programming models such as MPI, OpenMP or a hybrid MPI/OpenMP [39] but not in real production systems. Furthermore, they are far away from exascale. At most, they have been tested with applications running up to a hundred nodes.

Moreover, for some years GPUs have become an important trend in HPC in order to accelerate scientific applications. HPC is moving towards hybrid architectures composed of processors and accelerators such as GPUs. However, this option has been seen as a “non-green” solution due to the high energy consumption of accelerators. In fact, GPUs were firstly designed for desktops for which power consumption is not a problem. Several studies show that although the power consumption of GPUs is larger, the obtained acceleration results in a reduction of the overall power used by the application [39][40][41]. In [42] and [43] the authors show how the overall improvement in performance-per-watt using GPUs for four applications (NAMD, VMD, QMCPACK and MILC) is considerable. This performance-per-watt is calculated as $Power/Power_{accelerated}$*Speedup. However, this power saving is only achieved with speedups higher than 3x. If applications are accelerated but do not achieve a speedup of 3x, there is no power saving.

5 Performance Analysis Tools

HPC exascale systems will be composed of millions of homogeneous, or maybe even heterogeneous, processing elements. Running applications efficiently in such highly parallel and complex systems requires orchestrating different levels of concurrency (threads, message passing, I/O, etc.).

On one hand, it will be necessary to discover performance bottlenecks originating from the increase of complexity of each level of concurrency and to correct them in the application source codes. On the other hand, performance problems that originate from the use of shared hardware (network, file system, etc.) will not only affect a single application, it will disturb multiple applications running at the same time within the infrastructure. Observing this class of performance issues is fundamental since a minority of processes or processing elements can disturb the whole system. Therefore, performance monitoring of applications and the whole system will change from being a beneficial option to a necessity in such complex machines.

5.1 Performance Monitoring

5.1.1 Workload Monitoring

Workload analysis could help HPC managers to decide how to schedule or acquire resources in order to mitigate the contention between applications and hardware. For example, which parts of the infrastructure require an upgrade or what kind of new architecture increases system performance. The collected data could also help in the detection of contention between different applications. Thus, jobs with different complementary requirements could be scheduled together in order to increase the productivity of the HPC infrastructure. Furthermore, as auto-tuning of applications and libraries grows in importance, this system-wide approach could provide insights and feedback for performance decisions.

Approaches towards whole system workload performance monitoring already exist. However, there are not too many scalable tools available; Perfminer [49] is a system for the transparent collection, storage, and presentation of thread-level hardware performance data across an entire cluster. NERSC uses the Integrated Monitoring Network (IPM) [50] in order to collect 310,000 job profiles over the past 6 years. All these IPM performance profiles contain rich data sets to be explored on how machines are used and how different kinds of applications behave.

In addition, such a global lightweight profiling system could highlight applications running with performance problems easier. In addition, such global lightweight profiling will make the detection of poor performance applications possible. Afterwards, these applications could be further analysed for improvement with more specialised performance tools such as Vampir [63].

In brief, at exascale, performance will not only be dependent on the application under survey, but rather on the overall workload of applications. Monitoring of whole machines will become a necessity, but will introduce new challenges. Gathering and storing data from millions of processing elements at the same time will not be easily achievable. Furthermore, analysing all this data for visualisation will require scalable solutions. Automatic and online data mining methods and online analysis will be needed for this task

5.1.2 Application Monitoring

Running parallel applications efficiently on today's and future complex, highly parallel systems, with millions of concurrent processing elements, becomes more and more a challenge. Performance problems often only occur from a certain level of parallelism and due to the amount of concurrent processing elements (e.g. processes, threads) performance analysis – which should be part of the development cycle of an application – will be only possible by using appropriate and highly scalable performance analysis tools.

In the HPC community there are many established or experimental tools that assist programmers with performance analysis and optimisation of parallel target applications. Well-known examples are HPCToolKit [62], Jumpshot [67] Paraver [71], Periscope [63] Scalasca [66], TAU [65] and Vampir [64]. They focus on different aspects and provide complementary specialised features, but at the same time, there are many similarities and overlapping functionality. For instance, all tools have to cover the following parts of a performance analysis:

- Information generation
- Application monitoring
- Information processing and analysis

In the next sections each area is studied in detail and common advantages and drawbacks for exascale performance analysis are presented.

5.1.2.1 Information Generation

An important key point for a successful performance analysis is the information generation technique, to reduce intrusiveness of the monitoring technique and provide data to detect performance bottlenecks. Information can be generated by instrumentation, i.e. insert pieces of code into the application source code or binary. Or by using a sampling approach, i.e., observing the state of the application frequently.

The most prominent methods of instrumentation are:

- Compiler instrumentation inserts user-defined code snippets at the very beginning and end of each function;
- Source-to-source instrumentation transforms the original application and inserts code snippets at points/regions of interest;
- Library instrumentation intercepts public functions of an external shared library by using a `dlopen` interception mechanism;
- Binary instrumentation modifies the executable either at runtime or before program execution to insert code snippets at function entries and exits; and
- Manual instrumentation.

A big challenge for exascale performance analysis is the use of a suitable information generation mechanism, which may likely be a combination of mechanisms. This includes the choice between instrumentation and/or sampling, the selection of the right source code locations, the selection of the right time frame/duration, and the selection of the right processing elements. This will lead to the highest possible insight into the application and the overall system with the lowest possible intrusion. I.e., the performance analysis tools must be able to record only phases, events, and processing elements of interest with the best-suited information granularity.

An advantage of sampling is the ability to dynamically change the sampling frequency during measurement. However, it is challenging to define the optimal sampling frequency at a given time for a given processing element without any application knowledge. If the sampling frequency is too coarse-grained the number of samples and their total size will be easier to handle, but the root cause of a performance problem might not be detectable. In contrast, if the sampling frequency is too fine-grained, it will be possible to detect the cause of a performance problem, but this increases the intrusion significantly. Also, steering the sampling frequency of millions of concurrent monitoring processes during a performance measurement in order to reduce the measurement impact is challenging. Recently, some research has been focused on how to solve these challenges. In [52] coarse-grain sampling is combined with instrumentation information to obtain detailed performance information on iterative applications through a mechanism called folding. The folding gathers the metrics scattered all over the execution and creates synthetic regions that finely report the progression of the metrics.

Instrumentation as an information generation mechanism often results in detailed information, especially when tiny and often-used functions are instrumented, e.g., constructors and static class methods. However, for millions of processing elements this technique results in huge data amounts.

As a result, for plausible performance measurement on exascale systems, the performance measurement system should only record phases, events, and processing elements of interest. In [51][54][60] the authors use clustering techniques in order to detect application structure and application phases. In that way, performance tools can collect information only from relevant sections avoiding redundant data. In addition, combination of other techniques such as coarse-grained sampling and selective fine-granular instrumentation will become essential to overcome issues in intrusion, size and loss of information.

5.1.2.2 Monitoring

Basically, there are two main approaches to monitor the performance behaviour of parallel applications: profiling and tracing.

Event tracing is a well-established method for the performance analysis of highly parallel applications that records the measured information, i.e. each event, in detail. In particular, with tracing it is possible to identify outliers from the regular behaviour. Thus, it allows capturing the dynamic interaction between thousands of concurrent processing elements. As a result, tracing will produce an enormous amount of data that is challenging to handle. Tracing at an exascale level requires solutions to overcome this limitation.

In contrast to tracing, profiling aggregates the measurement information and generates statistics for the whole application run or for phases of interest. Flat profiles provide statistical information in a list style with various metrics such as inclusive runtime and number of invocations. For a more detailed analysis, in particular to analyse performance in the context of caller-callee relationships, call-path and call-graph profiles are scalable techniques to provide more insight into highly complex and parallel applications. Profiling, with its nature of summarisation, offers an opportunity to be extremely scalable, since information reduction is done over time. Afterwards, only the scalable reduction of all concurrent processing elements is needed, e.g. by using hierarchical reduction operations. Nevertheless, profiles may lack crucial information about message runtimes and bandwidth, since message matching is usually infeasible during profiling. Therefore, analysis of communication based performance issues is usually only possible by interpreting the aggregated time spent in the communication routines. One solution to overcome this issue is to use a piggybacking technique [61], which either modifies the messages by adding extra information or by using extra communication messages to exchange the message matching information.

5.1.2.3 Information Processing and Analysis

Nowadays, performance analysis techniques can be distinguished into online and offline techniques. Further, a classification into automatic and manual performance analysis exists. The former may detect performance bottlenecks and inefficiencies via a pattern analysis while the latter requires users to identify performance bottlenecks by interpreting the monitored information manually.

All performance analysis methods have advantages and disadvantages. The most important advantage of online performance analysis is that it processes and transfers only parts of the overall information. User interaction and tuning may influence this. However, online performance analysis techniques have to consider latency and bandwidth of the network, as well as the timing of the information processing, since users may expect real-time interaction during such an online performance analysis. In contrast, offline performance analysis techniques have to handle the amount of information of the whole measurement run and usually store this information in entire on the parallel file system. At an exascale level, creating one file per measured location (e.g., process or thread) results in disaster for current file systems. Current file systems can create only a few thousands files per second [69]. The bandwidth and capacity of

the entire memory hierarchy – including the file system – are also limiting factors for offline performance analysis, as long as no information reduction is done. Data mining and compression techniques, such as the data reduction on the Complete Call Graph (CCG) data structure [72] will become a necessity at exascale in order to perform on-the-fly information reduction. Moreover, data mining is useful for analysing the final reduced data as well. With data collected from millions of threads/processes, it will become difficult for a human to gain insight of the performance behaviour of their applications. Nowadays, several performance frameworks include data mining tools. For instance, PerfExplorer [56] allows parallel performance discovery and data mining through different runs or experiments of an application applying clustering, principal components (PCA) or regression.

Automated performance analysis processes the measured information either at runtime or in a post-processing step to provide a compressed data representation that only holds information about occurrences and causes of performance inefficiencies. Tools such as Periscope [63], Scalasca [66] or PerfExpert [57] offer automatic performance analysis. Furthermore, the use of data mining techniques has emerged as a solution for online automatic bottleneck detection. For instance, clustering different code regions can expose dissimilarities, communication bottlenecks or load imbalance [55].

In contrast, visual performance analysis highlights the original information and guides the user to the root cause of a performance bottleneck. This technique has to hold all information of a phase or of the whole measurement run available, which usually results in huge memory requirements for the analysis.

A combination of visual performance analysis and an online approach can reduce the amount of data to be transferred at a time. However, before such visualisation, the information of an entire time interval needs to be transferred and processed. For millions of concurrent processing elements this will become challenging. Also, visualisation of millions of different processing elements on a screen with a limited resolution is challenging. Exascale performance analysis will require solutions to reduce the amount of data that tools display. One such option is to only display “interesting” locations that behave differently, which requires an automatic approach for pre-processing.

5.2 Suitability for Future Architectures

5.2.1 Scalability on Exascale Computing Platforms

The main factors affecting scalability in the area of exascale performance analysis are performance and memory consumption. If the time taken to process the monitored information increases linearly (or even worse) with the amount of data, scalability will be limited. Also if the time taken to process the data increases linearly (or worse) with the number of analysis processes this indicates non-scalable processing techniques. If the memory consumption of the performance analysis increases linearly (or worse) with the amount of monitored information, this also indicates limited scalability. At small scale, memory consumption is often disregarded, but it becomes critical at large scale. The amount of memory that is available per core, which may be constant or even decreasing towards exascale, is a key concern, especially for long running applications, since fast analysis access is only possible when the monitored information will be held in the main memory.

The goal should be to use performance analysis techniques that require only constant space and time per monitored processing element or per interval. In recent years, several performance tools have made a lot of effort to solve these scalability issues. Tools such as Periscope [63] started to investigate the use of several distributed communication agents for online performance analysis. Each agent searches for bottlenecks in a subset of the application processes. TAUmon [58] and TAUoverMRNet (ToM) [59] investigate different distributed infrastructures such as MRNet [52] for online scalable performance monitoring. All these solutions are leading to a new trend in performance tools design, the allocation of additional resources for processing and analysing the performance data while the application is still running.

5.2.2 Resilience and Fault-tolerance

Since failures are expected to be common, due to the large number of components, the implementation of performance tools needs to be resilient and tolerant to faults. Fault tolerance may be needed from all levels of the stack – hardware, system software, and applications.

In the context of exascale computing, and in the performance analysis of millions of concurrent processing elements, the root cause of a performance problem may only occur at a single location. In addition, the possibility that one single thread of execution exits unexpectedly, due to a hardware defect, increases with the number of parallel elements. Therefore, performance analysis tools must be able to handle these situations without interruption of the on-going measurement and they must be able to process and analyse fragmented performance information.

One milestone for a successful exascale performance analysis is the investigation and development of resilient measurement and analysis techniques, which offer a degree of fault-tolerance. In this context it is reasonable to distinguish between

- Critical missing events and locations,
- Non-critical missing events and locations for exascale performance analysis.

For an automatic, replay-based performance analysis approach, events and locations are critical. A single piece of missing information may cause such an analysis to fail. Also, fragmented data with loss of the root cause of a performance problem can be classified as critical. All analysis tests will not be able to provide any insight for such data.

For a manual and offline performance analysis, missing events and locations are less critical, as long as they do not include the root cause of a performance problem. It is clear, that communication mapping will fail if a corresponding send or receive event is missing, but the analysis will still be possible. The absence of communication events results in a fragmented view of the detailed communication scheme and to incomplete communication statistics.

As long as the core performance problem is detectable, a fragmented view of the overall situation of an application will be one path to be able to analyse exascale applications. This is also one reason for using a coupled selective measurement model of instrumentation/sampling and profiling/tracing.

5.3 Recent Performance Analysis Tools and Future Research

Score-P [70] is a newly designed joint performance measurement runtime infrastructure for petascale performance monitoring for the tools Periscope, Scalasca, TAU, and Vampir developed by

- the Scalasca groups from Forschungszentrum Jülich, Germany and the German Research School for Simulation Sciences, Aachen, Germany,
- the Vampir group at Technische Universität Dresden, Germany,
- the Periscope group at Technische Universität München, Germany,
- the TAU group at University of Oregon, Eugene, USA,
- Computing Center at RWTH Aachen, Germany, and
- GNS mbH, Braunschweig, Germany.

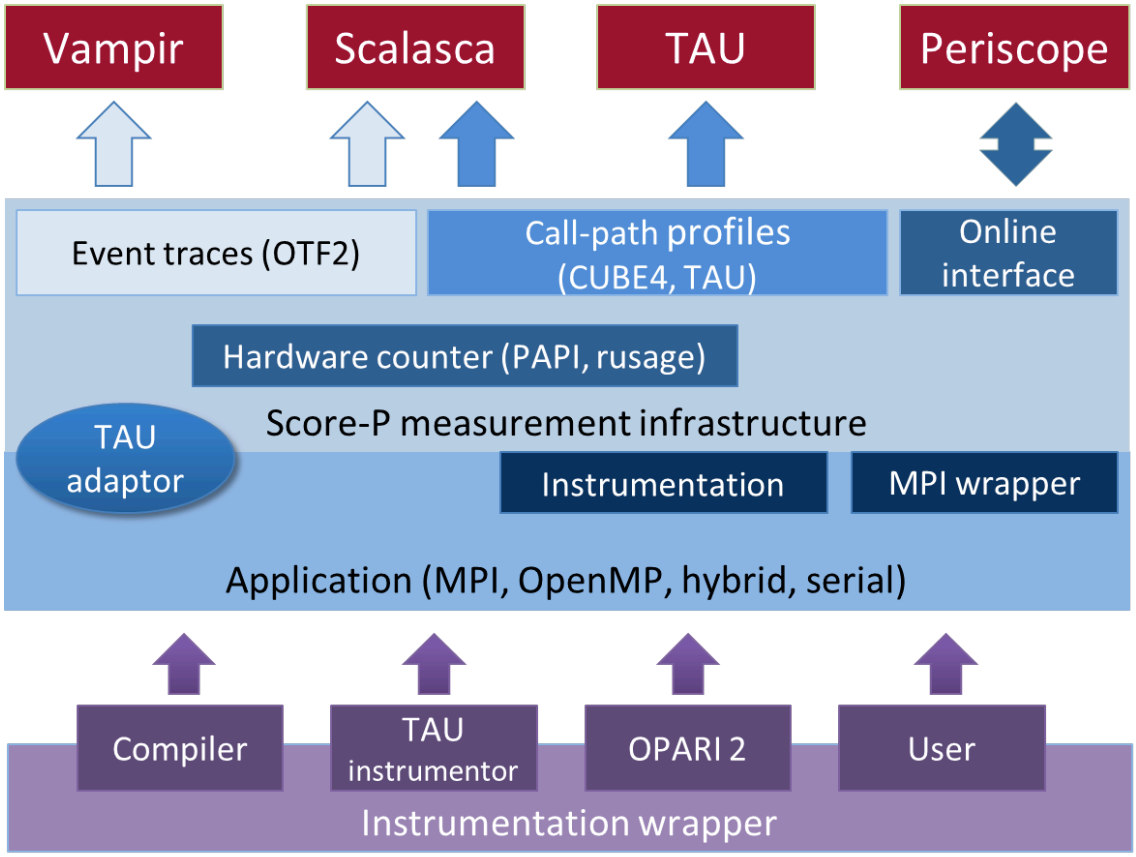

Figure 4: Overview of the performance measurement system Score-P.

In contrast to its predecessor measurement systems, VampirTrace and Scalasca, Score-P is the keystone for the monitoring techniques tracing and profiling with interfaces to various analysis tools, see Figure 4. It uses a flexible and efficient memory management system and a highly scalable tree-based unification operation at the very end of each measurement run. And is therefore ready for further improvements towards exascale performance analysis and will be the primary and reference measurement infrastructure within this project. At the moment, Score-P supports instrumentation of parallel MPI, OpenMP or hybrid combinations of these both. Extensions for Pthreads, PGAS and CUDA are planned in the near future. By default, Score-P runs in profiling mode but can also run in tracing mode.

One of the supported analysis tools of Score-P is Vampir, which is an interactive event trace visualisation software, which allows to analyse parallel applications with thousands of concurrent processing elements with various graphical representations in a post-mortem fashion. The scalability of Vampir is shown in Figure 5. This figure shows the visualised, colour-coded representation of 200,244 concurrent processes over a time of approximately 850s of the application S3D [68] running on the Jaguar partition, a Cray XT5, located at the ORNL Leadership Computing Facility.

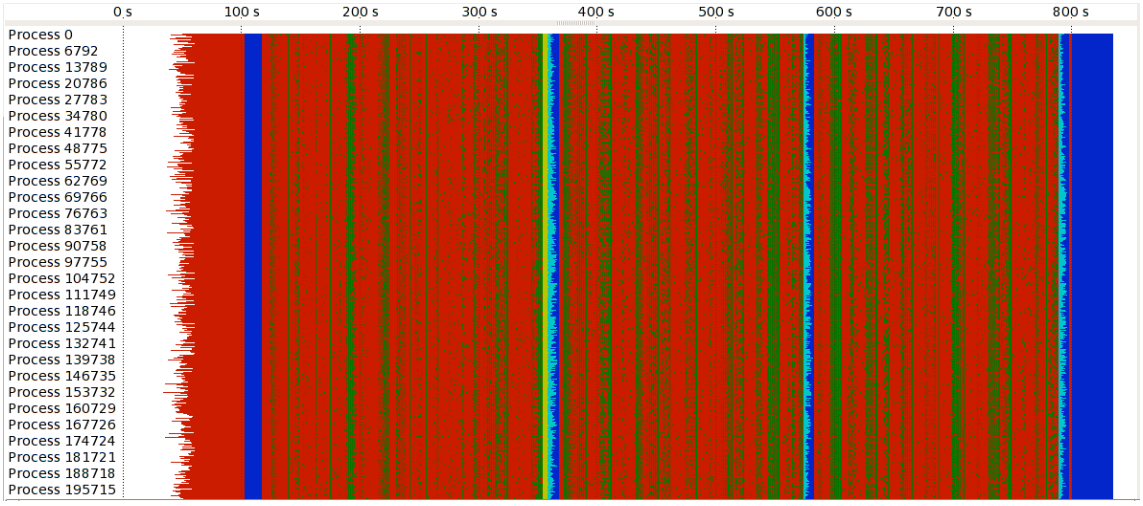

Figure 5: Color-coded visualisation of 200,244 processes over runtime of 850s of the application S3D [68] with Vampir [63].

Today, common performance monitoring and analysis tools will fail if a process or thread exits unexpectedly. Further, approaches fix the number of observed processes or threads at execution time. Thus, they do not support dynamic process creation and finalisation. Solutions for these limitations have to be developed in the future.

Also, performance analysis of an application and of the entire computing system is limited. At the moment it is possible to observe several performance metrics of the file system, the network, or of a hardware node, as long as there are interfaces that can be used by the monitoring system to make this information available in conjunction to the application behaviour. However, measuring all this information is no acceptable strategy to record full system behaviour, since resources for recording additional hardware information are limited. Therefore, it is advisable to record only hardware information, which provides an additional benefit for the performance analysis of a certain performance bottleneck. Unfortunately, common hardware information, especially information of shared hardware like network and file system, is often influenced by various factors and typically not only by one application. It is therefore laborious to determine the right set of information sources for a single application. Typically, a minority of applications can disturb the rest of the system and applications.

Another area that we have limited insight into is the performance of the network. Bottlenecks in the network hardware have serious implications for application performance and may be easier to detect given a network-centric view. Some networks support performance counters that can give insights into network behaviour so we intend to investigate if we could use such information (should it be available to monitoring frameworks) and present it in a way that provides benefit to the application developer.

Besides existing and often-used parallel programming models such as MPI and OpenMP, PGAS languages offer an opportunity to exchange data between distributed nodes within a global address space.

The question if PGAS, are suited to scale applications to exascale demands has to be researched in the near future. Therefore, performance tools must be able to record and analyse the communication and synchronisation behaviour. This can be done either by instrumentation of PGAS language calls/syntax (if possible) or by sampling the application. Depending on the vendor implementation these techniques have to be researched

6 Debuggers and Correctness Checking Tools

This section outlines the current state of the art for the process of debugging (the science of removing bugs from software) HPC applications. This contains the results and analysis of surveying some of the application developers within WP4, WP5, and WP6 in order to assess current perceptions of gaps and to understand requirements. The survey form is presented in Annex A.1. Further, we consider the suitability of current debuggers and error checking tools for future exascale computer systems.

6.1 Traditional Debuggers and Correctness Checking Tools

In the following we illustrate the state of the art for debugging in HPC. We first introduce approaches for parallel debuggers and approaches for runtime correctness checking afterwards. Finally, we illustrate different challenging aspects for these tools.

6.1.1 Parallel Debugging Tools

Currently, parallel debugging tools are developed by Allinea (Allinea DDT), and Rogue Wave Software (Totalview) and there are also at least two other less well used open source platforms - the Eclipse PTP project and the PERCS debugger (IBM). At this point, Allinea DDT is the most popular application in European HPC centres but Totalview is also used by at least two sites in CRESTA.

6.1.2 Correctness Checking Tools

Debuggers are frequently seen as observational tools - enabling the user to look into a program - but increasingly additional capabilities are built in. For example, memory debugging is a feature where the debugger understands more about the correctness of state and usage - rather than the traditional purely observational role. The two main commercial debuggers currently have memory debugging capabilities that include support for detection of beyond-bound memory accesses (read/write), or of memory leaks and common errors such as double de-allocations.

Besides these debugger extensions, runtime correctness tools exist that are specialised to certain paradigms. Examples of such are Helgrind [73] or the Intel Thread Checker [74] for OpenMP/Pthread paradigms and ISP [75], Umpire [76], MPI-Check [77], Marmot [78], and MUST [79] for MPI applications. These tools have built-in knowledge of a paradigm's semantics in order to check for wide ranges of paradigm-related usage errors. We collectively refer to these tools as correctness checkers. Correctness checks can include the detection of data-races for threading paradigms and communication errors such as type matching errors, detection of lost messages, and the detection of deadlocks for message passing applications.

6.1.3 Hardware Platform Support

In terms of platform support, debuggers presently support the typical mainstream HPC application developer platforms: from GPU workstations through to Petascale machines. Language support is provided for C, C++, Fortran in its major versions, along with Co-array Fortran and UPC. Accelerator support is found for NVIDIA CUDA and some of the languages for the CUDA architecture - but support is dependent on efforts of compiler authors, and on corresponding debugger efforts to support bespoke formats.

Correctness checkers are usually portable across wide ranges of machines, unless they are bound to a certain software/hardware stack, like Intel Thread Checker. However, some tools may have limitations with more specialised HPC systems. These tools also usually support the C, C++, and Fortran languages.

6.2 Current Debugger and Correctness Checking Tool Usage in CRESTA

Debugging of HPC software is an activity undertaken to remove a detected fault in an application. The presence of such faults can be detected at any point during software development, and often during deployment as a result of exposure to production environments that differ from the standard development environment - for example availability of larger machines, machines with different MPI implementations, different compilers or even different compiler flags.

Within the CRESTA applications groups, the largest job size on which a bug had ever occurred varied from under 100 processes through to many thousand (<10,000) processes. Application limits of scalability varied from under 100 cores through to 100,000 cores or more.

The currently practised methods of debugging noted by the applications showed variety - with commercial tools, open source tools, and ad-hoc `printf` tracing debugging all used. The majority of the centres have at least one commercial debugger licence available - and of those sites in CRESTA that did have such a tool, it was identified as being used regularly. It is worth noting that some users varied their own method, using different types of tools rather than just one.

Tools such as MPI checkers and thread checkers - for example Marmot and Intel Thread Checker were used by some sites, but not universally. Other open source tools from outside of the HPC specific domain are also often used - such as Valgrind, used for comprehensive memory debugging and its complementary Helgrind, for thread issues.

Although multiple techniques for resolving bugs were used, when asked how the largest job-size bug was fixed, every respondent had fixed it by using a debugger on the job at a smaller scale (usually), or the full scale (rarely). This shows that whilst print statements are effective at small scale, for success with a more challenging scale, the real debugger was used - and, even then, usually after simplification of the bug to a smaller scale. The existing MPI checking tools were not used - and this may be due to limitations of supported scale with these tools to date.

In recent times, the number of professionally organised development teams in HPC has increased and the discipline of software engineering is becoming more prevalent. It is worth observing the role of debugging within the development process: use of version control, unit tests, and continuous integration is becoming more commonplace. Within this environment, identifying the origins of software defects can be a much simplified process. Some sites used daily automated testing, with modest scale (eg. up to 10% of the expected job size) to ensure MPI capabilities were tested - others manually launched test-suites as and when required.

6.2.1 Perception of Debugging

Opinion was split as to whether debuggers currently were sufficient, or insufficient - although they identified that if tools were more integrated then they would likely use them more. Users did feel they have the time to explore new tools that would help their debugging.

The iterative nature of the debugging process is a challenge. In particular, having to re-run an application to narrow down causes is time consuming. Reverse debugging was seen as useful but needed to be scalable.

When asked what the current challenges of debugging at scale were, scalability of debugger performance was cited by the majority of users. The complexity of the debugger itself – i.e. understanding what the debugger was displaying and also interacting with it was a majority opinion. However, all respondents agreed, or strongly agreed, that the data set complexity at scale was a challenge.

6.2.2 Existing Bug Scenarios

When surveyed for typical kinds of bugs, a great deal of variety of experience was exposed. Problems with usage of the standard MPI library itself were common in some applications, although very rare in others. Reflecting the current architecture of typical HPC applications, where hybrid OpenMP/MPI is not the norm, OpenMP race issues occurred only occasionally.

The type of MPI errors noted included mismatched processes in MPI Communicators and MPI Collectives, or mismatched data types in MPI data transfer - for example custom MPI datatypes or mixing `MPI_INT` and `MPI_INTEGER`. A few of these errors can be detected at small scale by tools such as Marmot. The release of its successor MUST was too recent, and with that none of the developers applied this tool yet.

Existing typical (not HPC specific) bugs were also still prevalent - with developers experiencing memory leaks, memory overruns, use of uninitialised variables, or mismatches between software versions and the layout of datatypes between versions, all of which could lead to segmentation faults or incorrect results.

Some users had rarely or never experienced issues reaching higher scale, others occasionally. New environments - such as compilers, MPI libraries, driver versions (for accelerators) or machine configurations - had greater impact - with this cited as frequently an issue by multiple respondents. However, frequently a new environment will coincide with a new larger scale.

Unexplained differences between two versions of the same code, or two different machines running the same code happened frequently for some users. Given the vast scale of datasets - and, for example, different partitionings encountered in runs of different sizes - narrowing down such problems and identifying the first point of divergence is a significant challenge.

Another common problem was that incorrect data of one process “infected” the data of another process. Discovering the infection source of a variable/memory location on a process was identified as a hard task, as the data had come from a different process.

6.2.3 Desired Software Directions

Presently the overwhelming majority of HPC software is either MPI, OpenMP, GPU hybrid or a mixture of only two of the three - and uses C, C++ or Fortran.

As many-core becomes more pronounced, developers within CRESTA anticipated far less MPI-only code - with hybrid MPI becoming required instead. Some intended to consider PGAS languages as an alternative to MPI - without hybrids - for the expected simpler code-base and simpler support for overlapped communication and computation.

In terms of specific languages and models, the X10 and Chapel languages were not requested - but UPC, Co-array Fortran, StarSS, in addition to the current mainstay of C/C++/Fortran and MPI/OpenMP and CUDA or other accelerator solutions (OpenACC, OpenCL) featured.

Reliance on standard components featured prominently - with common component libraries such as BLAS, Boost, HDF5, Parmetis, NetCDF, PETSc and the C++ standard template library being part of many projects.

6.3 Suitability for Future Architectures

6.3.1 Scalability on Exascale Computing Platforms

In terms of scalability, currently only Allinea DDT is proven to Petascale - with the current production release being used at over 100,000 cores regularly at some national science sites in the USA, and modest 8,000-32,000 core sessions available in Europe. During acceptance testing at one site, Allinea DDT has reached full machine debugging for the largest available system at the time of 220,000 cores.

Currently common operations such as stepping every process at 220,000 cores took typically 100ms, with setting breakpoints in every process taking only 50ms or less. Job launch was shown to be no more than 10% higher when running with a debugger than running without, and application performance is unaffected. With performance at this scale, and a logarithmic tree network architecture, extending out by a factor of 10 would not be a challenge.

While debuggers can scale to 100,000 or more cores, correctness checkers suffer more from scale. Thread checking tools may impose extremely high overheads even for checking a few cores. MPI checking tools usually need to perform non-local analyses, such as message matching or MPI collective verification. These tasks are commonly run on a centralised instance and are a scalability bottleneck. With that, MPI correctness checkers currently scale up to about 1,000 cores, depending on the application.

6.3.2 Usability on Exascale Computing Platforms

Usability of the interface in a debugger at scale is a primary concern - also identified as an issue by the CRESTA applications. To this extent, Allinea's interface has a number of features that improve the complexity at scale. In particular they aim at making differences between processes simple. For example:

- Parallel stack views - scalable methods of examining the current location of processes
- Cross-process data comparison - with graphical display of variable variance over processes

Automatic correctness checking tools suffer less from this as they usually directly pinpoint to an error and can thus just present one instance of a detected error. However, in order to pinpoint an errors root-cause, it may be helpful to provide a good aggregation of error reports. With that tool users may understand regularities and patterns in correctness checking outputs more easily.

6.3.3 Support for New Programming Models

The main parallel debuggers have focused strongly on the models that are most popular: MPI and OpenMP. However, hybrid programming is supported by the commercial parallel debuggers - and Intel MIC is on the roadmap of both products. Given the focus on MPI or single-level parallelism, some of the techniques for simplification of data - such as comparing data across processes in a single one-dimensional MPI distribution (processes 0 to N) are not able to achieve the same for hybrid models where data is split between N processes each with M threads.

Task based parallel models are also less well supported - often runtime systems do not support querying the task queue and hence debuggers are powerless to display more information than the currently executing threads. Some work is presently being undertaken at HLRS in conjunction with Barcelona Supercomputing Centre on debugging the task-parallel StarSS. However, this is not integrated into a mainstream debugger yet. Other languages such as Intel's Cilk or OpenCL are also not covered.

Correctness checkers are extremely specific to the paradigm for which they were originally designed. As a result, only support for MPI, OpenMP, and basic support for mixes of the two paradigms are available. No approach for OpenCL, CUDA, any PGAS languages, StarSS, or Cilk is known to us, besides basic compiler based error detection capabilities [80].

7 Gap Analysis and Conclusions

The previous chapters of this document examined the state of the art and the new challenges in implementing development environments for current and future computing platforms. In the following sections, we analyse the needs and gaps to be filled towards the development of software tools for exascale computing. This chapter provides the directions for the future work in the CRESTA WP3. Different sections focus on the WP3 subtask topics: programming models, compilation and run-time environments, performance and debugging tools.

7.1 Programming Models

It is clear from the PRACE statistics (discussed above) that the vast majority of HPC applications are written using "traditional" HPC programming models, as defined in Section 3.1. The reasons for this are beyond the remit of this report, but include:

- The time required for a complete rewrite, especially if current performance is satisfactory.
- Lack of expertise, or access to expertise, in new languages
- Unproven performance for new programming models
- Limited vendor support restricting portability and risking obsolescence.
- Lack of supporting tools (particularly debuggers and performance analysis tools) limiting productivity in the new programming model.

Given this, it is important to establish to what extent the choice of programming model is and, more importantly, will become the limit to good scaling of an application, and whether this can be avoided through, for instance, new algorithms or replacing user code with third-party libraries (this is addressed in CRESTA Deliverable D4.1.1 "Overview of major limiting factors in existing algorithms and libraries").

If the programming model truly is the obstacle, this may be attributable to features missing from that programming model (e.g. the current lack of CAF collectives in the Fortran standard). Representation on, and participation in, the relevant standards committees is therefore very important for major projects like CRESTA.

If, however, a new programming model is needed to reach the exascale, developers will tend to prefer an incremental approach to a full rewrite in a new language, which will favour moves to programming models that interoperate well with the current communications model (which, as we have seen, is probably MPI). Moving from pure MPI to a hybrid MPI/OpenMP code is generally seen as being relatively straightforward and there is an increasing number of tools to help developers do this. Both models of parallelism are open standards and widely supported, both individually and when used together. Standardising interoperability of programming and parallelisation models is difficult, as each tends to be developed in isolation. Clear statements from vendors as to the interoperability that users can expect from their platforms are very desirable in the first instance. Moving forward, it would be useful if the various standards committees could develop some form of "memoranda of understanding" that describe the expected interoperability, and the form that this takes. These might provide a guide for compiler and library developers in an attempt to maintain interoperability wherever possible. Given the limitations to MPI interoperability (e.g. regarding the use of subcommunicators or derived datatypes), there may also be a need for additional tools to help users when introducing, for instance, PGAS features into their existing code. These might take the form of libraries that convert a given MPI call into a set of operations in the new programming model. It is unreasonable to expect such tools to be specified in any of the ratified standards, so these will probably be developed by interested user groups.

C++ has some particular needs. It is clear from the PRACE usage analysis that it already has a significant presence in HPC and in the CRESTA co-design applications in particular. There is, however, a tension between the productivity gains of using the high-level features of C++ and the need for good performance. This gap can be

bridged partially through the continued development of highly optimising compilers (and associated programming environments and tools) and runtime systems for C++. This is in part driven by the availability of representative C++ benchmark codes that can be used by the developers of these systems and the CRESTA benchmark suite will be a very valuable addition here, and should be widely publicised. In the meantime (at least), there is also a simultaneous need for documentation describing "best practice", Not only does this help guide users (in, for instance, the use of templating to improve cache usage), it will also provide a focus for compiler and runtime development.

We may also see C++ being used in a similar fashion to python in HPC, dealing with system management, I/O and configuration, but calling lower-level Fortran and/or C for the performance-critical kernels. Again, there is a need for documentation on "best practice" and "use cases", in parallel with a drive to easier and more efficient interoperability between these programming models.

Users of new programming models will also need help. Profiling and debugging tools are now supporting, or moving to support, PGAS languages and we expect this to improve greatly through the work of CRESTA, despite the difficulties in instrumenting codes using these lightweight communication models.

Novel programming models, such as Chapel, suffer from all the above limitations, perceived or otherwise, and have so far had a very limited uptake. It is an important task to understand and quantify the disruptive benefits of moving wholesale to such a framework. Also key is, where possible, to improve the interoperability with the traditional models. It is perhaps significant that many application developers have been willing to port to new programming models such as CUDA despite all of the same problems, largely because of the advertised performance gains of using GPUs and because they can start porting one kernel at a time.

Accelerator directives provide a mechanism for bridging the gap between applications written for the CPU and those ported explicitly to the GPU, offering interoperability both with the traditional CPU languages and *de facto* GPU standards like CUDA. They are likely to become increasingly popular as the distinction between the two is eroded (e.g. the slow PCIe link between the distinct memory spaces of CPU and GPU). The challenges, however, are to deliver an acceptable level of performance using this high-level approach, as well as to provide the support profiling and debugging tools. Moving forward, the current OpenACC directives, for instance, are unlikely to provide the full functionality required for an application to fully utilise a heterogeneous node, and there is an important task during the CRESTA project to ensure that the nascent standards, e.g. as part of OpenMP, evolve in the most productive way for HPC users of heterogeneous architectures as we approach the exascale.

7.2 Compilation, Autotuning and Runtime Systems

Concerning compiler tuning, there has been promising work in the area of compilers for serial and small-scale parallel kernels. The challenge is in extending these techniques to large-scale distributed memory applications such as those in the CRESTA benchmark suite. This study needs to be done on the current generation of petascale machines in order to select the approaches that may prove successful at the exascale.

Concerning autotuning in general, there is no one approach that covers all aspects of application tuning, there is no consistent way to mark up choices and expose those to an autotuning framework and autotuning is not something in the mind of the average HPC application developer.

Our goal is to define such a consistent approach that can target mark-up of choices at all the stages mentioned (algorithm choice, source, compilation and launch) and hopefully in a way that is not too onerous far the application developer. In particular we want to pay attention to important parallel tuning aspects (stencils, communication patterns etc.). We would also support definition of independent tuning spaces, plugin tuners for specific aspects and feedback mechanisms required for those tuning frameworks that can optimise based on execution metrics (for example performance

counter metrics) beyond the primary optimisation metric. The ideal situation of the future would be one where the application developer thinks of exposing such choices as part of application development. The development of the DSL will be informed by the co-design applications and availability and capability of open source autotuners. We expect to do some limited integration with one of these autotuners by the end of the project.

The analysis of the state of the art of runtime systems shows that there are many ongoing activities in this field. A general and satisfactory solution could not be developed so far due to the many factors influencing the execution of programs and the partly conflicting measures for optimisation and programmer productivity. However, there is a common set of challenges the development of exascale runtime systems faces. The deeper memory hierarchies and the relative reduced availability of certain resources per core, like memory size or communication bandwidth, are the most important challenges. Runtime systems need to use a much higher degree of parallelism as well as to adapt programs better to the system on that they are running. The dynamic adaption of the runtime system software becomes more important because of the gap between the computational capability and hardware.

7.3 Performance Analysis Tools

To elevate performance monitoring and analysis tools from petaflop to exaflop scale and therefore be able to monitor highly parallel applications in combination with the whole machine and its entire infrastructure (network, memory hierarchy) requires us either to develop new methods for monitoring and analysis of information or to combine existing methods to reduce the drawback of each single method for a better insight into the system and application by lowest possible intrusion. This means, that we should monitor not everything but rather to develop scalable strategies to selectively monitor systems and applications and to use analysis strategies that helps to identify outliers and provide sufficient insights into applications and systems behavior.

It is important to bridge the gap between coarse-grained profiling, which is to be used where sufficient, and fine grained event trace information, where necessary. Furthermore, data mining and reduction techniques will become a necessity in exascale in order to perform the on-the-fly information reduction that will be a requirement to deliver scalable, automated online performance analysis.

These strategies will lead to the highest possible insight into the application and the overall system with the lowest possible intrusion. I.e., the performance analysis tools must be able to record only phases, events, and processing elements of interest with the best-suited information granularity.

7.4 Debuggers and Correctness Checking Tools

Current software debugging practice and requirements gathered from the application developers showed a number of areas that were either needed already, or would become more vital as scale of applications increases.

- Automatic identification of anomalous values is becoming important: the volume of data is increasing as applications grow and is already unmanageable. Identifying earlier that a value is invalid would be helpful even at current application scale. Identification of the source-process of defect values is a capability that would be highly useful and is not available in any existing debugger.
- Identifying anomalous application activity is also important - current approaches, for example, viewing of the merged stacks of processes are helpful but need to be extended. Identifying, for example, the path of execution that led to a particular issue would be helpful.
- MPI usage errors remain important - and having MPI correctness checkers that can reach higher scales than 100-1,000 cores is crucial. Some MPI usage errors should be detected instantly by debuggers,

such as misused MPI parameters. Further, debuggers should provide more information about MPI types such as communicators where possible.

In terms of the directions of applications within CRESTA, many users were considering other established compilers or models - for example PGAS languages, or hybrid solutions - and emerging compilers and models (for example StarSS). Whilst the PGAS/hybrid solutions have support, emerging models are less well supported. The IESP roadmap expects evolution in the programming models - from 2012 through to 2015 before stability.

- As there is uncertainty in programming models beyond 2013, and debugging support is at risk and uncertain.

The trend in systems and applications towards hybrid architectures has implications for the type of errors likely to be encountered - and there are presently a number of gaps:

- Thread checkers - whilst such tools do exist, their slow-down increases as the number of cores per node increases; Overheads will become more noticeable as a result;
- Tools for GPU error detection - at this point there is no established framework for detecting errors automatically - for example, the premature use of returned data from the GPU or incorrect behaviour within a GPU kernel;
- There are currently no tools to automatically identify errors within UPC or Co-array Fortran codes - nor any static code checking tools.

Extending correctness checkers for these paradigms is very desirable, but will also be challenging as up-to today all correctness checkers where specialised to one paradigm, while they still lacked many desirable extensions in scalability and richness of checks.

Each of these tools (either postulated or already existing) on its own is not sufficient to give the required level of information to fully fix a bug. By integrating such tools within a debugger, the developer of the tools will have access to a complete environment in which the context of an application can be understood. Whilst a tool can prove there is a problem, the user must deduce how the problem occurs.

- Tool integration is lacking for many correctness checkers - some previous work with Allinea and HLRS enabled the use of Allinea DDT and the Marmot plugin for application MPI correctness checking, which also worked with the Intel MPI checker. However, the framework was applicable for small scale and it requires rework to support higher degrees of concurrency.

Deep large software stacks will require separation of concerns - and debugging appropriate to the development layer. By this we mean the ability to support debugging at the level being used by the application developer. The application developer will trust the libraries and layers on which his application is built. Assertions and verification of the lower levels becomes more important in this scenario - and libraries or programming will need to open up access to, and work with, debuggers such that the trust can be maintained - and that internal verification will pick up misuse.

As users move their code through to more hybrid architectures we should examine how we can support migration efforts.

As we have identified an increasing degree of professionalism in software development, with automation in testing or universal usage of version control, we should identify how debuggers can fit in to the workflow of developers. Specifically:

- Test case integration;
- Version control and identifying potentially significant recent changes; and

- Automation of fault finding - being able to iteratively run through a sequence of tests until the fault is found.
- The IESP identifies that there is overlap in the concerns of performance and debugging - with low performance often being recognised as a bug, and one that can result from external factors. Fault tolerance is expected to be a major challenge for exascale. Hardware failure is likely to occur more frequently - and will be responsible for software failure - or failure of software performance.
- There is a lack of integration of performance and debugging, but also of hardware information in debugging session.
- Resilience of the debugger to node failure is currently not provided - and the current MPI implementations and debugging APIs do not handle failure.
- Existing performance tools provide a good view of resource usage within a code but users could benefit from seeing the performance in the context of a running application to understand a performance issue - although debugging can impact application performance and research into whether the two objectives can coincide is required.

We make the following recommendations, which are a subset of the identified gaps:

- **Tools Integration in Scalable Framework:** Tool integration should be developed. Allinea's tools represent a scalable portable platform that will be enhanced to reach exascale and this platform can be opened as a way to provide a lower barrier to entry for other tools developers, removing the hard problems of scalability and portability and allowing them to concentrate on their strengths. Allinea will work with TU Dresden to analyze ways to enable the MUST (successor of Marmot and Umpire) MPI checker to work within the debugger to demonstrate this platform.

- **Support for New Programming Models:** As other elements of CRESTA will investigate programming models, the impact of debuggability is important - and an alternative model will be identified and debugging support for this will be considered with a view to discovering how such models will be debugged. There is currently a gap in correctness checking support for PGAS - e.g., UPC and Co-array Fortran - OpenCL, CUDA, and the Intel-MIC languages. Any newly developed model or compiler needs to consider its debugging strategy. This may involve:
  - Use of an API to enable debugger/runtime interaction;
  - Correctness checkers must intercept any use of the paradigm/model as some form of an event.
  - Specific library hooks to query consistency of internal data types;
  - Specific library code or support for representing the opaque data at a higher level.

- **MPI Correctness**: While an interesting paradigm/model for an extension of existing correctness checkers should be identified, MPI remains a primary interest for application developers. As a result, we will extend scalability of MUST to cope with more than 1,000 processes. This involves extensions for runtime message matching and deadlock detection. We will try to extend towards at least one additional programming model, the decision of the model is highly dependent on the application developers interest and the availability of a instrumentation/measurement framework for the model.

- **Clustered Anomaly Detection**: Debugging is both deductive and iterative, and yet iteration is not a process that we humans do well. At

current scale, and as we reach higher scales, we can automatically identify anomalies that happen - differences with previous successful runs, and with processes that are successful within the current task. This could cover both data changes, and process activity. Automated methods for asserting data integrity should be investigated that would allow, for example, a developer to efficiently detect incorrect values. This could involve developing both standard libraries for data verification, and model specific libraries.

- **Application/Library Model Awareness**: Better integration of layered models and the debugger should be investigated - with, for example, awareness of MPI communicators and the internals of request object or integration with runtime of task based parallel frameworks to visualise internal task lists.

- **Fault Tolerance:** We will closely investigate fault tolerance problems during the CRESTA timeframe to support WP2. Further, we will try to evaluate mechanisms from WP2 on a prototype/experiment level. However, no product-level actions for fault tolerance or debugger resilience in the debugging context is expected to be undertaken during CRESTA.

8 References

[1] I.D. Chivers and J. Sleightholme, "Fortran Resources", http://www.fortranplus.co.uk/resources/fortran_resources.pdf (accessed January 2012)

[2] B.W. Kernighan and D.M. Ritchie, "The C Programming Language" (1st ed., February 1978), Englewood Cliffs, NJ: Prentice Hall. ISBN 0-13-110163-3.

[3] OpenMP web site, http://www.openmp.org (accessed December 2011; "Using OpenMP: Portable Shared Memory Parallel Programming" Chapman, Jost and Van der Pas, MIT Press, ISBN: 0262533022

[4] Message Passing Interface Forum, http://www.mpi-forum.org (accessed December 2011)

[5] J. Levesque and G. Wagenbreth, "High Performance Computing: Programming and Applications", Chapman & Hall (2010), ISBN: 978-1-4200-7705-6.

[6] PRACE project home page, http://www.prace-project.eu (accessed December 2011)

[7] PRACE-1IP Deliverable 7.4.1: Applications and user requirements for Tier-0 systems, Mark Bull (EPCC), Xu Guo (EPCC), Ioannis Liabotis (GRNET), http://www.prace-ri.eu/PRACE-1IP-Public-Deliverables

[8] Co-array Fortran web page, http://www.co-array.org/ (accessed December 2011)

[9] UPC Language Specification (Version 1.2), http://upc.gwu.edu/docs/upc_specs_1.2.pdf

[10] OpenSHMEM Project, http://www.openshmem.org (accessed January 2012).

[11] X10 Language home page, http://x10-lang.org (accessed December 2011)

[12] Project Fortress home page, http://projectfortress.java.net/ (accessed December 2011)

[13] The Chapel Programming Language, http://chapel.cray.com (accessed January 2012)

[14] Top 500 Supercomputer Rankings web site http://www.top500.org

[15] NVIDIA CUDA C Programming Guide, version 3.2 (2010), available from: http://developer.nvidia.com/cuda-toolkit-32-downloads

[16] PGI CUDA Fortran home page http://www.pgroup.com/resources/cudafortran.htm (accessed December 2011)

[17] OpenCL home page, http://www.khronos.org/opencl (accessed December 2011)

[18] Portland Group PGI Accelerator compilers, http://www.pgroup.com/resources/accel.htm (accessed January 2012)

[19] CAPS Entreprise http://www.caps-entreprise.com/hmpp.html (accessed January 2012)

[20] OpenACC standard, http://www.openacc-standard.org (accessed January 2012)

[21] R.C. Whaley, R. Clint and A. Petitet, "Minimising development and maintenance costs in supporting persistently optimised BLAS", Software Practice and Experience (2005)

[22] F. Mateo and S. G. Johnson, "The design and implementation of FFTW3", Proceedings of the IEEE special issue on Program Generation, Optimization, and Platform Adaptation (2005)

[23] M. Pischelet *et al.*, "SPIRAL: Code Generation for DSP Transforms", Proceedings of the IEEE special issue on Program Generation, Optimization, and Platform Adaptation (2005)

[24] A. Tiwari *et al.*, "A scalable autotuning framework for compiler optimization", Proceedings of the 24th International Parallel and Distributed Proecessing Symposium (2009)

[25] Collective Tuning Project http://ctuning.org/wiki/index.php/Main_Page (accessed January 2012)

[26] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs.", SC10 Proceedings (2010).

[27] J. Ansel *et al.* “An Efficient Evolutionary Algorithm for Solving Bottom Up Problems”, Annual Conference on Genetic and Evolutionary Computation (2011)
[28] OP2 website, http://people.maths.ox.ac.uk/gilesm/op2/ (accessed January 2012)
[29] C. Augonnet, S. Thibaul, R. Namyst, “StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines”, (2010)
[30] StarSS homepage, http://www.bsc.es/computer-sciences/programming-models (accessed January 2012)
[31] F. Broquedis, N. Furmento B. Goglin, P.A. Wacrenier and Raymond Namys, “ForestGOMP: An Efficient OpenMP Environment for NUMA Architecture”, International Journal of Parallel Programming (2010)
[32] L.V. Kale, E. Bohm, C.L. Mendes, T. Wilmarth and G. Zheng, “Programming petascale applications with Charm++ and AMPI”, Petascale Computing: Algorithms and Applications (2008)
[33] A.Tabbal, M. Anderson, M. Brodowicz, H. Kaise and T. L. Sterling, “Preliminary design examination of the ParalleX system from a software and hardware perspective”, SIGMETRICS Performance Evaluation Review (2011)
[34] N. Kappiah, Vi. W. Freeh, and D. K. Lowenthal, “Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs”, Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC '05) (2005)
[35] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal, “Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs”, Supercomputing (2006)
[36] B. Rountree, D. K. Lowenthal, S. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz. “Bounding energy consumption in large-scale MPI programs”, Supercomputing (2007)
[37] M. Curtis-Maury, J. Dzierwa, C.D. Antonopoulos, and D.S. Nikolopoulos. 2006. “Online power-performance adaptation of multithreaded programs using hardware event-based prediction”, Proceedings of the 20th annual international conference on Supercomputing (ICS '06) (2006)
[38] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, “Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs”, Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (2008)
[39] Dong Li, B.R. de Supinski, M. Schulz, K. Cameron and D.S. Nikolopoulos, "Hybrid MPI/OpenMP power-aware computing", Parallel & Distributed Processing (IPDPS) (2010)
[40] S. Huang, S. Xiao, W. Feng, "On the energy efficiency of graphics processing units for scientific computing," Parallel & Distributed Processing, 2009. IPDPS 2009 (2009)
[41] M. Rofouei, T. Stathopoulos, S.Ryffel, W. Kaiser and M. Sarrafzadeh “Energy-aware high performance computing with graphic processing units”, Proceedings of the 2008 conference on Power aware computing and systems (2008)
[42] J. Enos, C. Steffen, J.Fullop, M.Showerman, S.Guochun, K. Esler, V. Kindratenko, J.E. Stone and J.C. Phillips, "Quantifying the impact of GPUs on performance and energy efficiency in HPC clusters," Green Computing Conference *2010* (2010)
[43] T. Udagawa and M. Sekijima, "The Power Efficiency of GPUs in Multi Nodes Environment with Molecular Dynamics" Parallel Processing Workshops *(ICPPW) (2011)*

[44] Z. Wang, Z. and M. O'Boyle “Mapping Parallelism to Multi-cores: A Machine Learning Based Approach”, 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2009)
[45] G. Tournavatis et al. “Towards a Holistic Approach to Auto-Parallelization: Integrating Profile-Driven Parallelism Detection and Machine-Learning Based

Mapping", Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (2009)
[46] A. Tiwari *et al*., "Auto-tuning full applications: a case study". International Journal of High Performance Computing Applications (2011)
[47] FTRANS User Guide by S. Booth, ftp://ftp.epcc.ed.ac.uk/pub/personal/spb/ftrans/ (accessed January 2012)
[48] Cray Fortran Reference Manual S-3901-74. Section 4.7 PGAS Directive. http://docs.cray.com/books/S-3901-74/ (accessed January 2012)
[49] P. Mucci, D. Ahlin, J. Danielsson, P. Ekman and L. Malinowski, "PerfMiner: Cluster-Wide Collection, Storage and Presentation of Application Level Hardware Performance Data", Euro-Par 2005 Parallel Processing (2005)
[50] K. Fuerlinger, N.J. Wright, D.Skinner, "Effective Performance Measurement at Petascale Using IPM", Parallel and Distributed Systems (ICPADS) (2010)
[51] J. Gonzalez, J. Gimenez and J. Labarta, "Automatic detection of parallel applications computation phases", Parallel & Distributed Processing 2009 IPDPS 2009 (2009)
[52] H. Servat, G. Llort, J. Giménez and J. Labarta, "Detailed Performance Analysis Using Coarse Grain Sampling", EURO-PAR 2009 - PARALLEL PROCESSING WORKSHOPS (2010)
[53] P. C. Roth, D. C. Arnold, and B. P. Miller. "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools", SC 2003, (2003)
[54] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior", Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS-X) (2002)
[55] X. Liu, J. Zhan, K. Zhan, W. Shi, L. Yuan, D. Meng, L. Wang, "Automatic performance debugging of SPMD-style parallel programs", Journal of Parallel and Distributed Computing (2011)
[56] K. A. Huck and A. D. Malony, "PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing", Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC '05) (2005)
[57] M. Burtscher, B.D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, "PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications", Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10) (2010)
[58] A. Nataraj, M.Sottile, A. Morris, A. Malony, S. Shende, "TAUoverSupermon: Low-Overhead Online Parallel Performance Monitoring", Euro-Par 2007 Parallel Processing (2007)
[59] A. Nataray, A.D. Malony, A. Morris, D.C. Arnold and B.P.Miller, "A framework for scalable, parallel performance monitoring", Concurrency and Computation: Practice and Experience (2010)
[60] G. Llort, J. Gonzalez, H. Servat, J. Gimenez and J. Labarta, "On-line detection of large-scale parallel application's structure", Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium (2010)
[61] M. Schulz, G. Bronevetsky and B. R. Supinski, "On the Performance of Transparent MPI Piggyback Messages", Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing (2008)
[62] L. Adhianto *et al*., "HPCtoolkit: tools for performance analysis of optimized parallel programs", Concurrency and Computation: Practice and Experience (2010)
[63] S. Benedict, V. Petkov and M. Gerndt, "Periscope:An online-based distributed performance analysis tool, Tools for High Performance Computing 2009 (2010)
[64] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M.S Müller and W.E Nagel, "The Vampir Performance Analysis Tool Set", Tools for High Performance Computing 2007 (2008)
[65] S. Shende and A.D. Malony, "The TAU Parallel Performance System", International Journal of High Performance Computing Applications (2006)

[66] M. Geimer, F. Wolf, B.J Wylie, E. Ábrahám, D. Becker and B. Mohr, “The Scalasca Performance Toolset Architecture”, Concurrency and Computation: Practice and Experience (2010)
[67] C.E. Wu, A. Bolmarcich, M. Snir, D. Wootton, F. Parpia, A. Chan, E. Lusk and W. Gropp, “From Trace Generation to Visualization: A Performance Framework for Distributed Parallel Systems“, Proc. of SC2000: High Performance Networking and Computing (2000)
[68] J.H Chen et al.,“Terascale direct numerical simulations of turbulent combustion using S3D“, Computational Science & Discovery (2009)
[69] S. R. Alam, H. N. El-Harake, K.R Howard, N. Stringfellow and F. Verzelloni, “Parallel I/O and the Metadata Wall“, Parallel Data Storage Workshop (PDSW’11) (2011)
[70] A. Knüpfer et al., “Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir“, Tools for High Performance Computing, Proceedings of 5th International Workshop on Parallel Tools for High Performance Computing 2011, (to be published)
[71] J. Labarta, J. Gimenez, E. Martínez, P. González, S. Harald, G. Llort and X. Aguilar: “Scalability of Tracing and Visualization Tools”, Proceedings of the International Conference ParCo 2005 (2005)
[72] A. Knüpfer and W. E Nagel, “Compressible memory data structures for event-based trace analysis”, Future Generation Computer Systems (2006), ISSN: 0167-739X
[73] A. Müehlenfeld and F. Wotawa, “Fault detection in multi-threaded C++ server applications”, Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '07)(2007)
[74] Intel Corporation, “Intel Thread Checker”, http://www.intel.com/support/performancetools/threadchecker (accessed February 2012)
[75] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: A Tool for Model Checking MPI Programs", Proc. PPOPP (2008)
[76] J. S. Vetter and B. R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," Supercomputing, (2000)
[77] G. R. Luecke, H. Chen, J. Coyle, J. Hoekstra, M. Kraeva, and Y. Zou, "MPI-CHECK: a tool for checking Fortran 90 MPI programs", Concurrency and Computation: Practice and Experience (2003)
[78] B. Krammer and M. S. Müller, "MPI Application Development with MARMOT", in Proc. PARCO (2005)
[79] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller, "MUST: A Scalable Approach to Runtime Error Detection in MPI Programs", Proc. Tools for High Performance Computing, Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing (2009)
[80] G. R. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Xu, E. Kleiman, and O. Weiss. “Evaluating error detection capabilities of UPC run-time systems”, Proceedings of the Third Conference on Partitioned Global Address Space Programing Models (PGAS '09) (2009)

Annexes

A.1 Survey Form

The following form has been completed by WP4, WP5 and WP6 partners to identify first the common practices among the co-design application experts, and second the priority in the development of environment tools.

What main tools/functions do you currently use to debug your applications?

	Never	Occasionally	Mostly	Always
Allinea DDT	○	○	○	○
Command line tools such as GDB, DBX or IDB	○	○	○	○
Printf/write statements inserted in code	○	○	○	○
Totalview	○	○	○	○
Valgrind	○	○	○	○

Do you use or have you used any automatic correctness tools?

- ☐ Helgrind - thread checking within the valgrind framework
- ☐ Intel Thread Checker
- ☐ Intel Message Checker (Now within the Intel Trace Analyzer)
- ☐ ISP (University of Utah)
- ☐ Marmot (HLRS/TU Dresden)
- ☐ MPI-Check (University of Iowa)
- ☐ MUST (TU Dresden)
- ☐ UMPIRE (LLNL)
- ☐ Other: ☐

Did you use any other automatic correctness tools that is not specialized for OpenMP/Pthreads or MPI? If so, which one?☐

Did your experience with any non MPI/OpenMP/Pthread paradigm yield the wish for automatic correctness checks? If so, what types of checks would you like for which paradigm?☐

	Strongly Disagree	Disagree	No opinion	Agree	Strongly Agree

	Strongly Disagree	Disagree	No opinion	Agree	Strongly Agree
Debuggers currently do everything I need	○	○	○	○	○
I don't have time to learn how to use tools	○	○	○	○	○
I don't have time to discover new tools	○	○	○	○	○
If tools worked together, I would use them more	○	○	○	○	○

What kind of software errors do you experience

	Never	Rarely	Occasionally	Frequently
Reaching new scale	○	○	○	○
New environment (eg. compiler change or different MPI)	○	○	○	○
Accelerator specific problems	○	○	○	○
MPI race conditions (random message orderings)	○	○	○	○
MPI deadlock	○	○	○	○
MPI data transfer bugs - eg. incorrect halo exchanges	○	○	○	○
OpenMP/multi-threading data races	○	○	○	○
Unexplained difference in output between two versions/runs of same code	○	○	○	○

Describe the most frequent cause/type of software bug that you have?

Techniques used to debugIf you wish to, outline some of the ways that you use to debug that you feel are not currently done for you by debuggers

About your software development

Please give an overview of how your primary application is structured and the technologies it uses, and how it is developed - or will be developed over the next few years

Currently used parallel structure

- ☐ MPI
- ☐ MPI + GPU
- ☐ MPI + GPU + OpenMP/multithreaded
- ☐ MPI + OpenMP/multithreaded
- ☐ OpenMP
- ☐ PGAS - Coarray Fortran, UPC, ..
- ☐ Other: ☐

Anticipated future application structure

- ☐ MPI
- ☐ MPI + GPU/MIC
- ☐ MPI + GPU/MIC + OpenMP/multithreaded
- ☐ MPI + OpenMP/multithreaded
- ☐ OpenMP
- ☐ PGAS - Coarray Fortran, UPC, ..

- ☐ Other: ☐

Language supportWhat languages/models will be important to your application in the next 1-2 years?

	Not important	Unknown	Important	Very Important
MPI	○	○	○	○
C and C++	○	○	○	○
UPC	○	○	○	○
F90 and derivatives	○	○	○	○
Coarray Fortran	○	○	○	○
StarSS	○	○	○	○
MPC	○	○	○	○
Python	○	○	○	○
Cilk	○	○	○	○
OpenMP	○	○	○	○
OpenACC (GPU accelerator standard)	○	○	○	○
OpenCL	○	○	○	○
Hadoop/Map reduce	○	○	○	○
Intel's offload pragmas for MIC	○	○	○	○
X10	○	○	○	○
Chapel	○	○	○	○

Other languages: If there are languages that were not mentioned above, please suggest others that could be important to your project☐

Would you consider extending your code to make debugging it easier? If given a framework for enhancing the debugger's understanding of your models, would you develop plugins/settings/code to use it? 

Which basic libraries are important to you? If you use packages - such as the C++ STL, PETSC, NETCDF, or others - which ones would you like the debugger to "understand" - allowing you easy access to predefined data types