Technische Universität München
Department of Informatics
Chair of Computer Architecture

ON USING DOMAIN KNOWLEDGE FOR ADVANCED PROGRAMMING TOOLS

ANCA BERARIU

Dissertation

September 2015

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl für Rechnertechnik und Rechnerorganisation

On Using Domain Knowledge for Advanced Programming Tools

Anca Berariu

Vollständiger Abdruck der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 29.09.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 22.12.2015 angenommen.

Ich versichere, dass ich diese Dissertation selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, 28.09.2015

Anca Berariu

To my beloved dad.

*

A square is both a rhombus and a rectangle,
but it still remains a square.

ABSTRACT

High Performance Computing (HPC) is becoming available to more and more scientists and end-users, but in order to use HPC power at its full potential, scalable and well-optimized applications are required. Although performance analysis and optimization of parallel scientific applications is already a well established field in computer science, users of HPC are still facing difficulties in the optimization process, since they are experts in their domain but without necessarily a lot of experience with HPC.

Firstly, the challenges come from the complexity of the new hardware and the variety of the existing parallel programming models. Secondly, the scientific applications are also gaining in complexity, both as computation models, as well as software implementations.

Current performance tools cover enough of the first aspects, being oriented towards low-level measurements and analysis, but lack of solutions for the intricate higher-level aspects of common large scientific codes. In the same time, most newcomers in HPC are natural or computational scientists, who bring along great knowledge from their specific science fields, but have less to no experience in application profiling and performance analysis.

In this thesis we propose a method to use the potential of Domain Knowledge (DK) as a solution for narrowing the gap between application developers and HPC. We define DK as the knowledge from a particular science domain. This knowledge represents the theoretical foundation and practical core of scientific applications. In our approach, DK is based on three main concepts: DK Entites, DK Operations and DK Phases.

We introduce the Language for Domain Knowledge (LaDoK) as a means to express and handle DK. LaDoK can be used by application developers to enhance their source code with DK annotations. In this way, application specific DK elements are exposed for further processing by advanced programming tools.

In the particular case of DK-enhanced performance measurement and analysis, we successfully deployed our approach in the analysis of ParaGauss, a real-world quantum chemistry application. By enhancing performance analysis with DK Metrics like Memory per Entity (MPE), a better insight in the qualitative and quantitative memory usage behaviour was achieved. Furthermore, we show how DK Operations could be deployed for analysing load-balancing problems.

Besides quantum chemistry, we show the applicability of our approach in several other domains like multigrid methods or applications which exhibit performance dynamics.

ACKNOWLEDGEMENTS

First of all, I would like to thank Prof. Michael Gerndt, who is my *Doktorvater* in the true sense of the word. He offered me all necessary support and guidance, leading me towards the right research questions, pushing me when results were stalling and fetching me again when in doubt.

Then I would like to thank Prof. Notker Rösch for accepting to be the second supervisor of this work and for the very motivating insights regarding the next *Aha!* in quantum chemistry.

Furthermore, I would like to thank Torsten Hoefler for the short but very fruitful time I spent in his research group. An important part of the framework was developed while being there, supported by the many practice-oriented discussions.

I am also grateful to Thomas Soini, Alexei Matveev and Sven Krüger for their valuable support and patience during our interdisciplinary discussion, and for kindly sharing their deep knowledge in quantum chemistry.

I would also like to thank my two former office colleagues, Marcel Meyer and Houssam Haitof, for the very friendly work atmosphere and for the discussions on parallelism and ontologies. I also thank all my colleagues at the Chair for Computer Architecture (LRR) for the creative atmosphere and the nice time we shared together.

There are also three extraordinary school teachers to whom I owe any of my academic results, and to whom I would like to thank now: my maths professor, Mr. Nicu Miron, who taught us a very structured learning method and targeted problem solving approach; my physics professor, Mrs. Ana-Maria Nechifor, who showed me that passion and hard-work can bring together great results; and my computer science professor, Mr. Mihai Chelariu, who knew how to plant and grow the seed of this great science field into the hearts and minds of his pupils.

I also thank Daniel Ratiu for always motivating and inspiring me in research and academia and I thank Carmen Carlan for keeping the young and enthusiastic spirit.

Many thanks go to Bishop Sofian von Kronstadt for not letting me quit at a certain point in time and for his entire personal support and guidance. A very big thank you goes to my friends at the St. Silouan Chapel, as well, for their continuous support and prayers.

And last but by no means least, thank God I've come so far!

INTRODUCTION

The time when people were astonished by what computers and machines in general could do is long past. Nowadays, using email systems, GPS navigators or smartphones comes in as natural as walking or singing. Compared to the beginnings of computer science, when computers were used only by highly qualified and specialized scientists, there is a clearly different trend now, the development of new technologies being highly targeted towards usability and end-user requirements/satisfaction. It is the case for entertainment and games industry, for mobile communication and business management. There are gadgets and software tools for almost everything a user might need or wish for.

It is time HPC meets its users and shapes itself for their needs.

It is now the turn for the more specialized field of *High Performance Computing (HPC)* to enter this phase. A couple of years ago there were only few users willing and able to make use of the tremendous computing power a supercomputer could offer. These users were mainly big companies or governmental institutions, running military code, financial forecasts or weather modeling. More recently though, more and more middle to small research and production projects, from both academia and industry, request computing power for their applications. It is time HPC meets its users and shapes itself for their needs.

The presented work deals exactly with this challenge, the challenge of finding and defining a meeting line between HPC and its users. More precisely, we analyse the particular case of *Performance Measurement and Analysis* field, questioning what are the needs of the current users and defining a means to both meet these needs and facilitate new usability features.

The main group of users of HPC resources that we address is the one of natural and computational scientist. Almost each of the main natural sciences has nowadays developed a corresponding parallel computational track. We talk about physics and computational physics, chemistry and computational chemistry, biology and computational biology, neuroscience and computational neuroscience, and the list could go on. Main HPC users are thus scientists and computational scientists of different fields, who usually produce and work with simulation applications.

The first point on the meeting line is with no doubt the porting and deployment of the simulations to supercomputers. This is where application developers rely on the supercomputing centre support for information and know-how regarding that particular machine on which they want to run their simulation. This step strongly depends on both the implementation of the simulation and the architecture and system setup of the machine. It means that one might have to cope with issues ranging from simple library linking, to ports configuration, or even node topologies definition.

The steps we consider in our work though, are only those following the porting step. We try to define the intersection line for the next two phases in the simulation deployment, namely the measurement and analysis of the performance of the application. The latter one is also very closely related to the optimization following it, as optimization decisions are based on the performance analysis. We thus always have in sight that performance analysis and optimization have to share some "points" of "the intersection line".

We introduce a new paradigm to define (something like) the space of all possible intersection/meeting lines between HPC and its users. We call it the *Domain Knowledge* and we define means for expressing and using it in the advanced programming tools. It is based on the information that typical users already possess about their applications, like specific parameter constraints, data dependency or execution patterns, information which has to find its way through the code, up to the performance measurement tools. There are thus three main challenges we have to address:

1. make users aware of the knowledge they hold and educate them in identifying relevant domain specific information;

2. highlight the Domain Knowledge included in specific applications;

3. express application performance using Domain Knowledge native constructs.

MAKING USERS AWARE OF THEIR KNOWLEDGE

It might seem strange at a first glance that we have to convince scientists about the science they know ... After all, users of HPC are scientists with deep knowledge in their particular field of study. When translating this knowledge into coded applications though, most of the information suffers a transformation. What used to be a well documented phenomenon becomes merely a list of function calls, what used to be empirically studied and theoretically described complex structures become a set of indexed arrays, and, in general, what used to be explicit knowledge of natural science, translates into *implicit information* expressed by means of instructions and data flow, execution contexts and parameter restrictions. This is what actually *simulation applications* primarily are: translations of natural science knowledge into machine-executable models which reproduce the reality/nature by means of an artificial/virtual environment. The better the model, the more accurate the simulation results. But better models usually mean larger models, more complex models, more computation expensive models. This lead to computation power thirst and actually to High Performance Computing development.

Simulation applications reproduce nature in a virtual environment by translating natural science knowledge to machine-executable code.

Natural science simulations in HPC experienced a few adjustment phases. First of all, existing models had to be adapted and ported to the parallel environment. Afterwards, optimization of the used algorithms, methods and solvers gain importance for the parallel performance improvement. Specialized parallel versions developed to overcome common encountered issues and bottlenecks, like master overload in master-slave patterns, or load imbalance for computations based on irregular grids. In order to cope with and to enhance the features of compilers and hardware developing alongside the applications, two other means of adjusting and preparing the code for HPC were taken into considerations: low-level optimization and application tuning. This brought again new improvements in code performance, using techniques like loop unrolling, array alignment, or cache optimization.

Where is the natural science behind all this code?

We are thus nowadays facing the challenge of having to run very complex models and code on very complex systems, and all this with the best performance possible! We struggle often on several optimization levels to achieve linear scaling and best execution times for the code we have ... but what is actually the main goal of the application? What is that is being computed/simulated? Can we still see the forest behind the trees? Where is the natural science behind all this code?

We introduce by *Domain Knowledge* the missing information layer, to hold just this bird's eye perspective of the application. As level of abstraction, it is situated just above the application layer. As contents and semantics, it summarizes valuable information hidden implicitly in code or written down in "passive" comments or even in published research papers. One could actually build a parallel to the comments developers usually insert in their code: they do not influence the code execution and, in general, some of the information is already implicitly included in code. Still, comments are useful for further developers, for users of library interfaces, if that's the case, or just for the same developer when turning back to this code after a while. The same could be said about Domain Knowledge too: applications perform just fine without perceiving it, but they could perform even better if they were aware of it.

Domain Knowledge: applications perform just fine without perceiving it, but they could perform even better if they were aware of it.

So what is this knowledge which scientists have but did not explicitly include in their applications yet? One simple example could be shown from computational chemistry field, more precisely from quantum chemistry simulations. Given is an application which computes the geometry of a cluster of molecules composed of about 100 molecules of 3 different types. The simulation uses the *conventional Fock matrix construction* for the two-electron integrals precomputing step and then the *SCF approach* to solve the Kohn-Sham equations. This information could be easily extracted from the code itself too[1], but it does not bring too much insight without the real domain knowledge behind it. Both methods are, from a computer scientist's perspective, computations on some array elements. It is the domain knowledge though, that could explain what these arrays are and what are the dependencies and resulting data flow. It is the domain knowledge that groups these arrays into orbitals or functionals and decides upon which molecule type is likely to require more computation time or more memory space. It is also the domain knowledge that explains which integrals should be grouped and computed together and whether the resulting Fock matrix will have a specific structure (and hence a particular memory access pattern).

HIGHLIGHT DOMAIN KNOWLEDGE INCLUDED IN APPLICATIONS

We have shown so far that there is more information in the source code than we actually use and there is even more information scientists hold and could be very useful. How to express and highlight this information?

Entities

First of all, we have to identify the structure of the information we want to express. Most of the time simulations start from a set of formulae which are modeled by the computational branch of that particular science field. Each variable and parameter of a formula is translated to a variable or constant of the programming language. But variables in formulae have their well established semantics, whilst variables of programming languages are just data structures. We thus have a first type of domain knowledge which has to be highlighted: semantics of the variables in formulae. We call these constructs *DK Entities*.

Operations

The main part of the code is built on algorithms and solvers based on these formu-

1 Techniques for extracting naming databases from codes are common in Domain Specific Languages (DSLs)

lae and working with these variables. They mostly translate to functions or libraries of functions, but again, there is more about a solver, for example, than it is written using the programming language. A detail as simple as the current recursion step is mostly missing. Another case which lacks of information is the case of message passing implementations (MPI), where usually there is no clue about the pattern or structure of data being exchanged. We thus identify the next type of domain knowledge, namely *DK Operations*. These are using and processing the previously defined DK Entities.

Phases

Even less documented in applications are the transitions between the different phases of a simulation. Unlike operations, which are more or less statically describing the planned instruction flow, there are often cases where the application flow depends on the computed values. A good example are the adaptive solvers, where depending on the results of the computation for some points of the grid, a refinement of it might be needed. In this case, usually the same *operations* will be applied again for the new grid configuration, but for the simulation semantics it is another *phase* of the computation which is being executed now. This is the third type of domain knowledge, namely the *DK Phases*.

Looking at the three types of information we identified, one can observe that all this knowledge is more or less orthogonal to the source code of the specific applications. Thus the means by which we express the domain knowledge has to provide both *inside the code* and *outside the code* documentation. We propose in this work the Language for Domain Knowledge (LaDoK) based on the LaTeX syntax and acting like both a documentation and annotation language. LaDoK includes exactly the three identified constructs: entities, operations and phases. It gives developers and domain experts the opportunity to enhance their application code with domain knowledge, it helps performance analysis tools to carry out focused measurements, and it supports domain users in analysing the performance of their simulations, to mention only some of the use cases.

EXPRESS PERFORMANCE USING DOMAIN KNOWLEDGE NATIVE CONSTRUCTS

One of the advantages of using domain knowledge in applications is an enhancement of the performance analysis processes. Structuring, defining and delivering constructs like entities, operations and phases for own simulation code, gives the developer the possibility to expose them to the performance measurement, analysis and visualization processes too.

Performance analysis tools developed primarily as tools for the computer scientists supervising their clusters and supercomputers. It is thus easy to understand why such tools are delivering performance results targeted actually towards computer scientists rather than towards natural or computational scientists, being thus not so easy to use for common application developers. The same as for the case of different optimization levels described above, performance measurement and analysis usually limits itself to either very hardware-close properties like *cache misses, stall cycles* or *floating point operations number*, or to analysing communication behaviour, mainly MPI, for which it delivers results regarding, for example, *broadcast synchronization times, delayed receives* or *message size*.

DK Metrics combine domain knowledge constructs with common performance properties

We introduce the *DK Metrics* to help performance tools and simulation developers find a common workspace. These new metrics combine the domain knowledge constructs explained above - entities, operations and phases - with the performance specific measurements, like memory usage, computation time or communication patterns.

One such metric is, for example, the *Memory per Entity (MPE)*. It is much easier for an application developer to decide upon performance and possible optimization of their application if it were provided, for example, not just with the overall memory usage information or the amount of allocated memory per each allocate function call, but rather with the MPE information, the memory used by an entity throughout the execution. For our previous example from computational chemistry, it would thus be useful to see for which type of molecule is the memory being allocated, than just the plain information about the function in which the allocation takes place.

Another example is related to the common *load imbalance* problems. It does bring more insight into an application if, for example, beside the comparison between the execution times of a specific function on different processes, the user would be provided with computation load analysis in terms of *computed entities*. Solutions for load imbalance problems could thus be found, in the improved entities distribution, or also in a per entity type operation customization.

Rather than being stand-alone metrics, DK Metrics are thought to be used in combination with consecrated metrics. As stated before, Domain Knowledge does not substitute the current performance measurement and analysis techniques, but rather enhances them by adding to the existing frameworks a level directed towards application developers.

For example, timeline visualization is proven to be very useful for visual qualitative (compared to quantitative) performance analysis. The so called Iterometers, which are another type of DK Metrics, use the timeline view to insert new information on runtime events. For example, the multigrid level indicator, which marks the changes between different levels of detail in the multigrid solvers.

CONTRIBUTIONS

The main contribution of the current work is the analysis and definition of the Domain Knowledge approach. Both the theoretical description as well as practical proof were considered: related work research, problem formulation and solution description, implementation and evaluation.

Throughout the work:

- we identify and highlight the benefits of using Domain Knowledge. In particular, we focus on the performance measurement and analysis for simulation applications in HPC;
- we define the Language for Domain Knowledge (LaDoK) as a means to express Domain Knowledge;
- we define a new type of performance metrics, the DK Metrics, and analyse their benefits in the analysis of both tracing and profiling performance measurements;

- we present our implemented LaDoK Framework, which provides the necessary support to deploy LaDoK on real world applications;
- we present our implemented DK-enhanced performance tracing framework, which features support for the DK Metrics along with common performance analysis;
- we evaluate DK on the quantum chemistry application ParaGauss;
- we present deployment methods of DK for different application types: memory-intensive applications, load-imbalance problems and multigrid methods.

This thesis is structured as follows.

In chapter 1 we describe the evolution of the computational sciences and we give an overview of the main aspects in simulation engineering and the challenges which scientific code has to face in HPC.

In chapter 2 we describe the state-of-the-art in performance measurement and analysis.

In chapter 3 we analyse which other fields from computer science make use of domain specific aspects. We also look at the related work in the performance tools context.

We then introduce our approach in chapter 4 and the LaDoK syntax and semantics in chapter 5.

In chapters 6 and 7 we introduce the two frameworks which we developed.

Lastly, chapters 8 to 10 present a series of use cases for DK and chapter 11 summarizes the thesis and provides future work idea.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 SCIENCES IN THE HPC CONTEXT

Regardless of the science field we would talk about, one cannot imagine research nowadays without computers. Be it history or linguistics, geology or music, any research work will be at least written in electronic format, questions and answers will be exchanged via emails, world wide web will be used as a huge library and results database.

There are some specific research fields though, where computers became nothing less but *the key* for their entire research work. It is the case of the relatively young *computational sciences*, which developed along the well established natural science research fields. Almost every natural science has nowadays a corresponding computational branch: chemistry - computational chemistry, physics - computational physics, biology - computational biology, finance - computational finance, astrophysics - computational astrophysics, neuroscience - computational neuroscience, and the list could go on. As their name implies, the main goal of this class of sciences is to find results and answers to scientific questions using computing methods, as opposed to the classic methods of theory and experiment. And what else would suit better for solving computational problems if not computers themselves? As a result, many scientific applications and simulation software emerged in the computing world and pushed computer scientists and engineers to deliver more and more computing power.

High Performance Computing (HPC) came as an answer to the computing demand and developed meanwhile into a research branch of its own. It aims to provide advanced computing systems and tools, on which simulations can be run at high performance. As there is no such thing as a free lunch though, using HPC brings along several challenges which both computational scientists and computer scientists and engineers have to learn to cope with.

1.1 FROM NATURAL TO COMPUTATIONAL SCIENCES

At their very beginnings, natural sciences developed from humans' quest to understand nature, to explain the world surrounding them. At about 600 BC, Thales of Miletus, who is now considered *"the father of western natural sciences"*, refused to admit the supernatural or mythological explanations of natural phenomena and set his purpose on explaining them by referring to the natural processes themselves. Thus the first principles of the nature of objects appeared and hypothesis regarding their specific behaviour were advanced. A remarkable contribution followed afterwards from Aristotle who brought the previously rather *speculative work* towards a more *empirical approach*, based on observation, analysis and categorization. As a result, physics, cosmology and biology teachings started developing towards stand-alone sciences; chemistry followed only later in time, when it managed to replace alchemy, which, in its quest for the philosopher's stone, managed to carry its mysticism over centuries to go.

Natural sciences started thus as rather descriptive works, mainly based on empirical investigations. The tendency changed gradually towards more explanatory approaches and along with that grew the importance of mathematics in the study of sciences too. If at the beginning only few mathematics were used, mostly geometry and basic algebra in astronomy and physics, the situation changed with the 16th century, when the potential of more advanced mathematics tools began to be perceived. It was not to be waited for too long though, as the period known as the *Scientific Revolution* brought along a huge step forward in both natural and formal sciences.

From philosophers to scientists

Enjoying now much better means to express thoughts in an abstract manner and disposing also of the *scientific methods* developed over the time, the once called *natural philosophers* changed to become *natural scientists*. It is at that time that Galileo promoted the idea that mathematics provided the certainty which was necessary in science. Hence experimental measurements and observations were now to be compared and validated against the mathematically computed values.

Computing for evaluating, computing for exploring

Computing emerged basically as a means to *evaluate* theories and experimental results against each other. Just a few centuries afterwards though, the advanced mathematical models used in all natural sciences were pushing human curiosity again. This time the quest was, for example, that of finding valid solutions in a huge spectrum of candidates, or exploring behaviour of matter at its natural limits, limits which could not be reproduced in laboratories or, if so, only at very high costs. Knowledge did not need to be validated anymore, but rather enriched and deepened with new information, new solutions and insights.

Computing machines came as a solution to overcome the human computing limitations. Even simple pounchcard-controlled systems were able to perform way more computations than would have been possible by "human force". But operating computers was not really an easy task to do and hence only a limited group of scientists and engineers would eventually learn how to use them. In the last decades this changed to having scientists that had to develop *programming skills*, but in the same time deepen in the *computational view* of their own research field. These are nowadays the *computational scientists*. They are most like the early natural philosophers and the practical scientists nowadays, in that they have some kind of *experimental laboratories*, set up from dedicated hardware, software frameworks and libraries, and they develop new *scientific methods* when they design their models and algorithms. In the same time, they also differ from the previous scientists, due to the great new challenge they have to face: they basically have to first create a complete *virtual replica* of the real world in order to be able to analyse and understand it. They need to *mirror all structures, rules and behaviour* of nature in a way that they can be processed by computers but in the same time remain consistent with their initial model, the nature itself.

Computational laboratories: experiment with virtual replica of the real world.

COMPUTATIONAL BY NATURE

Most scientific applications target some kind of numerical calculations of data.

Computational chemistry

In *Computational Chemistry*, for example, the goal is to determine different chemical and physical properties like energy, forces or geometry of atoms, molecules and clusters of molecules. The main challenge is that of calculating the electronic structure of the multi-electron systems, as all other properties can be accessed from this information. The starting point is the Schrödinger equation, for which several solving approaches were proposed over the years. Important to be mentioned here is the fact that all of these methods eventually use an approximation step - commonly the Born-Oppenheimer approximations [2] - which means that, except for the most simple atom - the hydrogen, no exact solution is being computed. This is why *several* methods had to be developed, as each of them is suited for a specific use case. It is thus in *"the art of computing"* to know which combination of solvers, functionals and approximation parameters to use for every particular problem. By *problem* we refer to both the type and size of the chemical system, as well as its specific property that has to be computed.

The Hartree-Fock method (HF) was introduced just some years after the Schrödinger equation was published[1]. As known to the day, Schrödinger equation cannot be solved exactly, except for one-electron atoms, and thus the solution of any system larger than that had to be approximated. Hartree's assumption was that the exact N-body wave function of a system can be approximated by the N-spin orbitals of its components. In other words, the exact molecular orbital (MO) can be approximated by an expression[2] of the individual electron orbitals (atomic orbitals - AO).

The HF method exhibits a high computation demand and it remained poorly used until the introduction of electronic computers in the 1950's. Another disadvantage of the method was represented by the fact that it did not account for the correlation energy of the electrons, which means that the computed energy values were impracticable for molecules with large number of electrons or with bound breaking behaviours [26]. There was effort put into finding methods for compensating this shortcoming by adding approximations for the correlation energy at later steps in the solving process. Variants of the HF like Møller-Plesset of n-th order (MPn) and Conjugate Gradient (CG) were developed, but in the end all these methods did not enjoy a wide spread acceptance.

What came in as a breakthrough among the computational methods, though, was the shift towards a more *intuitive theory*, as introduced by the Density Functional Theory (DFT)[3]. Although not accepted at the beginning in computational chemistry due to its large errors in molecular calculations, DFT made soon enough its way through the "legitimate quantum methodologies" and it is now the most widely spread computation method used by computational chemistry in HPC. Using the last developments in theoretical approximations, DFT delivers almost exact energy values and it is also much less computation demanding than HF.

Even if the "success key" was the switch from the *uninterpretable* wave function[4] to the *physical observable* electron density [12], there was still lot of effort to be invested in efficient implementations for parallel computers. Jong et al. offers a great survey in [35] on the development and current status of computational chemistry. We refer to some of these aspects in section 1.3.

To grasp the real importance of such applications and understand the attention given towards continuously improving both prediction power as well as computation speed, it is enough to look at the computational chemistry applications within

1 Schrödinger equation - 1926; Hartree method - 1927; Hartree-Fock method - 1935.
2 Hartree-Fock method uses the Slater determinant.
3 Hohenberg-Kohn theorems - 1964 [30]; Kohn-Sham equations - 1965 [38].
4 As Cramer says in [12] "a wave function is an inscrutable oracle that returns valuably accurate answers when questioned by quantum mechanical operators, but it offers little by way of sparking intuition".

materials science and within related industries: solar cells are based on the thin-film material technology, computer chips depend on the nano-materials fabrication, polymers and plastic industry rely on the chemical catalysts [11]. It is no wonder that a large percentage of CPU hours on supercomputers are currently being used by DFT computations.

Computational physics

Talking about importance, materials science highly depend on the computational chemistry, but there is also a strong dependence on other scientific fields, and among those *Computational Physics* occupies by far the first place. As a matter of fact, some fields of the computational chemistry are so tightly connected with computational physics, that there are even research branches like *Physical Chemistry* or *Chemical Physics*.

There are very many other fields in physics though and most of them have nowadays a computational branch. The chronological pattern already described for computational chemistry applies for most of these other sciences: starting from a given theoretical principle, one has developed numerical solutions based on approximations, existence and convergence theorems. These were translated to corresponding algorithms, once programmable computers appeared, followed by successive improvements of both the implementations and the methods behind.

In the following we mention only few physics branches which are of importance for the HPC field.

- *Molecular Dynamics (MD):* unlike computational chemistry which basically looks *inside* the molecules, MD looks at molecules and atoms from *outside*, studying their physical movements and interactions.
  The computations in MD are based on solving Newton's equation of motion for a system of interacting particles [62].
  MD application fields include biochemistry for molecule docking and protein structure determination.

- *Computational Fluid Dynamics (CFD):* aims at analysing problems regarding fluid flows [88]. CFD methods are based on solving Navier-Stokes equations using a discretization method like Finite Volume, Finite Element, or Finite Difference. Application areas range for automotive industry to aircrafts and wind turbines.

- *Astrophysics:* actually part of astronomy, it deals with the physical properties of celestial objects, as well as their interactions and behaviour. In astrophysics, very many disciplines of physics are applied [27].

COMPUTATIONAL BY "BRUTE FORCE"

All of the computational applications mentioned so far could be classified as being *numerical intensive*. That means that results are based on a large amount of calculations, mainly evaluating numerical formulae.

There is another increasing group of applications though, which is gaining field in the HPC. These applications do not mainly rely on numerical calculations, but rather on huge data processing activities. They can be characterized as *data-intensive* applications. Data in this case can be either input data, as for data mining and other search-related applications, or output data, as in weather forecasts.

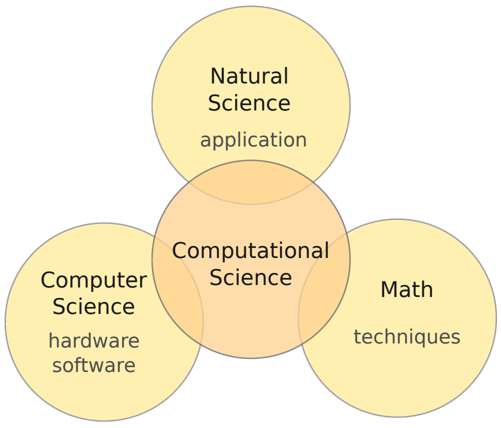


Figure 1: A representation of the multidisciplinary nature of computational science both as an overlap and a bridge between natural science, applied mathematics and computer science (adapted from [40]).

One typical example are the applications for *drugs discovery* in the biochemistry field. These applications are based on the molecular docking approach and they search for combinations of keys - small drug molecules - which fit into the given lock - proteins [2], [43]. The single fitting checks are not very compute expensive, but there are very many combinations to try out.

Another interesting example comes from the *computational neuroscience* [81] which aims at mimicking the human brain connections and activity by using computers. The theoretical number of required computing units is comparable to the number of neurons to be simulated.

1.2 SIMULATION ENGINEERING

If theory applies theorems and rules, and practice carries out experiments and measurements, than the computational side of the scientific work uses mainly computer simulations. At the beginning, simulations emerged as a means to mirror the way nature works. It was an opportunity to replace expensive or impractical experiments. Think about crash experiments in automotive industry or star collision, for example. On the other hand though, simulations do not only *replace* experiments, but actually *stimulate* a new way of thinking. Computational potential inspires new questions to be answered, new views to be explored. Landau, Paéz, and Bordeianu explains in [40] that computational science is not just simply the *overlapping* of three other sciences, namely, *Computer Science, Mathematics* and a *Natural Science* (see Figure 1), but represents a field by its own, *bridging* these other sciences together.

It is worth having a look at the main components of the bridging area in computational science, as these are the ones leading both the challenges as well as the achievements of simulation applications.

Mathematical formulae

As seen in the previous section, all computational "native" sciences are based on a mathematically formulated principle from theory: Schrödinger equation for computational chemistry, Navier-Stoke equations for CFD, Einstein's general relativity based equations for astrophysics and so on. They define the unknowns or variables to be computed, and the operators, functions and relations between them. It is important to be mentioned here, that these mathematical models represent the linking point between the real nature and its virtual representation. Hence, for the latter one, mathematical models represent *the truth* which needs to be followed. The models could state, for example, which values are allowed for specific variables and in which parameter configuration, or whether two terms depend on each other or not.

Numerical methods

The main issue about the equations mentioned above is that they cannot be solved directly, and hence numerical methods have to be applied. In order to do so, most equations are translated into an algebraic form, involving usually matrix and vector operations.

Numerical solutions bring along an important challenge: computational errors. These can result from the numerical approximation, like discretization and truncation errors, or from computer number representation, like roundoff errors. No matter which kind of errors, they all have to be assessed quantitatively along with the chosen computational method, but also validated qualitatively from the natural science point of view.

Classical examples of numerical methods include, but are by far not limited to QR factorization, Jacobi, Gauss-Seidel and Runge-Kutta methods or multigrid solvers.

Computer applications

The path from the numerical algorithm to the working simulation application is neither straight, nor neat. First of all, one has to carefully select the *programming language*. It is not just about learnable programming skills that one or the other might acquire, but it is more about language particularities and limitations that strongly influence the quality of the future simulation. Common programming languages for large simulation codes include C, Fortran or Matlab, but other languages like R or Python are becoming prevalent too.

Recently, an increased attention has been given to the *architecture* of the machines on which simulations are to be executed. Multi- or many-core systems, graphic cards (GPU) or general purpose processors, clusters or single powerful machines, provide as many potential execution environments, which should be carefully thought about.

Another trend regarding the simulation applications is the software *composability*. This has its roots in the advances of computer science techniques beyond object-oriented programming, to plug-in based frameworks. Computational scientists dispose nowadays of a large database of libraries and software packages for almost any method or solver, which are deployed mainly as black-box sub-components of larger simulations. One is thus spoilt for choice, as it is not an easy job to choose the perfect solver implementation, given a particular sub-problem. For example, one cannot rely only on the API description of a library alone. The choice for one specific library or the other should be backed-up by a good knowledge and practical experience in the scientific application field.

Simulation data

What Hamming was saying in the 70's about numerical methods [5] could be extrapolated nowadays to simulations: *"The purpose of simulations is insight, not data."*. Still, in many cases handling the simulation data requires special attention due to the challenges it could present.

One such challenge is the *size of the data*. No matter if input, output or intermediate data, size has always been a problem. For the first two types of data, I/O operations are the main issue: reading and writing data has always been slower than processing/using it. As for the runtime or intermediate data, it is the problem of the limited

5 Exact quotation: "The purpose of computing is insight, not numbers." [23]

main memory size. Although it offers a much better access speed, it does not compare in capacity with the secondary memory. This is the *no free lunch* of the memory systems: speed versus capacity.

Fortunately, most simulations only have to deal with one of the two challenges: it is either I/O or main memory excessive load, but only seldom both at a time. *Data-intensive applications* have either large input data sets, for example the data-mining or visualization applications, or large output data sets, like applications from molecular dynamics, CFD or weather and climate modeling. On the other hand, *memory-intensive applications* do not have large input or output data sets, but need and generate during computation large intermediate data sets. This is the case for applications from bioinformatics and quantum chemistry and for simulations involving graph algorithms [75].

Simulation parameters

As strange this might sound, simulation parameters are the most fuzzy, almost mystical part of the simulation engineering. We refer here by parameters to all values fed to the application either as input data, as environment or startup settings, or even as hard-coded constants. It is often the case that these parameters, although mostly simple integer or floating point values, posses a kind of *magic*, with whose absence one could not even start the simulation, not to mention about getting correct and useful data.

Parameters are very tightly related to the scientific domain and particular problem an application deals with. Much like in the experimental approach, where the scientists carrying out lab experiments are very much aware of the quantities they are working with - milligrams of chemical substance, voltage on a circuits board, radiation intensity, etc. -, the same applies for computational scientists using their simulations: they have to have a sense of what the maximum iterations count should be, which extra environment variables to set, or which grid size to use.

More often is not even the exact value that is needed, but at least the order of magnitude of that parameter. A good example in this sense is given by the time values for Molecular Dynamics simulations [62]. Newton's equations of motion for a very large system of interacting particles have to be solved in this case. There are two time-related questions for the simulation:

TOTAL TIME DURATION: what is the time span that should be simulated?

TIMESTEP: what is the time span between two successive evaluations of the forces acting on the particles?

For the first parameter, one has to consider two contradictory aspects: on the one hand, simulations should be relevant for the natural process they mirror and hence at least as long as one "step" in the kinetics of the simulated process. One cannot analyse the walking process from just half of a footstep. On the other hand though, due to the large amount of particles, simulations are very ill-conditioned, which means that the computation error propagates and amplifies with the number of iterations, and hence with the total time duration. Under these constraints, simulations for DNA, for example, have a total simulated time spanning from nanoseconds (10^{-9} s) to microseconds (10^{-6} s).

Regarding the other time variable, its lower and upper limits are given by other two constraints. First of all, evaluating the forces acting on the particles is very time consuming, so one would like to have as less evaluations or iterations as possible, and hence as large timesteps as possible. Think about the fact that a DNA simulation of the order of microseconds might require as much as a couple of CPU-months or CPU-years to complete. In the same time, a too large timestep introduces discretization errors, and hence it should be smaller than the fastest vibrational frequency in the system. This is why a typical time span for an iteration is only in the order of femtoseconds (10^{-15} s), which is by some orders of magnitude smaller than the total time duration mentioned above.

1.3 HPC CHALLENGES

We have seen so far that simulations "thirst" for computational power. It is now time to see which are the challenges that simulations have to overcome in order to access this power on the HPC systems.

As HPC gains its tremendous computational power by integrating a large amount of nodes of medium performance, it is obvious that the prerequisite for any application to run on such a system is to be implemented with parallel support. Nowadays solutions include either combinations of a common programming language, like C or Fortran, and a library for the parallel support - here MPI and OpenMP implementations are by far the most widespread solutions -, or specialized parallel programming languages and extensions like UPC, HPF, X10, Charm++, Cilk, and the list could go on.

It is out of the scope of this section to argue about advantages or disadvantages of the different programming languages, neither about the parallel software design. For these topics we refer the interested reader to [look for references]. We rather want to give a picture of the basic challenges of parallel implementations in general and move on to the similar issues in the HPC context, always keeping in sight the natural sciences behind.

Parallel computing implies two closely interrelated concepts:

WORK DISTRIBUTION: each computing unit executes only a part of the entire computation work

DATA DISTRIBUTION: each computing unit uses or processes only a part of the simulation data

In literature one distinguishes between *distributed* and *parallel* computing [57], we shall use the terms in this section interchangeably, closer to the *parallel* meaning.

Intuitive examples for parallel computing are delivered by CFD simulations. Given is a simulation domain - a parallelepiped for example -, the properties of a fluid which flows into this domain, possibly an obstacle inside the domain - a cylinder or alike -, and the initial pressure, temperature and velocity conditions. Required is the state of the fluid inside the domain at a given point in time.

In order to solve the Navier-Stoke equations, the domain is discretized to a grid of points. The way the work should be distributed is straightforward: each processing unit solves the equation only for a subdomain of the entire computation domain. This determines the data distribution too, namely each processing unit receives and holds the data of the grid points it computes for.

Data dependency, communication and synchronization

Two questions arise though even for this very simple scenario: in order to evaluate the velocity at one grid point, one needs information about pressure from the neighbouring points. This means that for the grid points situated on the boundary of a subdomain, information held by another processing unit, the one computing for the neighbouring subdomain, is needed. Moreover, the values required from the other processor have to be from exactly the same timestep as the one of the computation on the current processor. *Data dependency*, *communication* and *synchronization* are thus key issues in parallel computing.

The same principles apply when talking about High Performance Computing too, but in this context the *large scale* characteristic adds up for the level of complexity of each of them.

Granularity

We use the same example to show another challenge which comes along with this large scale characteristic. Consider now that the number of available processing units is so large, that each one would receive a subdomain of only few grid points to work with. In the same time, it will still have to communicate with all of its neighbours to exchange computed values. Communication is much slower then computation, and thus too less computation always results in processors being idle while waiting for communication to finish. One talks about the *granularity* of an implementation to relate to the amount of work a processor is executing between two consecutive communications. Embarrassing parallel applications have the ideal situation where no communication is needed during computation, like the QCD codes for example. All other applications have to find the optimal combination of computing and communicating.

Another straight forward example of how common issues in parallel computing grow worse when changing to large scale computing is given by the very common *master-slave paradigm*. Consider the case where work packages are not statically defined by the computation domain, like in CFD, but have to be dynamically allocated at runtime. For a rather small number of total processing units it should be enough, if one processor took the *master* role and distributed upon request work packages for all the other processes, called now *slaves*. What happens now, if the number of slaves increases by some orders of magnitude? Depending on the size of the work chunks too, the master will be soon overloaded by work requests. In the same time, the slaves will be idle waiting for work packages. This is one kind of *performance bottlenecks* in parallel computing which are most easy to be detected.

Master-Slave bottleneck in HPC: too many slaves for one master

The talk about dynamic work allocation brings us to the next very interesting and demanding subject in HPC, namely *load balancing*. As long as all working nodes have the same amount of computation to carry out, it is said that the overall execution is load balanced. If, in contrast, some nodes have more work to do than others, then the computation is load imbalanced. The problem is that in load imbalanced simulations some of the nodes are idle, while they could help other nodes and thus the entire computation would be accomplished faster.

Computational load depends, of course, on the size of the work chunks each node has to take care of. Again, the ideal situation is the one similar to the CFD example mentioned previously, where all processing units receive equal subdomains to work on.

Difficult to balance: two-electron integrals

Things are not that trivial in all applications, though. In the implementation of the HF and DFT methods referred in the previous section on Quantum Chemistry, achieving a load balanced computation of the two-electron integrals is a complex job. The computation time for such an integral depends on the one hand on the angular momentum of the two electrons considered, and on the other hand on the contraction of the basis functions. A highly contracted basis function will impose a larger computation time due to the increased number of floating point operations required to use this function in the computation of the integral [35]. It is only with careful consideration for integrals grouping and *reusage of intermediate computation results*, that a balanced computation loading could be targeted.

New challenge in HPC: fault tolerance

An emerging challenge specific for the large scale computing systems is the *fault tolerance*. Given the increasing number of computing nodes on each machine, the probability of a fault occurrence is getting large enough to make a difference. For exascale computing it might be expected for example, that the node fail rate will be less than an hour. This is much less than complex simulations need to finish their computations. Current applications do not account for the possibility that a node delivers wrong or no results at all. Solutions and support is to be sought for at the edge between computer science and computational science in research topics like check-pointing or recursive protocols.

2 PERFORMANCE MEASUREMENT AND ANALYSIS

The huge computing power which High Performance Computing systems offer nowadays was asked for mainly by simulation developers and scientific application users. Their argument therefor was commonly expressed as a time requirement: *"we need to compute the results faster"*. But for such complex hardware architectures and systems like those in HPC, "faster computing" is not anymore simply based on the improved CPU clockspeed. Even for simple end-user systems, when it comes to appreciate the system's performance nowadays, one considers besides the frequency of the processor, at least the number of cores and the amount of main memory. Even more for HPC, one has to consider components and values like cache memory size, bus bandwidth, or hardware accelerators to just get to a rough idea about the performance of a single node or socket. It follows then the network connection between the nodes and between the racks of nodes, with particular topologies and access speeds and we start to realize just about how many components contribute to achieve the otherwise simply formulated requirement - "faster computing".

Unfortunately, simply running an application on a high performance system, does not implicitly mean that the application executes with a high performance, too. As also seen in section 1.3, one needs to tune their application in order to take advantage of the computation power. Even though the system is able to perform a huge number of floating point operations per second (FLOPS), there might be many idle times, due to synchronization points for example, where some processes wait for other processes to finish their work. In this case, the results are indeed computed very fast, but the overall execution and final results delivery might not be accomplished as fast as expected.

Moreover, current trends in HPC add new performance requirements for applications. It is no longer just the time component that is important for the execution, but also resources utilization, efficiency or power consumption, like in green computing or environment friendly computing.

In this context, where application performance is influenced by so many hardware and software properties and is combined with different economic terms too, it is almost impossible to properly track and evaluate performance without using appropriate tools. Such tools have already been developed and they are based on the performance analysis flow depicted in figure 2.

In the remainder of this chapter we first discuss each of the four main phases of the performance analysis flow, together with the basic concepts used in the *Performance Measurement and Analysis* field, and then we provide a short overview of the main existing tools.

2.1 PROFILING VS. TRACING

In order to address the performance of an application beyond the mere overall execution time, one needs a more or less complete image of the behaviour of that specific

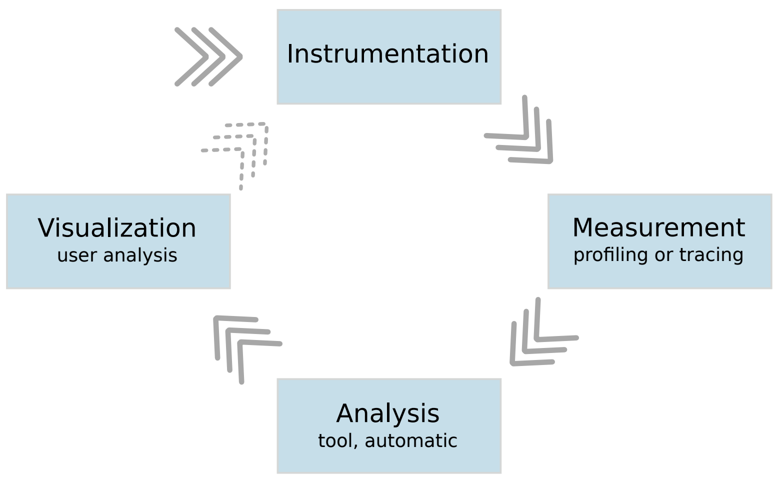


Figure 2: The main phases in performance measurements and analysis: instrumentation, execution, analysis tool-side, visualisation (analysis user-side).

application. One needs a profile where relevant information with respect to execution characteristics and settings are gathered.

There are mainly two methods used by the current tools to create such profiles: *profiling* and *tracing*.

PROFILING: in the profiling method, performance properties are *accumulated* and *processed* throughout the execution, such that in the end only the representative value is presented for each such property. For example, the total cache misses count, or the execution time of each function.

TRACING: compared to simple profiling, tracing delivers more detailed information. Each measured property or intercepted event is stored over the execution, such that in the end all these values are available, not just the accumulated end values. For example, one could check how many cache misses occurred throughout different calls of a function, instead of only considering the overall count.

Very often, a timestamp is associated to the measured values and thus events can be correlated in order to deliver more insight into the execution flow. For example, this is the case for send and receive events in a message passing based implementation: late sends could be detected, in that the timestamps of the send-receive pairs are compared with each other.

Which exact information is being gathered either by profiling or tracing, depends on the tool capabilities, the libraries and the hardware support and it can include:

- time information: timestamps, execution times;
- counter values: software counters (e.g. function calls count), or hardware counters (e.g. cache misses, stall cycles);
- MPI and OpenMP events: MPI sends and receives, OpenMP barriers, etc.
- system status: memory or CPU usage.

Depending on the type of information which has to be gathered, the tools implement either an *event-based strategy*, where every new event triggers a new record in the profile log, or a *sampling-based strategy*, where the status of the execution is checked at fixed time intervals. The first strategy applies, for example, for MPI communication events, while the second strategy is used for counters interrogation.

Overhead and limitations

In order to have a complete performance profile, one would have to store all events which occur throughout the execution and all counter values, at the highest sampling rate possible, of course. But this is mostly impossible, or highly unfeasible, especially for tracing long-runnig or large scale applications.

On the one hand, large execution times for applications using many processes or threads, generate a very large amount of tracing data, which might just not fit on the machine. The size of the data increases with the number of events and with the sampling rate.

On the other hand, any profiler or tracer introduces some overhead, which also depends on how "agressive" their measurement strategy is. The more queries or write commands, the larger the overhead introduced.

A good balance of the amount of measurements and of the stored data is needed.

A common practice is to first run profiling measurements for an application, then based on the results, spot the possible performance bottlenecks and then run tracing measurements targeted to the identified possible problematic spots.

2.2 INSTRUMENTATION

Some of the information gathered by the performance tools can be retrieved directly from the environment where the application is being executed. Such are the information regarding the main memory status, which is received from the operating system, or the values of the hardware counters, which are provided by special libraries.

But there is also information which cannot be retrieved without direct support from inside the application itself. Take for example, the value of a loop counter. There is no means to retrieve this information, but only by inserting in the application a call to a back-end library in order to expose it. This process of modifying and enriching the application for performance tools support is called *instrumentation*.

Source code and binary instrumentation

Current performance measurement tools implement instrumentation either at the source code level, at some intermediate code level, or directly in the binary of the application. If the user is given the possibility to insert own instrumentation calls, then this will happen at the source code level and it is labelled as *manual instrumentation*. The other option is the *automatic instrumentation*, which is done by tools based on their standard strategies, as well as based on the parameters and settings specified by users. For example, one might choose to instrument only MPI calls, or to leave aside from the instrumentation some modules or source files.

2.3 VISUALIZATION AND ANALYSIS

The whole purpose of running performance measurements after all, is that of finding out how good the performance is and whether it could be improved. Hence, after having instrumented an application and having executed profiling or tracing runs on it, it is time to *see* what the performance is and *decide* upon possible improvements: *visualisation* and *analysis* are the key steps leading to performance optimization.

Due to their strong interdependence, one could argue about the appropriate order in which these two steps should be considered. In figure 2 we present analysis as preceding the visualization step, while here we are stating that one first has to *see*, i.e. visualize, in order to be able to *decide*, i.e. analyse. The key for answering this problem is to differentiate between the *tool-based analysis* and *users' analysis*. From the perspective of a profiling tool, the analysis comes right after or during data acquisition. The raw data is processed and the results are delivered for being visualised by users. From a user's point of view, the performance data must be first looked at, in order to be able to draw some analysis. Following the main idea that *Domain Knowledge* should penetrate from the abstract layer to the low-level layer, we present in this section the second perspective.

2.3.1 *Visualisation*

As shown above, there are many factors and data which build up what is called the performance of an application. It is thus to be expected that there are also many ways in which performance profiles can be displayed.

Raw, text output

```
> Time measurements:
main()          30.5 s
gen_matrix()     8.3 s
distribute()     5.9 s
comp_idx()       5.2 s
...
```

Consider, for example, *time measurements*. One straightforward solution is the simple tabular or ASCII output of the elapsed time for each function or instrumented region. This is useful for a fast check or comparison of execution times.

Processed values

```
> Time measurements:
main()       30.5s 100.0%
gen_matrix() 8.3s  27.2%
distribute() 5.9s  19.3%
comp_idx()   5.3s  17.3%
...
```

A step towards improved visualisation and analysis experience is to process the raw measurement values and to provide with extra information. The most simple example in our case are the percentage values. The percentage of the total execution time spent in a specific function is a better means to express the cost of a function, than the sheer absolute time values.

Correlated measurements

```
> Time measurements:
                 Total  Self
main()           30.5s  0.9s
:-gen_matrix()   8.3s   2.0s
: :
: `-comp_idx()   5.3s   5.3s
:
:-distribute()   5.9s   5.9s
:
...
```

Another plus is brought along by combining related measurements or information, and thus enabling additional concepts. For example, combining the time measurements we already mentioned and the call tree information, enable the simple but powerful concept of *inclusive/exclusive execution time*. The inclusive time is the time spent from entering a function until returning. The exclusive time is the time spent in the body of the function alone, also called "self-time". The call tree gives the information about which functions were called from inside a specific function. The exclusive time for this function is given by subtracting the time spent in this calls (inclusive times) from the own inclusive time.

Graphical visualisation

All these types of information - raw, processed, correlated - build up the *quantitative* and *objective* support for the performance analysis: fixed values, numbers and mathematical relations. This might be enough for the few users which are very skilled and experienced, but the majority users would benefit a lot if this was backed up with a *qualitative* and rather *subjective* support: colors, space perception and data navigation techniques. Graphical visualisation ease and enrich the analysis process.

One good example in this sense is given by the *timeline* view of the time measurements. Function or region groups receive a color coding which can be recognized and qualitatively evaluated on the timeline of the entire execution. Further investigations on shorter execution intervals are possible by simply selecting the specific interval on the timeline.

Other graphical displays and views used by current tools range from common diagrams to processor topology or color-coded table of exchanged messages. A thorough summary of visualisation techniques could be found in [55].

2.3.2 *Analysis*

Even the very fact that there are so many visualisation approaches in usage today leads us to the thought that performance analysis is not a straightforward job to do. This is true indeed and it is also one reason why there was only little progress in *automatic optimization* of HPC applications up to now.

The complexity of performance analysis is manifold. On the one hand, the measured performance values have to be given *a meaning*, they have to be associated with specific performance properties. It is mostly the case that these are not one-to-one mappings: different hardware counters can be influenced by the same performance aspect and, again, different performance issues add up to alter the value of a single counter.

On the other hand, performance measurements have to be interpreted according to the platform/architecture/machine on which the application was ran. Even for similar measured values, due to different system configurations, poor performance numbers on a particular machine might be acceptable performance for another one.

And then finally everything depends on the application itself. Having drawn some conclusions regarding the behaviour of the application on a specific machine, it is still a challenge to conclude upon the general performance and then to suggest possible improvements. In the HPC context one has to get beyond the already common optimization hints which focus basically on the source code level like loop unrolling or appropriate data localization. Optimization hints for HPC applications have to take into consideration higher level issues too, like the parallelization approach used or execution phases.

2.4 EXISTING TOOLS

At the time of writing this thesis there is only a handful of well-known profilers and tracers which are used in the supercomputing centres. We shortly present in the following what we consider to be their most relevant features in the context of this work. Our intention is to give an overview from a high-level point of view of a common user, rather than technical and implementation details interesting for tool developers.

2.4.1 *Vampir and Intel Trace Analyser*

Intel® Trace Analyzer and Collector

VAMPIR

Often seen as two close versions of the same product, Vampir and Intel Trace Analyser (ITA) share indeed the same roots. Currently they are being developed on independent tracks, though: Vampir at the *Center for Information Services and High Performance Computing (ZIH)* of TU Dresden and ITA by Intel.

Both are very mature projects offering well-proven solutions for performance analysis. Vampir reads *Open Trace Format (OTF)* files which can be generated by tools like TAU [80] or VampirTrace [84]. ITA uses instead the *Structured Trace Format (STF)*, which is a proprietary format owned by Intel. ITA is used in combination with Intel Trace Collector (ITC) [32], which generates STF traces.

The most distinguishing feature of both ITA and Vampir is probably the *timeline view*. As described above, this is a great means to investigate the acquired performance measurements. Moreover, these tools add along the timeline analysis also other secondary, synchronized views, which display information like hardware counters or exchanged messages size for the time interval selected on the timeline. See figure 3 for an explanatory screenshot.

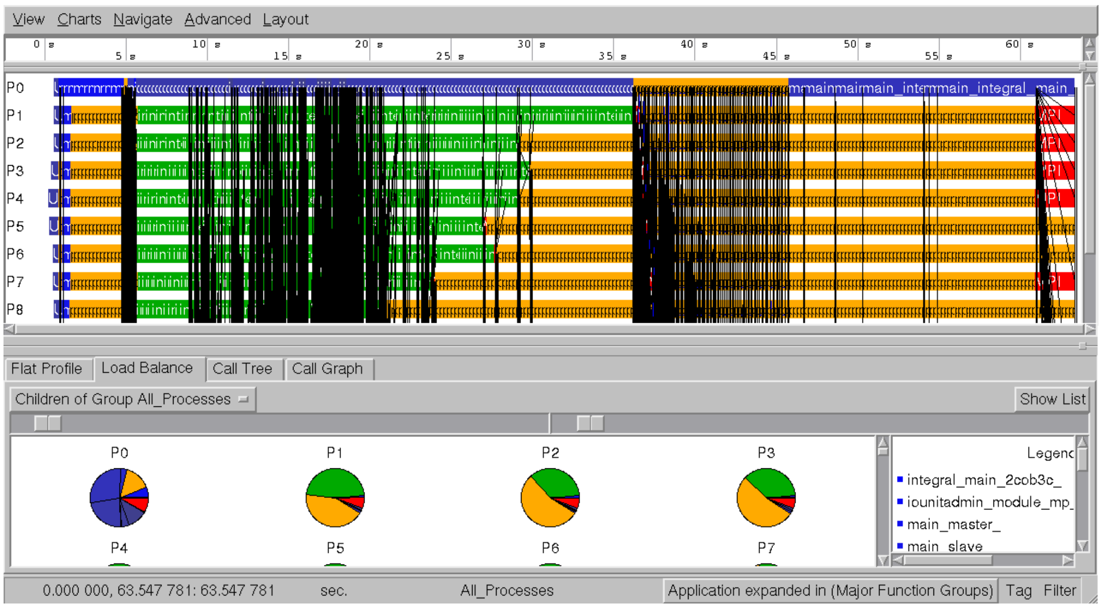

View Charts Navigate Advanced Layout
P0
P1
P2
P3
P4
P5
P6
P7
P8
Flat Profile Load Balance Call Tree Call Graph
Children of Group All_Processes
Show List
Legend
integral_main_2cob3c_
iounitadmin_module_mp_
main_master_
main_slave
0.000 000, 63.547 781: 63.547 781 sec. All_Processes
Application expanded in (Major Function Groups) Tag Filter

Figure 3: Screenshot of a trace analysis with Intel Trace Analyzer. On the top, the timeline view with 8 processes visible. On the bottom, the pie diagrams for the function execution times per process - 4 processes visible in the screenshot. The black lines between processes on the timeline represent groups of MPI messages.

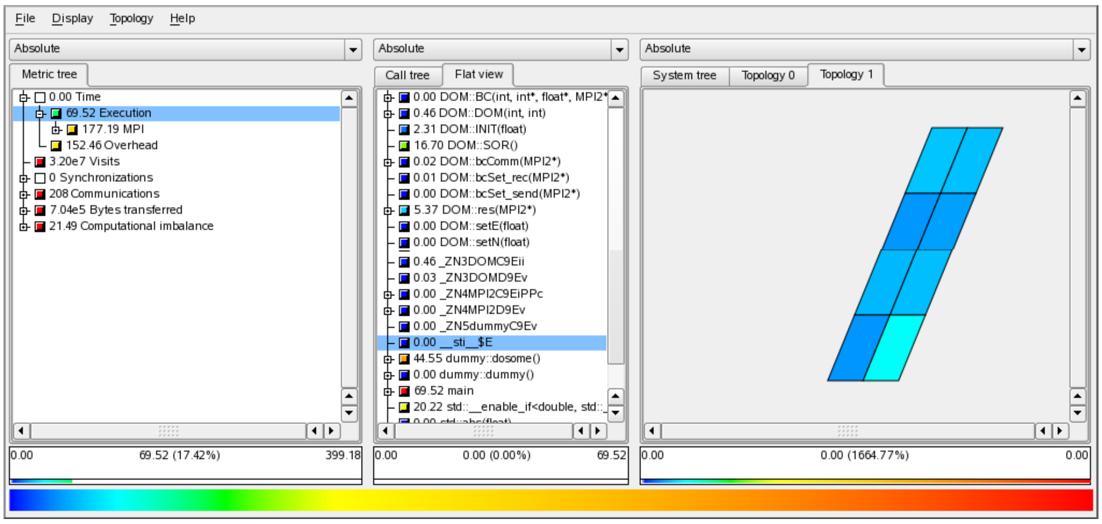
2.4.2 *Scalasca*

scalasca

Another very mature performance tool is Scalasca [69]. This is an open-source joint project of the Jülich Supercomputing Centre from the Forschungszentrum Jülich and the Laboratory for Parallel Programming from the German Research School for Simulation Sciences.

Scalasca offers a complete solution for performance measurement and analysis, from the instrumentation phase, to the visualisation. A specific feature for Scalasca is the automatic analyser for possible inefficient communication patterns. Cases of *late senders*, for example, can be detected post-mortem from the trace files automatically, thus being of grate help in analysing large scale applications with huge performance traces.

While ITA and Vampir use the *timeline* as the guiding mechanism for navigating through the traces, Scalasca adopted another approach for exploration. The focus is on the measured metrics. One has to choose a metric from a metrics tree and then the values for this metrics are shown in the call-path tree. It is thus a mapping from the measurements to the location in the source code, whereas in ITA and Vampir one looks first for the location and then to the metrics values. See figure 4 for an explanatory screenshot of Scalasca.



```
0.00 Time
  69.52 Execution
    177.19 MPI
    152.46 Overhead
3.20e7 Visits
0 Synchronizations
208 Communications
7.04e5 Bytes transferred
21.49 Computational imbalance
```

```
0.00 DOM::BC(int, int*, float*, MPI2*
0.46 DOM::DOM(int, int)
2.31 DOM::INIT(float)
16.70 DOM::SOR()
0.02 DOM::bcComm(MPI2*)
0.01 DOM::bcSet_rec(MPI2*)
0.00 DOM::bcSet_send(MPI2*)
5.37 DOM::res(MPI2*)
0.00 DOM::setE(float)
0.00 DOM::setN(float)
0.46 _ZN3DOMC9Eii
0.03 _ZN3DOMD9Ev
0.00 _ZN4MPI2C9EiPPc
0.00 _ZN4MPI2D9Ev
0.00 _ZN5dummyC9Ev
0.00 __sti__$E
44.55 dummy::dosome()
0.00 dummy::dummy()
69.52 main
20.22 std::__enable_if<double, std::_
```

```
0.00        69.52 (17.42%)     399.18
0.00        0.00 (0.00%)        69.52
0.00        0.00 (1664.77%)      0.00
```

Figure 4: Screenshot of the Scalasca visualisation window. The information is being displayed in three fixed views, arranged in three columns. On the left most column the user can choose the metric to explore. Upon selecting an element in the left column, the information in the middle column adjusts to show the values for the metric as per function call. Finally, on selecting a function call, the distribution over processes and threads can be seen on the right column.

2.4.3 *TAU*

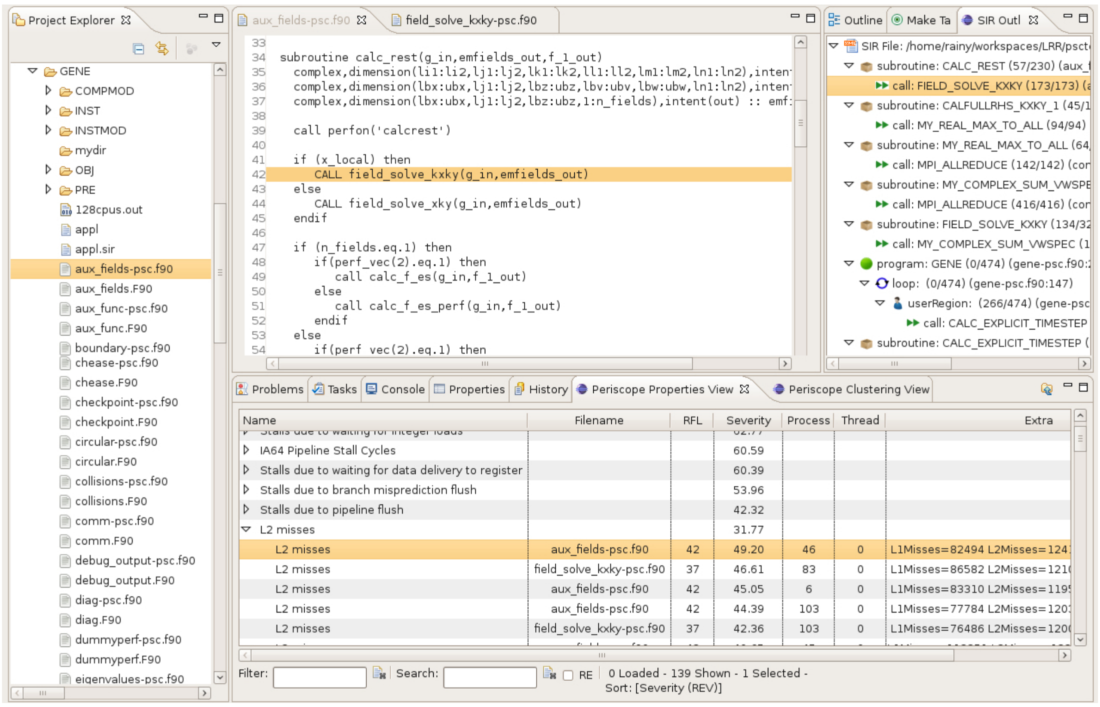
A somehow different software approach was adopted by the TAU [80] project. Instead of delivering *one solution* for the complete profiling flow, TAU (Tuning and Analysis Utilities) offers an entire *toolkit*, a framework which is able to make use of separate tools to accomplish the different phases of the flow. Noticeable is that late versions of TAU also offer integration of Vampir to be used as an analyser and visualiser for the generated tracefiles.

One feature supported by the TAU framework is the integration of runtime instrumentation using the Dyninst [16] implementation. Another interesting feature is the 3D visualisation of performance measurements with ParaProf [72].

2.4.4 *Periscope*

Periscope [18] [19] is a freely available performance analysis tool, currently being developed at TU München. It distinguishes itself from other tools through the ability to perform automatic online performance analysis. Following predefined strategies, a hierarchy of analysis agents carry out on-the-fly analysis of performance and adapt their measurement parameters online, or even re-run critical regions of the code when more detailed measurements are necessary.

The guiding navigation mechanism through the performance data is neither the time, nor the metrics, but the *severity* of the identified issues. See figure 5 and caption for an explanatory screenshot.



```
subroutine calc_rest(g_in,emfields_out,f_1_out)
  complex,dimension(li1:li2,lj1:lj2,lk1:lk2,ll1:ll2,lm1:lm2,ln1:ln2),inten
  complex,dimension(lbx:ubx,lj1:lj2,lbz:ubz,lbv:ubv,lbw:ubw,ln1:ln2),inten
  complex,dimension(lbx:ubx,lj1:lj2,lbz:ubz,1:n_fields),intent(out) :: emf:

  call perfon('calcrest')

  if (x_local) then
     CALL field_solve_kxky(g_in,emfields_out)
  else
     CALL field_solve_xky(g_in,emfields_out)
  endif

  if (n_fields.eq.1) then
     if(perf_vec(2).eq.1) then
        call calc_f_es(g_in,f_1_out)
     else
        call calc_f_es_perf(g_in,f_1_out)
     endif
  else
     if(perf vec(2).eq.1) then
```

```
SIR File: /home/rainy/workspaces/LRR/psct
  subroutine: CALC_REST (57/230) (aux_
      call: FIELD_SOLVE_KXKY (173/173) (
  subroutine: CALFULLRHS_KXKY_1 (45/1
      call: MY_REAL_MAX_TO_ALL (94/94)
  subroutine: MY_REAL_MAX_TO_ALL (64
      call: MPI_ALLREDUCE (142/142) (cor
  subroutine: MY_COMPLEX_SUM_VWSPE
      call: MPI_ALLREDUCE (416/416) (cor
  subroutine: FIELD_SOLVE_KXKY (134/32
      call: MY_COMPLEX_SUM_VWSPEC (1
  program: GENE (0/474) (gene-psc.f90:
    loop:  (0/474) (gene-psc.f90:147)
      userRegion:  (266/474) (gene-psc
          call: CALC_EXPLICIT_TIMESTEP
  subroutine: CALC_EXPLICIT_TIMESTEP (
```

Name	Filename	RFL	Severity	Process	Thread	Extra
IA64 Pipeline Stall Cycles			60.59			
Stalls due to waiting for data delivery to register			60.39			
Stalls due to branch misprediction flush			53.96			
Stalls due to pipeline flush			42.32			
L2 misses			31.77			
L2 misses	aux_fields-psc.f90	42	49.20	46	0	L1Misses=82494 L2Misses=124
L2 misses	field_solve_kxky-psc.f90	37	46.61	83	0	L1Misses=86582 L2Misses=121
L2 misses	aux_fields-psc.f90	42	45.05	6	0	L1Misses=83310 L2Misses=119
L2 misses	aux_fields-psc.f90	42	44.39	103	0	L1Misses=77784 L2Misses=120
L2 misses	field_solve_kxky-psc.f90	37	42.36	103	0	L1Misses=76486 L2Misses=120

Filter: Search: RE 0 Loaded - 139 Shown - 1 Selected -
Sort: [Severity (REV)]

Figure 5: Screenshot of the Periscope plug-in for Eclipse. Performance issues are displayed in a table in the bottom view, together with the computed severity. Upon choosing a specific item in the table, the exact source of the problem is being highlighted in the source code view in the middle. In the right view, the special "src" configuration file offers an overview of all instrumented regions.

2.4.5 *Score-P*

Score-P [71] was a successful initiative aiming at building a system which delivers some standardized support for performance optimization. The results of the Score-P project is a measurement infrastructure which offers a common backend integrating all tools mentioned above: Vampir, Scalasca, TAU and Periscope.

2.4.6 *HPCToolkit*

HPCToolkit [31], as the name recommends it, is an integrated suite of tools for measurement and analysis of application performance. It is similar in this sense to TAU. Due to the fact that it uses sampling methods for measurements acquisition, HPC-Toolkit has a very low measurement overhead. It offers two visualisation tools, specialized each one for either profiling views or trace views. The first one uses as main navigation mechanism the callpath of the execution, while the second one is based on timeline views.

2.4.7 *Other projects*

Besides the already mentioned projects, there are also several smaller initiatives or tools which either provide only very specific solutions, or did not developed to a mature product.

First to be mentioned here is the *Dyninst* [16] project which offers a library for runtime code patching. Especially interesting for performance measurements, Dyninst API provides the possibility to insert instrumentation into the binary of an application. It is currently being developed within the Paradyn team of the University of Wisconsin and University of Maryland.

Paradyn [56] is the performance measurement tool developed inside the group which is providing Dyninst. A special feature of Paradyn is the *Performance Controller* which automatically checks for predefined performance bottlenecks patterns and informs the user on potential problems.

Another tool worth to be mentioned here, and which is also based on Dyninst, is *MATE* (Measurement, Analysis and Tuning Environment) [45] from the Barcelona Supercomputing Center. MATE offers a framework for automatic performance optimization. Users define *tunelets* - performance models and possible optimization decisions - for their applications and MATE carries out performance measurements and adapts at runtime the tasks of the application based on the tunelets.

3 ADDRESSING DOMAIN ASPECTS IN APPLICATIONS/*COMPUTER SCIENCE*

There are several research and application fields within computer science, which overlap with the Domain Knowledge area. As we can only loosely define the overlapping borders, and because the contributions come from so different directions, we do not intend in the following to give an exhaustive survey of all related work carried out so far. We rather aim at providing the reader only with an overview of the main results and projects relevant to this thesis.

We consider works from *Computational Science and Engineering (CSE)*, *Programming*, *Software Engineering (SE)*, and *High Performance Computing (HPC)*. The diagram in figure 6 shows these fields and their connections to Domain Knowledge. Although not part of Computer Science, we also included *Natural Sciences* in the diagram, as being an important factor for at least two of referenced topics. One could argue that the overlapping between Natural Sciences and Domain Knowledge (DK) should be broader, as the *knowledge* is *from*, or *inside* the sciences; yet, we consider DK as a holistic approach which serves as a bridge between the different fields.

Given their approach with respect to the concepts of *domain* and *application*, the research topics we consider below subscribe to one of the two cases depicted in figure 7:

1. the domain is perceived as *outside* the application

2. the domain is perceived as *within* the application

The sections 3.1 to 3.5 focus on the same general aspect, namely that of supplying applications with domain specific information. Issues like how to identify and model domain information at the design level, how to express and encode it through the implementation process, or how to embed it alongside the source code, all belong to the case depicted in figure 7a. Here the domain is perceived as being *outside*, *external to* the application. Thus, the focus is set on transferring/mapping it *towards* the application. First of all, different constructs and aspects are identified inside the domain - represented here by the red contours inside the blue form. Afterwards, by means of different actions such as marking, formatting, or referencing, these fragments of domain information are translated towards the application and, in the end, an application with domain specific information is obtained. An important notice is that the initial domain is herewith only partially and more or less inaccurate imaged within the application.

In the last section of this chapter, section 3.6.1, we focus on the research directions which map to the second setting, depicted in figure 7b. The focus drops here firstly on the implemented application, unlike the previous case, where the first concern is the domain. Potential domain aspects are perceived *within* the application itself - the dotted blue contours on the top-right form - and hence the identification of the domain constructs is strongly depending on the application. This also leads to the

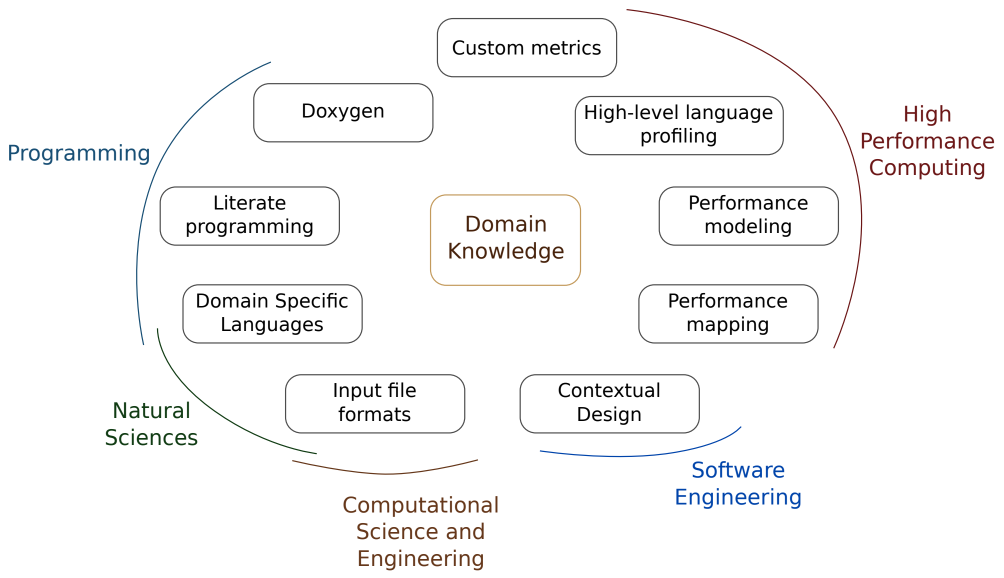


Figure 6: Research and application fields which relate to Domain Knowledge and their connecting points.

concepts being only roughly outlined. More explanations on the representation on figure 7b are provided in section 3.6.1.

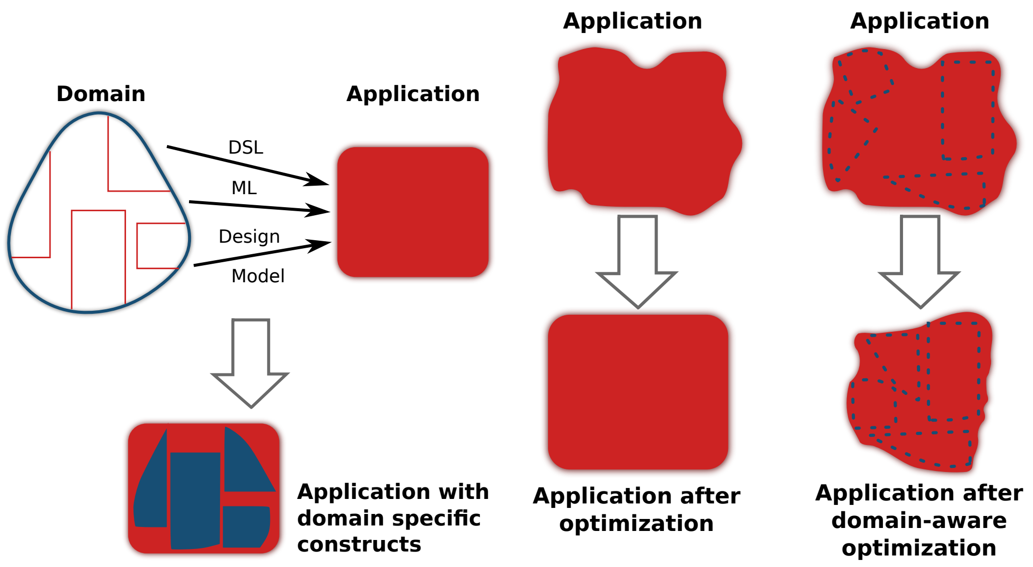
In the following, we present the projects from the point of view of their contributions to the general "picture" of Domain Knowledge, and leave the discussions regarding their relation and influence on our own work for chapter 4, when presenting our own approach.

3.1 CONTEXTUAL DESIGN

It might be considered as a true *outsider* among the other related projects we take into account for this overview, but *Contextual Design* has an important contribution in understanding Domain Knowledge.

Introduced in the mid-90's by Beyer and Holtzblatt in [8], *Contextual Design* is a software design process which targets great usability of designed systems by focusing on understanding the needs and requirements of the future users. This idea actually lies at the foundation of the entire *User-centered design (UCD)*. What is special about Contextual Design though, is that designers are brought *into* their users' real work environment, they are observing users in their *"context"*. Also, in the interpretation and consolidation steps which follow this first step of information gathering, the "contextual" component is being preserved by teaching the designers to extract and use in their diagrams the same technical term they have learned from their users while interviewing and observing them.

Apart from being an appealing design strategy to apply to the current profiling tools, Contextual Design also offers an insightful parallel for the Domain Knowledge based performance optimization: user context influences and is present in the entire design process, the same way as domain knowledge influences and is present, for example, in the optimization process.


Domain
Application
DSL
ML
Design
Model
Application with domain specific constructs
Application
Application after optimization
Application
Application after domain-aware optimization

(a) domain perceived outside the application results in a kind of movement from domain towards the application

(b) domain perceived as within/along the application, hence there are adjustment actions needed

Figure 7: Two approaches for the relation between domain and application.

The drawback of the above described method in Software Engineering, consists in the difficulty to actually produce structural diagrams directly from the contextual information. The designers lack the means to express the knowledge, especially if not used with the type of diagrams. This is similar to another issue encountered in computer science: how to encapsulate more information into an application than the sheer source code written out for instructing the machine what to execute.

How to express an application beyond the programming language?

3.2 LITERATE PROGRAMMING

One seminal proposal was made back in 1984 by Knuth when he coined the term *Literate Programming (LP)* in his paper [37]. The main idea was that programmers should not write code just for computers, but they should write code for humans. In order to do so, *Literate Programming* emphasized the importance of a good documentation of an application *alongside* the source code. Thus, developers should actually write their programs as if their were explaining them to another person. The documentation should be more like an essay about the solution being implemented, while the source code text should be written in the order which makes sense for the flow of a logical explanation and not in the flow of the execution. The *WEB*[1] system, which was the original system implementing the literate programming methodology, is a combination of a document typesetting system and a source-to-source processor. The typesetting was being taken care of by a tool called *weave*, based on the TEX system, while the application code generation was achieved by a second tool called *tangle*, which originally produced PASCAL code (see also figure 8).

1 This was before the World Wide Web appeared and it referred to exactly what it means in English, a woven piece of material

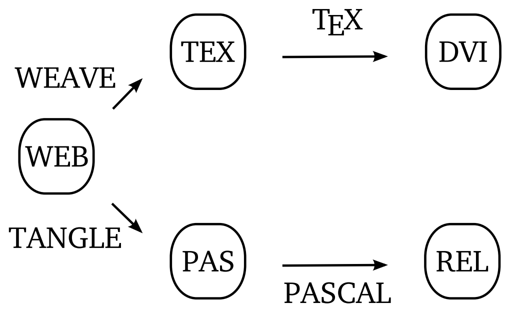


Figure 8: Dual usage of a WEB file. Source: [37].

As opposed to TeX, which soon became popular and today is *the* typesetting system to be used in scientific papers, *Literate Programming* did not enjoy the same acceptance. There were indeed several programming environments developed to support LP - Pieterse, Kourie, and Boake give a good summary of LPEs (Literate Programming Environments) in [59] -, but it never advanced to become a standard or common programming paradigm. In the recent years, though, one could observe a fine movement towards revitalizing LP through works like [60], [29], or [61]. They advocate for LP given today's application development context, especially in the computational science fields, like that of quantum chemistry or molecular physics. Furthermore, Palmer and Hillenbrand propose in [54] a new language for writing literate programs. *Ginger* can represent both code and prose, as well as the literate connections needed in the original *web*s. Also, Schulte et al. present in [70] a working extension to *Emacs* [20], not only to support LP, but actually to enhance it with new possibilities for documentation, like project management details and diagrams or tables with execution results.

3.3 DOCUMENTATION EXTRACTION

If *Literate Programming* was a first trial to get more knowledge *into* the code, then the actual successful strategy, in terms of user acceptance, proved to be almost the opposite one, namely getting the knowledge *out of* the source code itself. This is what *documentation extraction* systems provide, at different levels of detail, automation and usability.

The most widespread documentation systems to date are those respective to the most widespread active programming languages: on the one side, there is the Javadoc [53] native for Java language, and on the other side, there is Doxygen [25] which was native for C/C++ languages. Nowadays Doxygen also supports the Javadoc notations and it can be used with a considerable number of other programming languages [25].

The main idea behind *documentation extraction* is very simple: encapsulate the knowledge about the application inside comments with predefined format and then parse the source code to extract the comments and generate documentation files. Very useful is that the generated documentation files are provided nowadays in many formats. Such are HTML, LaTeX, hyperlinked PDF, RTF (MS-Word), and Unix man pages. Also very welcome is the integration of document extraction systems within different IDEs, in order to display documentation as context information while writing source code.

DSL	Application Domain
BNF	Syntax specification
Excel	Spreadsheets
HTML	Hypertext web pages
LaTeX	Typesetting
Make	Software building
MATLAB	Technical computing
SQL	Database queries
VHDL	Hardware design

Table 1: Some Widely Used Domain-Specific Languages. (Table adapted from [44])

An even more advanced feature is the extraction of the documentation *from within* the code text. That means that given a source file, the system will try to generate the appropriate documentation based on a complex analysis of function and variable names, function prototype, variable types and code statements [77].

3.4 DOMAIN SPECIFIC LANGUAGES

Another successful approach to binding together knowledge and applications are the *Domain Specific Languages (DSL)*. Mentioned already in the late 60's as *application-oriented languages* in [67], DSLs have experienced an impressive growth, as nowadays there are probably thousands of such languages. Deursen, Klint, and Visser [15] define DSLs as follows: *A domain-specific language is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.*[51]

A list of some of the most widely used DSLs to date, along with their application domains is given in table 1.

The important characteristic which all DSLs share is their *expressiveness*. As shown in [44] and later in [51], this main characteristic results from the fact that DSLs are rather *declarative* and *descriptive* languages, as opposed to the common General Purpose Languages (GPL), which are imperative languages.

Of special interest for our study on Domain Knowledge, are those DSLs belonging to the particular class, for which *"domain"* represents one of the computational or natural science domains.

We give in the following two examples of projects from the Quantum Chemistry field and a third one from the broader scientific computing area.

3.4.1 *ACES (Advanced Concepts in Electronic Structure) III*

One mature and, in the same time, complex project is ACES III [41]. It offers a complete framework for developing simulations based on the electronic structure theory. The main idea, upon which the entire system is built on, is that each computational chemistry program can be expressed in terms of *Super Instructions* operating on *Su-*

```
pardo M,N,I,J
        tmpsum(M,N,I,J) = 0.0
        do L
                do S
                        get T(L,S,I,J)
                        compute_integrals V(M,N,L,S)
                        tmp(M,N,I,J) =
                                V(M,N,I,J) * T(L,S,I,J)
                        tmpsum(M,N,I,J) += tmp(M,N,I,J)
                enddo S
        enddo L
        put R(M,N,I,J) = tmpsum(M,N,I,J)
endpardo M,N,I,J
```

Listing 1: A SIAL code fragment [68]

per Numbers, also called blocks [68]. The programming model resembles the parallel task-based model, where each worker executes a set of instructions on a given chunk of data. The important difference is that in ACES III on the one hand, the parallelism details are transparent to the user and, on the other hand, in the implementation of the parallel methods the specific domain particularities are being taken into consideration. The is is achieved by combining a DSL with a virtual machine:

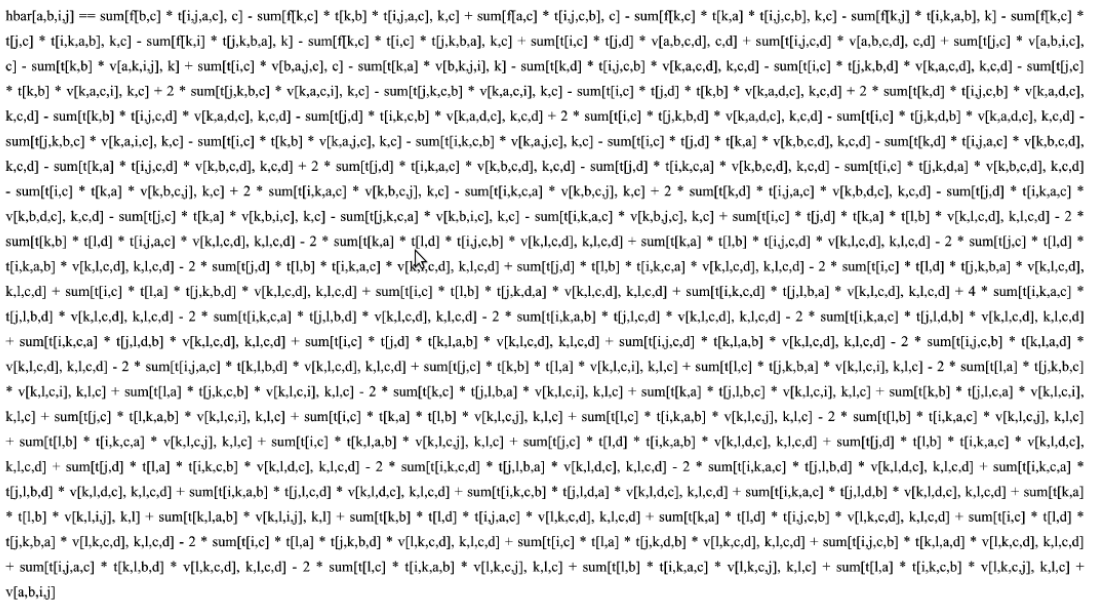
1. the user writes the code in the Domain-Specific Language called SIAL (Super Instruction Architecture Language). See listing 1;
2. the code is compiled to SIA bytecode;
3. the bytecode is executed by SIP (Super Instruction Processor), a parallel virtual machine which manages all details related to the parallel, optimized execution.

The most important features of SIAL, as presented recently in [14] and [68], are:

- the particular types of data structures, like AO (Atomic Orbitals) indices, occupied MO (Molecular Orbitals) and unoccupied MO indices;
- intrinsic distributed arrays;
- `PARDO` - an explicit parallel execution controlling statement.

SIAL also offers intrinsic super instructions which express common operations needed when writing quantum chemistry code, like a block contraction operator, for example. Moreover, ACES III comes with several SIAL implementations of the most common methods, such as HT (Hartree-Fock), CCSD (Coupled Cluster with Single and Double excitations) or MP2 (Second Order Møller-Plesset Perturbation Theory.

A particular detail of the SIP processor, which is of importance in the current context, is its support for performance measurement [4]. Using a tracing mechanism, SIP offers the user the possibility to query timers recording super instructions execution or even timers monitoring the MPI traffic. Due to SIP being targeted for executing programs resulting specifically from SIAL code, tracing information can be mapped back up to the line number of the initial source code. This is a known challenge for



```
hbar[a,b,i,j] == sum[f[b,c] * t[i,j,a,c], c] - sum[f[k,c] * t[k,b] * t[i,j,a,c], k,c] + sum[f[a,c] * t[i,j,c,b], c] - sum[f[k,c] * t[k,a] * t[i,j,c,b], k,c] - sum[f[k,j] * t[i,k,a,b], k] - sum[f[k,c] *
t[j,c] * t[i,k,a,b], k,c] - sum[f[k,i] * t[j,k,b,a], k] - sum[f[k,c] * t[i,c] * t[j,k,b,a], k,c] + sum[t[i,c] * t[j,d] * v[a,b,c,d], c,d] + sum[t[i,j,c,d] * v[a,b,c,d], c,d] + sum[t[j,c] * v[a,b,i,c],
c] - sum[t[k,b] * v[a,k,i,j], k] + sum[t[i,c] * v[b,a,j,c], c] - sum[t[k,a] * v[b,k,j,i], k] - sum[t[k,d] * t[i,j,c,b] * v[k,a,c,d], k,c,d] - sum[t[i,c] * t[j,k,b,d] * v[k,a,c,d], k,c,d] - sum[t[j,c]
* t[k,b] * v[k,a,c,i], k,c] + 2 * sum[t[j,k,b,c] * v[k,a,c,i], k,c] - sum[t[j,k,c,b] * v[k,a,c,i], k,c] - sum[t[i,c] * t[j,d] * t[k,b] * v[k,a,d,c], k,c,d] + 2 * sum[t[k,d] * t[i,j,c,b] * v[k,a,d,c],
k,c,d] - sum[t[k,b] * t[i,j,c,d] * v[k,a,d,c], k,c,d] - sum[t[j,d] * t[i,k,c,b] * v[k,a,d,c], k,c,d] + 2 * sum[t[i,c] * t[j,k,b,d] * v[k,a,d,c], k,c,d] - sum[t[i,c] * t[j,k,d,b] * v[k,a,d,c], k,c,d] -
sum[t[j,k,b,c] * v[k,a,i,c], k,c] - sum[t[i,c] * t[k,b] * v[k,a,j,c], k,c] - sum[t[i,k,c,b] * v[k,a,j,c], k,c] - sum[t[i,c] * t[j,d] * t[k,a] * v[k,b,c,d], k,c,d] - sum[t[k,d] * t[i,j,a,c] * v[k,b,c,d],
k,c,d] - sum[t[k,a] * t[i,j,c,d] * v[k,b,c,d], k,c,d] + 2 * sum[t[j,d] * t[i,k,a,c] * v[k,b,c,d], k,c,d] - sum[t[j,d] * t[i,k,c,a] * v[k,b,c,d], k,c,d] - sum[t[i,c] * t[j,k,d,a] * v[k,b,c,d], k,c,d]
- sum[t[i,c] * t[k,a] * v[k,b,c,j], k,c] + 2 * sum[t[i,k,a,c] * v[k,b,c,j], k,c] - sum[t[i,k,c,a] * v[k,b,c,j], k,c] + 2 * sum[t[k,d] * t[i,j,a,c] * v[k,b,d,c], k,c,d] - sum[t[j,d] * t[i,k,a,c] *
v[k,b,d,c], k,c,d] - sum[t[j,c] * t[k,a] * v[k,b,i,c], k,c] - sum[t[j,k,c,a] * v[k,b,i,c], k,c] - sum[t[i,k,a,c] * v[k,b,j,c], k,c] + sum[t[i,c] * t[j,d] * t[k,a] * t[l,b] * v[k,l,c,d], k,l,c,d] - 2 *
sum[t[k,b] * t[l,d] * t[i,j,a,c] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[k,a] * t[l,d] * t[i,j,c,b] * v[k,l,c,d], k,l,c,d] + sum[t[k,a] * t[l,b] * t[i,j,c,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[j,c] * t[l,d] *
t[i,k,a,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[j,d] * t[l,b] * t[i,k,a,c] * v[k,l,c,d], k,l,c,d] + sum[t[j,d] * t[l,b] * t[i,k,c,a] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,c] * t[l,d] * t[j,k,b,a] * v[k,l,c,d],
k,l,c,d] + sum[t[i,c] * t[l,a] * t[j,k,b,d] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[l,b] * t[j,k,d,a] * v[k,l,c,d], k,l,c,d] + sum[t[i,k,c,d] * t[j,l,b,a] * v[k,l,c,d], k,l,c,d] + 4 * sum[t[i,k,a,c] *
t[j,l,b,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,c,a] * t[j,l,b,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,a,b] * t[j,l,c,d] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,k,a,c] * t[j,l,d,b] * v[k,l,c,d], k,l,c,d]
+ sum[t[i,k,c,a] * t[j,l,d,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,c] * t[j,d] * t[k,l,a,b] * v[k,l,c,d], k,l,c,d] + sum[t[i,j,c,d] * t[k,l,a,b] * v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,j,c,b] * t[k,l,a,d] *
v[k,l,c,d], k,l,c,d] - 2 * sum[t[i,j,a,c] * t[k,l,b,d] * v[k,l,c,d], k,l,c,d] + sum[t[j,c] * t[k,b] * t[l,a] * v[k,l,c,i], k,l,c] + sum[t[l,c] * t[j,k,b,a] * v[k,l,c,i], k,l,c] - 2 * sum[t[l,a] * t[j,k,b,c]
* v[k,l,c,i], k,l,c] + sum[t[l,a] * t[j,k,c,b] * v[k,l,c,i], k,l,c] - 2 * sum[t[k,c] * t[j,l,b,a] * v[k,l,c,i], k,l,c] + sum[t[k,a] * t[j,l,b,c] * v[k,l,c,i], k,l,c] + sum[t[k,b] * t[j,l,c,a] * v[k,l,c,i],
k,l,c] + sum[t[j,c] * t[l,k,a,b] * v[k,l,c,i], k,l,c] + sum[t[i,c] * t[k,a] * t[l,b] * v[k,l,c,j], k,l,c] + sum[t[l,c] * t[i,k,a,b] * v[k,l,c,j], k,l,c] - 2 * sum[t[l,b] * t[i,k,a,c] * v[k,l,c,j], k,l,c]
+ sum[t[l,b] * t[i,k,c,a] * v[k,l,c,j], k,l,c] + sum[t[i,c] * t[k,l,a,b] * v[k,l,c,j], k,l,c] + sum[t[j,c] * t[l,d] * t[i,k,a,b] * v[k,l,d,c], k,l,c,d] + sum[t[j,d] * t[l,b] * t[i,k,a,c] * v[k,l,d,c],
k,l,c,d] + sum[t[j,d] * t[l,a] * t[i,k,c,b] * v[k,l,d,c], k,l,c,d] - 2 * sum[t[i,k,c,d] * t[j,l,b,a] * v[k,l,d,c], k,l,c,d] - 2 * sum[t[i,k,a,c] * t[j,l,b,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,c,a] *
t[j,l,b,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,a,b] * t[j,l,c,d] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,c,b] * t[j,l,d,a] * v[k,l,d,c], k,l,c,d] + sum[t[i,k,a,c] * t[j,l,d,b] * v[k,l,d,c], k,l,c,d] + sum[t[k,a]
* t[l,b] * v[k,l,i,j], k,l] + sum[t[k,l,a,b] * v[k,l,i,j], k,l] + sum[t[k,b] * t[l,d] * t[i,j,a,c] * v[l,k,c,d], k,l,c,d] + sum[t[k,a] * t[l,d] * t[i,j,c,b] * v[l,k,c,d], k,l,c,d] + sum[t[i,c] * t[l,d] *
t[j,k,b,a] * v[l,k,c,d], k,l,c,d] - 2 * sum[t[i,c] * t[l,a] * t[j,k,b,d] * v[l,k,c,d], k,l,c,d] + sum[t[i,c] * t[l,a] * t[j,k,d,b] * v[l,k,c,d], k,l,c,d] + sum[t[i,j,c,b] * t[k,l,a,d] * v[l,k,c,d], k,l,c,d]
+ sum[t[i,j,a,c] * t[k,l,b,d] * v[l,k,c,d], k,l,c,d] - 2 * sum[t[l,c] * t[i,k,a,b] * v[l,k,c,j], k,l,c] + sum[t[l,b] * t[i,k,a,c] * v[l,k,c,j], k,l,c] + sum[t[l,a] * t[i,k,c,b] * v[l,k,c,j], k,l,c] +
v[a,b,i,j]
```

Figure 9: CCSD doubles expression from quantum chemistry [5].

High-Level-Languages based on GPLs: through the process of compiling first high-level constructs into GPL constructs and then the GPL code to final binary, while in the latter several compiler optimizations are usually being applied to the GPL source code, only less of the initial source code layout is being preserved. Hence, mapping the performance information to the exact line number or instruction turns out to be very cumbersome, almost impossible.

3.4.2 *TCE (Tensor Contraction Engine)*

Another example of a DSL for quantum chemistry is the TCE (Tensor Contraction Engine) [3], mainly known as a module of the NWChem software[2]. Unlike ACES III, which addresses a broad range of computational chemistry methods and problems, TCE is targeted towards a very particular aspect, namely the evaluation of tensor contraction expressions. These expressions are very often encountered in quantum chemistry and general many-body problems, as for example in the "golden method" of computational chemistry [68], the CCSD. *A tensor contraction is essentially a collection of multi-dimensional summations of the product of several input arrays* [5], which for complex systems might get really hard to manage manually (see figure 9 for an example).

TCE offers a means to easily declare operators using a High-Level Language, which qualifies as a DSL for quantum chemistry (see listing 2 for an example). After parsing the code, the engine undergoes several stages to synthesize the final expression of the tensor contraction [5][3]:

1. algebraic transformations: use algebraic properties like commutativity, associativity and distributivity, to generate an equivalent expression with minimal operation cost;

2 NWChem [82] is a known software package, widely used in the computational chemistry field.

```
range N = 3100;
range V = 3000;
range O = 100;

index la,mu,nu,om : N;
index a,b,c,d,e,f : V;
index i,j,k : O;

mlimit = 100GB;

function F(N,N,N,N);

procedure P(in Co[N,O], in Cv[N,V], in T[O,O,V,V], out E)=
begin
   E==sum[sum[sum[sum[sum[sum[F(mu,nu,om,la)*Co[la,k],{la}]*
         Cv[om,b],{om}]*Cv[nu,f],{nu}]*Cv[mu,a],{mu}]
         *
         sum[sum[sum[sum[F(mu,nu,om,la)*Co[la,k],{la}]*Cv[om,b],
         {om}]*Cv[nu,e],{nu}]*Cv[mu,c],{mu}], {b,k}]
         *
         sum[T[i,j,a,e]*T[i,j,c,f],{i,j}],
         {a,e,c,f}];
end
```

Listing 2: A TCE code fragment [5].

2. memory minimization: annotate the expression tree with information about best loop fusion[3] options;
3. parallel optimizations: apply common optimization strategies like efficient data distribution and partitioning - to minimize parallel communication -, or space-time transformation - for the best trade-off between storing and recomputing data;
4. code generation: TCE generates Fortran code which implements the optimized tensor contraction expression, using the Global Array Toolkit [48] for parallel computing. Recent work also targets generating CUDA code, in order for TCE to be able to address heterogeneous systems [39].

3.4.3 *Broadway and Telescoping Languages*

A different approach to the pure DSLs presented above has been followed by two similar projects: the Broadway Compiler, introduced by Guyer [21], and the Telescoping Languages, introduced by Kennedy et al. [36]. Although there is no recent activity for either of the projects, they still provide a good reference for how domain knowledge can be fostered within applications.

3 Loop fusion is nowadays a common compiler optimization step. It essentially joins two or more loops which run over common indices, mostly to spare the memory used otherwise for storing the intermediate results exchanged between the loops.

The idea behind was to make use of the information which could be added to domain-specific libraries. Following the trend towards modularization and component-based development of software, the authors claimed that some scientific applications lack on performance because of their using of unspecialised library calls or unoptimized combinations of them [22].

In order to achieve an improvement of the final application, the libraries have to be enhanced with annotations describing details such as:

- variable types;
- dependencies of routines on the input and output parameters;
- per-routine code transformations for some defined guarding conditions.

The Broadway compiler offers as a result an optimized version of the initial source code, while TeleGen (Telescoping Languages Generator) produces a complete DSL to be used from within a scripting language. It also produces an optimized script processor.

An important secondary result can be implied from these two research projects, namely that library-based code optimization cannot be achieved without additional information from within the original application domain.

3.4.4 *Some considerations on DSLs*

There is no doubt that DSLs are a good example to follow when considering Domain Knowledge usage.

First of all, there are the *language components*, i. e. variables, types, operations, which not only succeeded in the user acceptance test, but actually polished continously their usability and improved their semantics. As such, there is a proof that domain information can be expressed using rather few language constructs and, even more, there is a large database of concrete sets of such constructs for each of the common application domains.

Then there are the DSL *frameworks*, which in turn provide valuable information on how particular domain knowledge, like for example the algebraic properties of the tensor operations, can be used in the optimization decisions, like in this case the expression simplification through algebraic transformations.

On the other hand, it is also generally accepted, that DSLs do not always represent a solution for the scientific or industrial projects. A very simple proof is the plethora of scientific and industrial code still being developed using GPLs. Hence one also has to consider the limitations of DSLs with regard to implementation process. Oliveira et al. [51] explain the differences and limitations of DSLs versus GPLs through three pairs of dichotomous characteristics:

1. Abstractness vs. Concreteness;
2. Low-Level vs. High-Level
3. Expressiveness vs. Computational Power

In the end one actually has to decide how much does the domain information have to be separated from the implementation information.

3.5 INPUT AND OUTPUT FORMATS

Input and output files: real "knowledge containers"

Strongly depending on the domain, type of problem and particular implementation, the inputs and outputs of applications do carry along a fair amount of domain specific information. By this we do not refer to the special case of data-intensive applications, like for example genome applications, where the main task is to process huge input data for extracting or searching specific information. We rather look at the information included implicitly in the structure of such files. In this case it does not really matter, for example, how many different genomes or DNA structures are stored in a file, but rather how is one genome described, how are its components defined, how is the relation between these components represented, and so on.

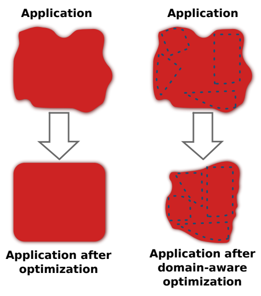
One could consider in this sense that input files are a kind of initial configuration setup, thus many of the structure elements encountered in an input file map directly to data structures and parameters inside the application. The same applies, of course, for output data too.

Important work has been done towards standardization of such input and output formats. On the one hand, common formats support the cooperation and exchange between research groups, and, on the other hand, storing computation results in an appropriate format makes them available for reference and reusage in a long-term basis.

The general widely accepted solution for defining such standard formats is to use Markup Languages (MLs), mostly XML based languages. The general structure of the files are being described using XML Schemas, while the technical terms are usually grouped in Dictionaries. There is quite a large number of existing MLs, which were defined for different application fields, like AML (Astronomical ML), CML (Chemistry ML), MathML (Mathematics ML), NeuroML (Neuronal ML), PhysicsML (Physics ML), SBML (Systems Biology ML), and so on. As for any standard though, its success only depends on whether or not it is accepted by the target users.

This is the case, for example, for the CML (Chemistry Markup Language), which was introduced early in 1999 by Murray-Rust and Rzepa [47] and which is now being accepted as *the* standard for many chemistry applications. Recently, an extension to the CML Schema was proposed, CompChem [58], which targets the quantum chemistry subdomain. CompChem also includes a CML Convention and CML Dictionary to encapsulate the special requirements and methodologies imposed in quantum chemistry. See listing 3 for an example.

Another interesting initiative is the SED-ML (Simulation Experiment Description ML) [85], which aims at providing a standard to encode the description of simulation experiments. It is thus not only the knowledge from a specific domain to be encoded, but also the specific knowledge from the simulation process itself.


Application
Application
Application after optimization
Application after domain-aware optimization

Domain inside app. (see figure 7b)

3.6 DOMAIN-AWARE PERFORMANCE OPTIMIZATION

As indicated at the beginning of this chapter, we switch now to another group of research projects, namely those perceiving the domain *from*, or *within* the application.

The main research field we consider in this context, is that of performance optimization. The goal is to produce a well tuned application, by means of performance measurement, analysis and optimization decisions. Two common and rather simplis-

```
<module dictRef="cc:jobList">
   <module dictRef="cc:job" title="Geometry␣optimization␣with␣Gaussian␣03">
      <molecule id="m-int" formalCharge="0" spinMultiplicity="1"
          convention="convention:molecular">
         <atomArray>
            <atom id="a1" elementType="O" x3="0.0" z3="-0.39016"/>
            <atom id="a2" elementType="H" x3="0.76357" y3="0.0" z3="0.19508"/>
            <atom id="a3" elementType="H" x3="-0.76357" y3="0.0" z3="0.19508"/>
         </atomArray>
      </molecule>
      <parameterList>
         <parameter dictRef="cc:method">
            <scalar>B97-1</scalar>
         </parameter>
         <parameter dictRef="cc:basis">
            <scalar>6-311+G(d,p)</scalar>
         </parameter>
         <parameter dictRef="cc:goal">
            <scalar>Geometry Optimization</scalar>
         </parameter>
         <!-- omitted lines -->
      </parameterList>
   </module>
   <!-- omitted lines -->
</module>
```

Listing 3: A fragment from a CML file storing computational chemistry output [58].

tic understandings of an optimized application, as also depicted at the bottom of figure 7b, are that of a

- *smooth execution*: the square on the left-hand side, as a regular geometric figure, or a
- *perfect fit*: the figure on the right-hand side, with optimal space usage.

These are not the only representations, nor the most accurate ones, and in practice one often encounters a combination of such "pure" understandings. They do serve our goal, though, to explain the domain-aware performance optimization.

Without considering the domain specific constructs or the semantics of the high-level programming languages, the optimization result tends towards the picture on the left: the "corners" and "irregularities" with respect to the performance data are cut or polished to achieve a good performance. This performance though, is mostly based on metrics related to the hardware, the system, or maybe the middle-ware being used. Thus the resulting optimized application is modeled to fit exactly to the machine.

On the other hand, if the semantics and the abstractions at the application level would be accounted for in the performance analysis, the optimization result would tend more towards the picture on the right-hand side.

The very fact that there do exist different optimized versions of the same applica-

Whether, and how, does the domain specific information influence the performance optimization?

```
PROGRAM example
parameter (N=1000)
integer A(N+1,N+1), B(N,N), asum

A = 0
DO k = 1,10
        FORALL (i = 2:N+1, J = 2:N+1) A(j,i) = k * (i+j)
        asum = SUM(A)
        FORALL (i = 1:N/2, J = 1:N/2) B(j,i) = A(j,i) + A(j+1,i+1)
END DO
END
```

Listing 4: Example CM Fortran program [34]

tion raises up a question: *Whether, and how, does the domain specific information influence the performance optimization?*

We already saw in chapter 2 that the performance measurement and analysis workflow implies several challenges, starting from the very definition of the performance itself, to the many theoretical and practical issues associated to each step in the process - instrumentation, measurement, visualisation, and analysis. Each of these issues bring their contribution to the question above, and they, in turn, are also being influenced by the more general approaches regarding the performance optimization process. One example is the concept of *domain-specificity* or *domain-awareness*. We already presented in the first part of this chapter, the implications of the domain related concepts into other branches of computer science. In the remainder of this chapter we focus on the research projects which either contributed to, or actually used the domain specific information in performance optimization.

We start with the research concerning *performance mapping* and *data-centric performance*, which we consider to be the building blocks for introducing domain specific information into the optimization process. We then introduce two projects addressing to some extent conceptual extensions of the performance mapping, namely *custom metrics definition* and *phase-based performance analysis*. In the end we present performance modelling and auto-tuning as complementary techniques, which also employ domain specific information.

3.6.1 *Performance mapping*

Performance mapping was first introduced by Irvin [33] and extended later by Shende [73]. The main idea is that of propagating performance data between the different abstraction layers of complex applications. The goal is to achieve a better profiling for high-level languages.

Express performance in terms of the semantics of the programming language.

Irvin assessed the need of expressing performance in terms of the semantics of the programming language, such that programmers could react and adapt their applications, while remaining at the abstraction level of the initial source code. He proposed as a solution the *Noun-Verb (NV) model*, a framework which provides the support for several performance mapping techniques.

The code snippet in listing 4 is an example of a fragment of source code written in CM Fortran [78] with common variable declarations, subroutine calls and loops. Based on this example, the NV model consists of [34]:

1. *nouns (N)*: any program elements for which performance measurements can be made; such are the `FORALL` loops, the arrays (`A` and `B`), the `A = 0` statement, but also the `program` (first line of the code);

2. *verbs (V)*: any potential action that might be taken by a noun or performed on a noun, e. g. statements *execution*, array *assignments*, or *reduction* `ASUM = SUM(A)`;

3. *mappings*: relations between nouns and verbs from one level of abstraction, to the nouns and verbs of another level of abstraction. The CM Fortran code is compiled into a sequential program and a set of node routines. The node level is the lowest level of abstraction in this case. Some verbs at this level include *Compute*, *Wait*, or *Broadcast Communication* and the nouns are all the compute nodes. One mapping could in this case relate a particular array (a noun in the higher level) to a set of particular compute nodes (nouns in the lower level).

4. *sentences (S)*: particular execution instances of the program constructs described by verbs. A sentence consists of a verb, a set of participating nouns, and a cost, where the cost could be a metric like time or channel bandwidth.

This model was integrated within the ParaDyn tool [56] and several experiments were ran on CM Fortran applications. Figure 10 shows a histogram produced with Paradyn using the NV model, where the cost function was chosen to be the computation time. The upper trace shows the overall computing time, while the lower two lines represent the computation time used for a particular data structure and one of its subcomponents. This illustrates very well one of the main outcomes of Irvin's work, namely that of having shown that "data views of performance can lead to more focused explanations of performance", and, in particular, that "performance problems are localized among a few parallel arrays, while being diffused among many code statements" [34].

Performance problems are localized among a few parallel arrays, while being diffused among many code statements.

While being probably the first research towards a semantics-aware performance and, particularly, a data-centric approach of performance, the NV model exposes though several limitations at both conceptual and implementation levels:

A. the nouns are bound to the programming level structures, whereas higher levels of abstractions would also be useful.

B. in some cases the cost function of a sentence at higher abstraction level cannot be derived exactly from the cost functions of the sentences at the lower abstraction level. For example, given a mapping of type *many-to-one* which is associating several sentences on the higher abstraction level to one sentence of the lower abstraction level, then the cost at the higher level will be either expressed as an aggregated value (only one cost for all sentences together), or each sentence will be assigned the same cost (with the cost on the lower level being thus evenly split to the sentences at the higher level). None of the two strategies is an optimal one.

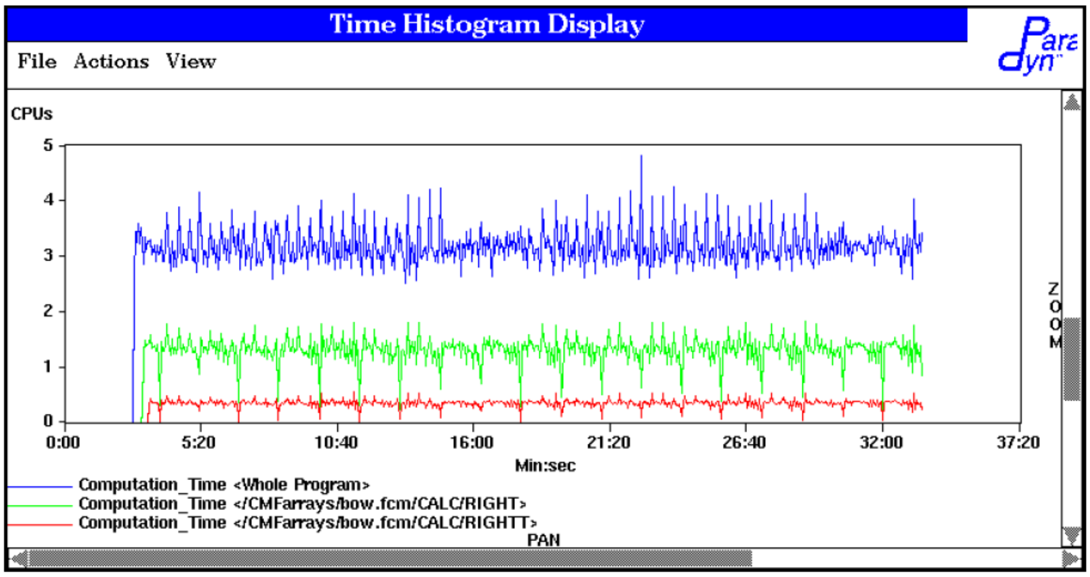

Time Histogram Display
File Actions View
Paradyn
CPUs
5
4
3
2
1
0
0:00
5:20
10:40
16:00
21:20
26:40
32:00
37:20
Min:sec
ZOOM
PAN
Computation_Time <Whole Program>
Computation_Time </CMFarrays/bow.fcm/CALC/RIGHT>
Computation_Time </CMFarrays/bow.fcm/CALC/RIGHTT>

Figure 10: Data view of time performance measurements, generated with Paradyn using the NV model. The green line shows the computation time for the array `RIGHT` declared in function `CALC` in the file `bow.fcm`. Source: [34].

c. the model lacks a technique for handling asynchronous executions of verbs at different abstraction levels. More precisely, it can only map the costs between sentences which are being active at the same time. If, for example, a data transfer is implied by the sentence at a higher-level, but the actual communication at the lower-level is being postponed, then the cost for the latter one is not going to be accounted for, when determining the cost of the sentence at the higher level.

d. the cost function cannot refer to hardware counter values, being limited to metrics measurable based on instrumentation.

The first three issues were also observed by Shende [73], who addresses them in the extension of the NV model that he proposes. The *Semantic Entities, Attributes, and Associations (SEAA)* dynamic mapping technique is focusing on enhancing even more the role of semantics within performance measurements:

- *Semantic Entities*: the nouns in the NV model are replaced by semantic entities, which are not necessarily bound to program constructs. This way a higher abstraction level is allowed, actually the domain specific information level we encountered before.
- *Attributes*: any entity can receive semantic attributes and thus the issues regarding asynchronous activities are solved.
- *Associations*: entities or application objects can be associated directly with performance metric entities, overcoming this way the cost accounting problems.

Shende explains the importance of semantics using the example in listing 5. The code fragment mimics a simulation where particles belonging to the six faces of a cube have to be processed, given that the characteristics of the particles depend on

```
Particle* P[MAX]; /* Array of particles */

int GenerateParticles() {
   /* distribute particles over all surfaces of the cube */
   for (int face=0, last=0; face < 6; face++){
      int particles_on_this_face = ... face ; /* particles on this face */
      for (int i=last; i < particles_on_this_face; i++) {
         P[i] = ... f(face); /* properties of each particle are some function f
             of face */
      }
      last+= particles_on_this_face; /* increment the position of the last
          particle */
   }
}

int ProcessParticle(Particle *p){
   /* perform some computation on p */
}

int main() {
   GenerateParticles(); /* create a list of particles */
   for (int i = 0; i < N; i++)
      ProcessParticle(P[i]); /* iterates over the list */
}
```

Listing 5: A hypothetical particles simulation [73].

the face they are belonging to. Using the NV model described above, one could only generate performance data at the *particles* level, since these are represented through program constructs and can be defined as nouns. What the application developer would probably like to analyse though, is the performance data measured per *face*, since this is the actual approach followed by the simulation.

Listing 6 shows how to integrate the SEAA concept within the same code fragment, using a set of library calls. The *faces* entities are implicitly declared through the time counter created for each of them within the iterations of the outer loop in `GenerateParticles`. The particles are then being assigned to faces by adding a mapping to the appropriate timer, which is then going to be started throughout the time of computing that each specific particle.

The custom timers support was integrated within the TAU [80] performance toolkit. Shende presents the results for running performance analysis on two C++ based applications and shows that performance for semantic entities like *tasks* and general *array operations* could be measured and analysed.

Although the SEAA model managed to bring another important step towards semantics-aware performance analysis, it also inherited some important limitations:

A. lack of formalization: no means are provided to actually define the semantic entities and the semantic attributes. The use of custom timers keeps the application semantics still "implicit", or at the "logical level", which is exactly what one would like to avoid.

```
Particle* P[MAX]; /* Array of particles */

int GeneratePartic1es() {
   /* distribute particles over all surfaces of the cube */
   for (int face=0, last=0; face < 6; face++){
      int particles_on_this_face = ... face ; /* particles on this face */
      t = CreateTimer(face); /* information relevant to domain */
      for (int i=last; i < particles_on_this_face; i++) {
         P[i] = ... f(face); /* properties of each particle are some function f
             of face */
         CreateAssociation(P[i], t); /* Mapping */
      }
      last+= particles_on_this_face; /* increment the position of the last
          particle */
   }
}

int ProcessParticle(Particle *p){
   t = AccessAssociation(p); /* Get timer associated with particle p */
   Start(t); /* start timer */
   /* perform some computation on p */
   Stop(t); /* stop timer */
}

int main() {
   GenerateParticles(); /* create a list of particles */
   for (int i = 0; i < N; i++)
      ProcessParticle(P[i]); /* iterates over the list */
}
```

Listing 6: Integrating the SEAA concept using timers [73].

B. restrictive metrics set: there is only a limited choice of performance data which could be gathered and analysed for the entities, namely time and channel bandwidth.

C. implementation restrictive to language: due to it being based on the capability of accessing the C++ object addresses (which cannot be done for a Fortran implementation), the implementation solution provided for the timers association is rather restrictive.

We leave the discussion on how these shortcomings could be overcome for the next chapter, where we give a presentation of our *Domain Knowledge* approach.

Further research using the concepts of performance mapping followed mainly two directions:

- *data-centric* profiling, based on the observation of Irvin [33] regarding the importance of relating performance to data: "Performance problems are localized among a few parallel arrays, while being diffused among many code statements.".

- *dynamic performance* analysis, based on features like the dynamic timers and associations integrated within TAU.

While *data-centric profiling* is a valuable approach in performance measurement and analysis, the works dealing with this subject focused rather on the technical challenges of the exact mapping of the collected performance data onto the application data structures. The *semantics* level highlighted within SEAA and which is also of interest for *Domain Knowledge*, did not receive further attention within this context.

As of the second research path, that of performance dynamics, we detail on the particular concept of *phase-based performance analysis* in the following section.

3.6.2 *Phase-based performance analysis*

The concept of *phases* in scientific applications has long been introduced and used in the context of performance visualisation and analysis [24], as well as in performance modelling [87]. It is common practice for application developers to think of their code structure and execution in terms of phases or stages of the natural phenomena modelled/simulated therein. Hence, the performance analysis tools should give the user the opportunity to associate performance data with the corresponding phases of the application.

Phases = segments of time.

A straightforward approach is the one considering the *time* characteristic of phases. During its execution, an application passes through different consecutive computation phases. This time development could be tracked by means of phase start and end timestamps. Entering and leaving a phase can be signalled based on events, which in turn have to be defined by the application developers. Current tools like Vampir [83] and TAU [80] support this through their manual instrumentation features.

An interesting on going work is that of automatically detecting performance patterns and application phases out of the performance traces [10].

Obviously, this time-related approach can be applied only with techniques which keep track of the timestamps of the performance data, i.e. *performance tracing*. The current issue with performance traces for HPC applications though, is that they imply considerable overhead due to the large amount of data which has to be gathered and stored, thus often producing costly writings/flushes to the secondary memory. Moreover, large trace files also imply increased post-mortem processing times.

Phases = execution context.

The common alternative in this case is to apply *profiling* instead of tracing. Profilers use different techniques to process at runtime the retrieved performance data, thus keeping in the memory only a summary or an aggregated value of the retrieved performance data. This implies, of course, that no information with respect to the timestamps can be stored or used, and thus *phases* in profiling techniques cannot be characterized by the time dimension in this case.

Malony, Shende, and Morris [42] offered a new view on phases, highlighting their *context holding* property and showing how they can be used in performance profiling. Continuing the direction of SEAA, they stress the fact that a phase "generally represents logical and runtime aspects of the computation, which are different from code points or code event relationships", thus repositioning phases at the higher abstraction layer of *semantics*, as opposed to the layers of the source code and application events.

Phase-based profiling was integrated within TAU, using the concepts presented above for SEAA. The technique resembles the *callpath profiling* technique, where at the occurrence of an application event, the tool also checks the current phases stack in addition to the common callstack query. An important role play the *dynamic phases* which are nothing else but additional events associated with dynamic timers as already presented with SEAA. A detailed example of applying phase-based profiling can be found in [74].

Phases = recurring operations.

Another view on phases was explored by Szebenyi, Wolf, and Wylie [79] in the context of *performance dynamics*. Considering that in HPC one often encounters long-running applications, or also applications using adaptive algorithms, it is expected that the performance of these codes will vary during one execution. In order to track these changes, while still using profiling instead of tracing technique, a particular type of phases are used. Based on the observation that most scientific codes iterate over a main loop to compute the final solution, one chose this recurring unit of execution to be defined as a phase. More precisely, each iteration of the main loop is recorded as a separate phase in the execution. Thus, changes on the performance behaviour can be observed as changes between different recurring phases or iterations. In their work, Szebenyi, Wolf, and Wylie integrated this feature in the Scalasca [69] tool, using a manual instrumentation technique similar to the dynamic phases presented above for TAU.

All three cases presented above actually share the same idea: the user can contribute to the performance measurement and analysis by explicitly specifying through phase definitions the logic and semantic structure of their application.

3.6.3 *Custom metrics*

Another means for users to interact/intervene in the performance analysis and measurement process is to specify the performance metrics which they consider relevant for their application. As described by [17], performance optimization is driven by "programmers' hypothesis on potential performance problems". It would thus be useful, if they were given the opportunity to model and express by themselves these possible bottlenecks and subsequently check out the corresponding measurements.

User custom metrics offer to users exactly this possibility: to define the metrics which the performance tools should measure and display for them.

In essence, this is also a great opportunity to link performance with application semantics, as users define performance metrics or properties and make assumptions on performance problems based on their mental model of the application execution. Current implementations though, limit themselves to at most application level metrics. It is rather like applying the "printf debugging"[4] technique, just that in this case one tries to eliminate performance bottlenecks instead of application bugs.

Custom metrics were introduced early in performance analysis, mostly as the mere combination of the predefined low-level metrics. An important step towards including semantics within metrics was done in EARL and G-PM projects. The latter one added to metrics also the application events dimension. Thus users could obtain cus-

4 Commonly known as "printf debugging", this technique is aiming to find software bugs by watching output traces. The name comes from the `printf` statement used in C for printing out the needed values and information.

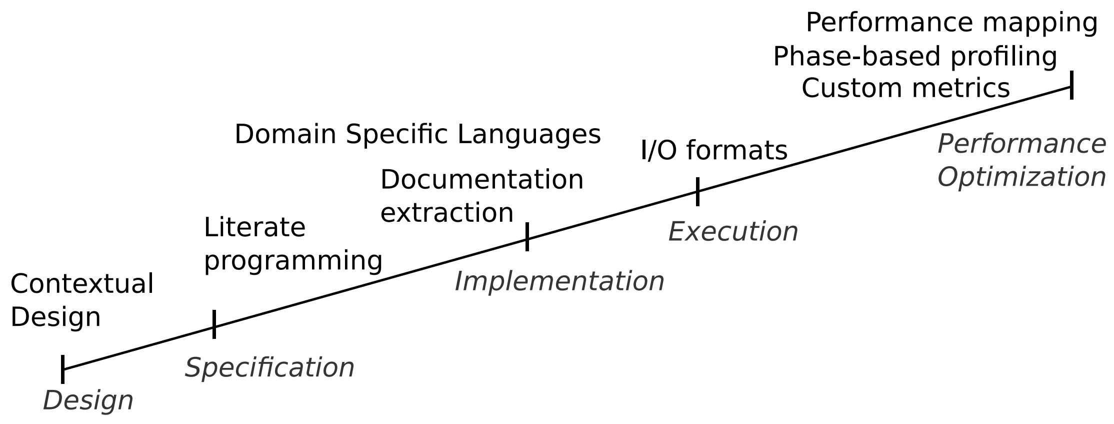


Figure 11: Schematic development timeline of an application and corresponding research and application fields which tackle DK.

tom metrics by combining existing predefined low-level metrics, but also specifying particular application events for which these metrics should be computed. G-PM uses the PMSL language to specify the custom performance metrics, while the application events are declared by instrumenting the source code with function dummies.

A similar approach, but with an easier to use instrumentation API, was to be introduced through the already mentioned dynamic counters technique of the SEAA and TAU. Reimplemented later in the Score-P [71] project, dynamic counters offer a good support for custom metrics definition. Using a simple code instrumentation, e. g. `DEFINE_CUSTOM_METRIC(var)`, users are being given the possibility to pass to the performance measurement framework any application information, like variable values or strings.

One could argue that custom metrics are the right technology to support semantics and with this, higher abstraction levels of information. Nevertheless, it is still lacking a better formalization to support, for example, the connection between metrics and other semantics-based concepts, e. g. the phases discussed in the previous section.

3.7 related work summary and conclusions

We have seen throughout this chapter that there are many computer science areas in which different aspects related to *Domain Knowledge* have been explicitly or implicitly employed throughout the years. One could follow these areas on a schematic application development timeline, like that in figure 11. In some cases, there is no clear delimitation with respect to the development phases influenced by one particular field. For example, the Domain Specific Languages could cover all phases from design to execution, and sometimes even performance optimization. Nevertheless, we use this schematic timeline to point out the segments where there are special contributions of these fields with respect to Domain Knowledge.

To begin with, we consider the design phase of applications. We saw that *Contextual Design* sets the focus on observing the user in their environment and also reveals the importance of the terminology used by users in their domain of activity. The designers have to identify and integrate domain specific words in the application development. This is similar to the case where a performance engineer from a computing center receives the task to help in optimizing the performance of a particular scientific application. He or she has to identify in that particular scientific domain, the concepts and the definitions that scientific applications are working with.

With respect to the next phase, the application specification, we have presented the early idea of *Literate Programming (LP)*. LP is actually spanning over both phases of specification and implementation, conforming to the idea that the two phases should actually merge into a single one. The specification of the yet to be implemented application should be written as explanations within the source code. This is to say that the actual implementation is *driven by the semantics of the application*, as the programmer is mainly describing the solution and the implementation is "naturally" following it.

As the LP approach did not enjoy a good acceptance among programmers, we also had a look at other possibilities of embodying and extracting documentation to and out of the source code, such as the Javadoc and Doxygen documentation frameworks. Using corresponding annotation rules, valuable information can be carried over with the source code and, when needed, properly extracted and visualized. The information is mostly related to details of the code structure or functions description. These approaches are not targeting higher abstraction levels, like domain information, but mostly stay within the code level information. Nevertheless, they are of importance in our study, as they do enjoy a great acceptance in software engineering, and thus provide a valuable hint on preferred documentation method.

Further on in our schematic timeline, we considered the *Domain Specific Languages*. This is again a field which spans over several application development phases. As a matter of fact, there are complex frameworks like the ACES III which support the complete development process. DSLs are high-level languages which encompass domain specific information directly within their syntax and semantics. From the users' point of view, this is the ideal case, as they can express themselves "naturally" in such a programming language. The pitfalls of DSLs are their limitation to mostly very particular application fields and also the mostly untouched parallel performance aspects. To our knowledge, there are no performance measurement tools which support applications written in a DSL language.

Searching for other Domain Knowledge resources in current application development, we highlighted the potential which different *Input and Output formats* expose in this context. In particular, the different Markup Languages (MLs) shaped for specific scientific domains offer a great means to store and handle domain specific information. CML, for example, shaped its terms and identifiers over several years in the community and could now serve as a template for pasting domain information onto source code.

In the last part of this chapter we had a more detailed look inside research fields related to the last phase in our timeline, namely the performance optimization. A very important work in the strive towards Domain Knowledge is constituted by the *Performance Mapping* methodology. The goal is to involve semantics in the performance analysis process, by mapping the low-level performance data to the high-level code structures. This is also an important backbone on which to build the support for DK in performance analysis. Other results came from *phase-based profiling* and *custom metrics*, which also tackle the importance and usage of semantics and thus narrowing down towards the DK philosophy.

	Identification	Definition	Application
Contextual Design	domain terminology		
LP		solution description	
Documentation extraction		Annotation Language	
DSL	language design	special syntax	data structures, operations
I/O formats	structures, relations	Markup Language	
Performance mapping			code semantics
Phase-based profiling	execution phases		
Custom metrics		user-based	flexibility

Table 2: Classification of the related works, based on their contributions to DK.

To conclude on the various related works presented above, we structure them in table 2, based on their contributions to DK. We consider three guidelines in developing a proper support for DK: first, the users have to become aware of their knowledge - *identification*; then they need a means to express and pinpoint the knowledge - *definition*; and finally, there have to be the means to work with and use the knowledge - *application*. We detail on each type of contribution in the next chapter, when we present our approach, the Domain Knowledge paradigm.

4 "HARVESTING" THE DOMAIN KNOWLEDGE

In this chapter we introduce the *Domain Knowledge (DK)* approach. The focus here is on improving the performance optimization process, but we show that DK can and should go along with and enhance the other application development phases as well.

We start with a short discussion on the differences between *domain specific information* and *domain knowledge*. We then continue with the description of the DK philosophy and the design considerations, based on the related work previously discussed. Then, in the last part, we present the three main constructs on which is the DK built upon: *entities*, *operations* and *phases*, and we introduce the new *DK Metrics* for performance analysis.

4.1 ON *information* AND *knowledge*

In the previous chapters we used the terms *information* and *knowledge* almost interchangeably. We would like to point out here though, the light but important difference between the two of them.

The DIKW (Data Information Knowledge Wisdom) pyramid [65] is a well known representation in Information Management and in Knowledge Management. The four concepts, data, information, knowledge and wisdom, are presented as the layers of a pyramid, with the data being on the bottom and wisdom on the top. Each layer builds on the next layer below itself. Thus *information* is the layer which gives semantics to raw data and *knowledge* is the layer which gives the active reasoning, based on the information below. Without entering the recent polemics as whether it should really be just a pyramid, or the iterative aspects of the process, we extract those characteristics of knowledge and information which are most relevant to us: *knowledge* is *subjective*, *belonging or depending on a person*, while *information* is *objective* and can be *formalized* and *transmitted* as such.

Extending these remarks to our research context, one could observe, that previous related work always refer to *domain specific information* or *application specific information*. Even *semantics* and *higher abstraction levels* eventually do map to the concept of information, as they receive a formalized, objective expression.

In contrast, *domain knowledge* is not the mere information on a specific domain, but actually all the information, reasoning and maybe even particular approach style, one person could have for that specific domain.

One could even say that applications could, and some actually do carry along domain specific information, but the knowledge related to the domain will still be held by the developers or the users.

A good example is given by the different domain specific libraries. If well designed and well documented, they can offer rather complete domain specific information, as for example, the parameters required for a particular solver, or the mathematical model being used by a method. But the knowledge about which solver to choose for a given particular problem, how to combine them, or how to go about different limitations of the library, this has to be provided by the person using that library. It thus depends on how skilled the user is in that specific domain.

A very simplified definition would be that domain information is eventually the *object* of the formulations and expression process of domain knowledge. Furthermore, we could also say that previous related work mainly focused on finding out most appropriate format and properties of this *objects*, such that it could be used within application development. In our work though, we do not target the *form*, but rather the means of generating and using such *objects*.

Also worth mentioning is the fact that data and information are "raw matter" and thus locally understandable, while knowledge and wisdom represent holistic, global aspects.

4.2 our approach

It is widely accepted in the HPC community that developing applications for supercomputing still poses a great challenge for the HPC users. One of the main challenges in this sense is to achieve a good performance of the applications on those very complex systems, that supercomputers are nowadays.

We address this particular challenge and propose as a solution the integration of the *Domain Knowledge (DK)* paradigm within the application development and optimization process.

DK implies a twofold *general view* on the application. Firstly, a general view on the semantics of the application. It is like having in mind the final picture of a puzzle, when searching and fitting the single pieces. Secondly, a general view of the application development process. Almost every phase in the development encapsulates pieces of knowledge.

At the conceptual level, we refer by *Domain Knowledge* to the knowledge which developers and users have about the particular scientific domain their application resides within. These could be, for example, acquired skills regarding different solving methods or limitations of the mathematical models. These skills usually come from own experiences with applications/packages or can be adapted from others sharing their knowledge.

At the implementation level, the *Domain Knowledge* paradigm is supported through the *Language for Domain Knowledge (LaDoK)* and the particular performance metrics, as well as their corresponding extensions of the performance tools (figure 12).

We define DK by means of the following three elements: *identification*, *expression/definition* and *application/usage*.

4.2.1 *Identification*

It is common practice to depict the task of having to put in relation two or more groups (of people or elements) which expose evident differences and challenges, as having to "build a bridge", to "bridge a gap" or to "close a gap". Although this could apply also in the case of scientific code and HPC, or natural scientists and computer scientists, we instead find it more suitable to present this task more like having to "cut a path through a forest", in order to use the various wood essence present in there.

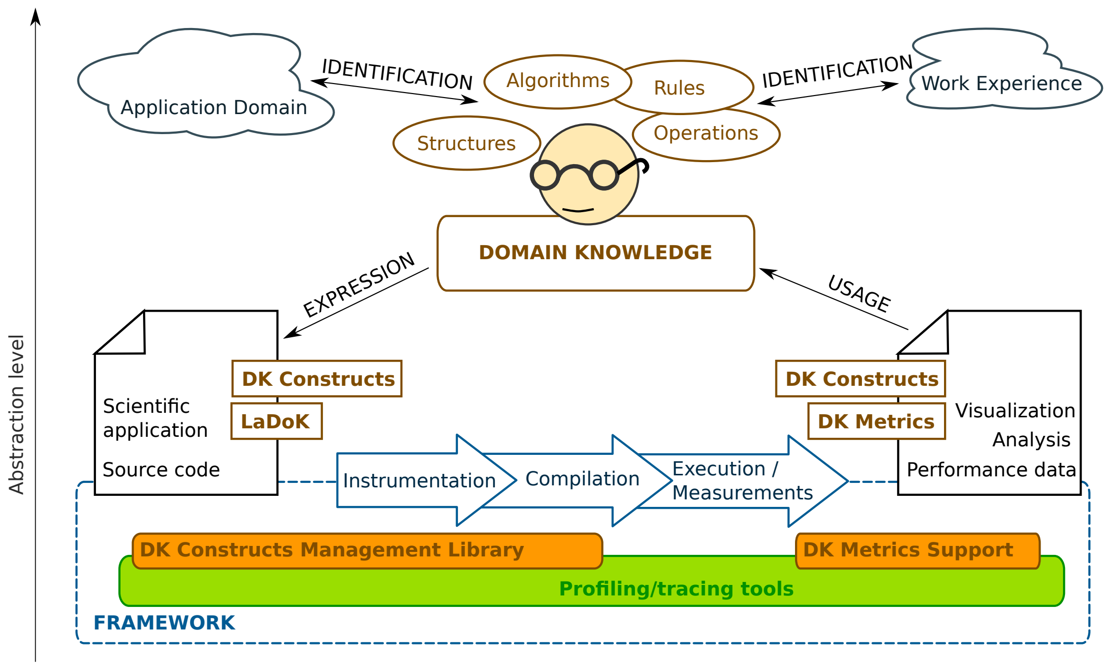


Figure 12: DK approach overview.

The main idea is that DK relies on the elements which are already present in the application development process, offering tools to extend the usage of those elements. One does not have to build an additional "system", a bridge, but rather structure and exploit what is already provided. In this sense, the knowledge from different scientific domains is already there, otherwise the developers could not actually generate their applications either. But they need to get aware of this knowledge, such that one can take advantage of it throughout the entire application development.

In Contextual Design (see section 3.1), the application developers are getting in the working context of the future users to observe them on-site. This gives them the opportunity to identify the specific terminology their users employ. Even if they do not completely understand all terminology, they are able to identify it. Actually, it is because they are from outside the application domain and the terminology is not common for them, that they can easily identify these special terms.

In common application development for HPC, the prerequisites are somehow different. Here, the application developers can also be considered as application users. They do not have the advantage of an "outside observer" to easily identify all special terms, but they do have the great advantage of being able to master all the terminology for that specific domain.

Considering the case of applications developed using Domain Specific Languages (see section 3.4), one could compare the domain knowledge identification to the learning phase of a new DSL. While learning a DSL, the future developers eventually map the terms they have been using so far for reasoning inside their scientific domain, to the data structures and operations provided by the new programming language. It is for sure easier to identify this way the terminology and knowledge one already holds, but it is also to some extent limiting: DSLs might be very domain specific, and thus leave no room for adaptation, when the "hardcoded" knowledge does not entirely overlap with the knowledge the developers already hold.

As pointed out in the previous section, knowledge has also a subjective component. It is thus rather difficult to formalize and elaborate a general method for identifying DK. For the purpose of this work, one could follow the example of DSLs and start with reading *Language for Domain Knowledge (LaDoK)* samples. We present LaDoK later in chapter 5.

4.2.2 *Expression / definition*

Having identified knowledge, one needs the proper framework to also make use of it. Knowledge processes, analyses, uses information. It is based on information. Thus, above all, the framework has to provide a means to express/define domain specific information.

The key point in designing such a framework is to keep in mind that the end purpose is not just carrying or pinpointing the domain specific information, but rather that of providing the necessary support to apply domain knowledge for improved application development.

Good examples of such frameworks are given by the more advanced DSLs presented in the related work chapter, like ACES III (section 3.4.1) or TCE (section 3.4.2). In these cases, the developer *implicitly* provides the necessary domain specific information, through the very fact of writing the program using the specialized language. The language encapsulates by its nature domain information.

There is no doubt, specialized languages offer a suitable solution for expressing domain information. But while high-level languages like the DSLs above are very promising, one should also consider the limitations which come therewith due to their high specialization.

A good alternative is given by *annotation languages*, as those used by the Doxygen and Javadoc *documentation frameworks* (section 3.3). One preserves thereby a fair level of abstraction, and in the same time separate the actual implementation of the application from its description. This provides the flexibility required when it comes to adopting slightly different semantics.

In order to answer the requirement that information has to be expressed in such a way that it offers support for the application of knowledge, one should consider the important observations raised up by *Literate Programming (LP)*. In LP, the programs had to be written in such a way that other developers, e. g. students, could easily read them and understand the solution. The documentation in this case *explains* the implemented solution. Likewise, in our context, the developers make use of their domain knowledge to process the information delivered along with the application.

The suggested solution is to use a *markup language* to highlight information directly in the source code. If we were to think of scientific applications as scientific papers rewritten in a programming language, then the LaTeX instructions used to mark the raw text in order to properly format and highlight the actual information in the document, could be "rewritten" in LaDoK instructions which mark the source code in order to properly highlight the actual domain specific information in the application.

4.2.3 *Application / usage*

As specified earlier in this chapter, we will limit the presentation of our approach to the context of performance measurement and analysis. Nevertheless, DK could be applied in a broader spectrum of software development fields, as explained in chapter 11.

The actual utilization of DK is conditioned by the following three aspects:

1. *DK management:* the domain specific information has to be registered, stored, and retrieved when needed. The object of this management process are a set of DK constructs, which we present in the next section. Chapter 7 then gives more detail on our implementation of such a DK management library.

2. *DK integration in profiling and tracing:* the actual performance measurements have to be enhanced with domain knowledge elements, in close connection with the DK management system. A good example is given by the dynamic counters introduced already in TAU and Scalasca (see section 3.6.1). We detail more also on this subject in chapter 7.

3. *DK support in performance analysis:* this is the "reward" for all the effort made so far. The developers can use their domain knowledge to reason on the performance results, based on the domain information which penetrated through the entire process. Specialized analysis instruments are also available, like, for example, the DK Metrics we describe in section 4.4.

One of the important contributions of DK is that of easing and improving the developers' experience in performance analysis. They can analyse performance using constructs which are familiar to them and suited for the specific scientific domain, but without being restricted to predefined structures. In the same time, through the underlying framework they are also given guidance on how to wield their applications to obtain suitable performance information.

4.3 DK CONCEPTS

In this section we elaborate on the particular information structures and concepts, which constitute the building blocks for the *Domain Knowledge* paradigm.

Shortly re-rendering the observations in chapter 1, we saw that the development of new scientific applications includes somewhere in the process, the transition from phenomenon and observations to formulae and methods/algorithms and finally to the variables and functions/routines of the written source code.

The only research we could find related to keeping the connection between the initial scientific formulations and the final executable code is still rather far from our current context. These are the specialized Markup Languages used to describe *input and output scientific data formats* (section 3.5). Their purpose is to offer a means to describe the results and input of scientific computations. The main concepts the languages are based on are those of *containers* and *controlled vocabularies*. The first ones are self-describing, the data being packed together based on different rules. The latter ones, offer a standard list of terms to be used as tags or attributes. See section 3.5 for a code snippet.

The very fact that there are so many MLs, one or more for each of the computational sciences, shows clearly that it is a hard task to find only one set of constructs to use in all scientific applications. On the one hand, there is the need to provide general semantics, but on the other hand, there are many particularities in each of the sciences which have to be preserved.

Mathematics are the common language for all computational sciences.

We thus have to turn back to the higher abstraction level of mathematical formulae, as it seems that *mathematics are the common language* for all computational sciences. Therefore, we reconsider the transition from formulae and methods to written source code and search at this level for the basic constructs.

4.3.1 *DK Entities*

First of all, each variable and parameter of a formula is translated to a variable or constant of the programming language. But variables in formulae have their well established semantics, whilst variables of programming languages are just data structures. We thus have a first type of information which has to be preserved: semantics of the variables in formulae. We call these constructs *DK Entities*.

An *entity* is to some extent similar to the notion of *container* mentioned above. It groups together programming variables which describe one semantic construct and it can also hold different attributes to better characterize them.

4.3.2 *DK Operations*

The main part of a scientific code is built on algorithms which either emulate some complex mathematical operations or implement methods solving equations and alike. These are the next building blocks for *domain knowledge*, we call them *DK Operations*.

Again, *operations* could be thought of as containers which group together a set of functions in the source code. The important feature, though, is that DK Operations not just *group functions*, but also *enhance them* with domain semantics, which can be used later on. For example, a function which simply processes the grid in a multigrid solver, does not keep the information regarding the current coarsening level. But this would be a valuable information in the performance analysis process and could be delivered as an attribute of a corresponding DK Operation.

Also, DK Operations can be linked to DK Entities. It is natural and mostly only implicit information, that specific actions are performed using or generating specific data. One is now provided the possibility to explicitly link instantiations of these two concepts.

4.3.3 *DK Phases*

There are cases in mathematics too, where one has to consider exceptions from the general rule or theorem being applied. This usually depends on some particular values or combination of conditions which fulfil simultaneously. When implemented in an application, running such a code would lead to the situation that sometimes, special tracks in the code will be executed, given that special results occur in computations.

We relax a bit the "exceptional" aspect of such events, as not to conform just to the exception handling common in programming, but rather to denote all particular cases which might appear during the execution of a scientific code. We call these cases *DK Phases*.

From a pure developer's point of view, the main purpose of DK Phases is to mark down runtime events - execution branches which depend on the computed data. For a computational scientist however, DK Phases empower the expression of the logical aspects or stages of the modelled phenomenon. In any case, the result is a better experience in performance analysis.

We already saw in section 3.6.2 different approaches towards application phases definition. DK Phases are introduced at a rather conceptual level, and hence they could function as any of the mentioned phase types - *time segments*, *execution context*, or *recurring operations*.

One specific property is that, unlike DK Entities and DK Operations which are a kind of *containers*, DK Phases hold a global status of the execution. To relate to the previous parallels between the DK constructs and the ML constructs: there is no direct correspondence between the DK Phases and the ML constructs, as the first ones refer to *runtime* events, and the latter ones to input and output *static* properties.

4.4 DK METRICS

As pointed out earlier in this chapter, the main purpose of the DK paradigm is to improve the performance analysis experience. Following the idea, that HPC users should be able to interact with the performance tools in a "natural" manner, i.e. using concepts and domain specific terms which are common for them, we introduce the *DK Metrics*. These new metrics combine and integrate the DK Constructs explained above - entities, operations and phases - with performance specific information like memory usage, computation time or communication patterns. They are the application instrument of DK in performance analysis.

We distinguish three types of DK Metrics and counters:

- *extension metrics:* DK is being integrated within existing metrics, extending their semantics over DK Constructs or other components to support DK. Such are, for example, the *Iterometers* presented below.

- *pure metrics:* these are "stand-alone" metrics, for which special targeted measurements are being conducted, rather than re-using performance data from other metrics. An example is given below by the MPE metric.

- *support parameters:* application specific counters, variables and parameter values which provide dynamic runtime information.

We would like to stress the fact that the DK Metrics are thought to be *tools* for performance analysis. This means that the application developers are expected to utilize them as they need and based on the particular domain knowledge they have. These tools are designed to work with the previously custom defined constructs. One could "pick" a *DK Entity* and "measure" it as needed.

The difference to the *Custom metrics* we presented in section 3.6.3, is first of all the integration of the DK elements. The new metrics we propose are not just the combination of the existing metrics, but they add a reference to the main constructs and concepts of DK. Secondly, there is a difference with respect to the flexibility of these metrics. With DK Metrics, users are more restricted with respect to defining own metrics, but they have more flexibility regarding the measured object, in the sense that they already customly defined the DK Constructs.

We give in the following a brief description of three DK Metrics. For examples and results of applying the metrics on real applications, we refer the reader to chapters 8 to 10.

4.4.1 *Memory per Entity (MPE)*

As the name implies, *Memory per Entity (MPE)* provides information with respect to the memory used by an entity throughout the execution of an application. It is more useful for a developer in reasoning about performance and possible optimization of their application, if it has the opportunity to inspect, for example, not just the application overall memory usage or the amount of allocated memory per each allocate function call, but also the memory used by one particular entity throughout the execution. One use case is provided by the applications in Quantum Chemistry, which are known to produce large amounts of intermediate data. It is of great help in this case to have a better insight of when and what was the memory used for. Chapter 8 gives a complete use case evaluation.

4.4.2 *Computed Entities Count (CEC)*

This is one example of a common metric extension. *Execution time* and its specialized variant, *Computing time*, are by far the most used metrics in performance analysis. Especially in *load imbalance* problems, it is common to compare these times considering, for example, the duration of the same function call on different processes. CEC builds on top of these metrics. It integrates the previously defined *entities*, in the time performance analysis, giving better insights. Expressing the computation load in terms of *how many entities were computed*, assists the developer in decisions regarding, for example, the entities distribution method, or the customization of domain specific operations, based on entity types.

4.4.3 *Iterometers*

Iterometers are a class of extension metrics which target the analysis of issues related to the iterative processes within applications. Most scientific applications are composed of one or a few main loops, which iterate over application-wide dimensions like the timestep or the number of particles in a simulation. Given the common timeline visualization, an *iterometer* adds domain specific runtime events, like, for example, the multigrid level indicator, or the residual value for some current timestep.

An iteration implies the existence of a loop and one or more iteration indexes which run between given start and end boundaries. Sometimes the indexes or the boundaries might not be given explicitly. For example, a `while`-loop executing some convergence algorithm until the residual is reaching an acceptable value. The semantic is still that of an iteration, but the code structure might lack of corresponding data structures. DK annotations would provide here the necessary semantic information which could then be used for targeted analysis with Iterometers.

4.5 comparison to seaa and nv

In the chapter 3 we presented two important related works: the *Noun-Verb (NV)* and the *Semantic Entities, Attributes and Associations (SEAA)* paradigms (see section 3.6.1). We showed that NV was the first to use a mapping technique to explain performance data by mapping it to the structures of the high-level language, and that SEAA extended the concept introducing for the first time a reference to the higher abstraction level represented by the semantics of an application. We presented in section 3.6.1 both the features and the shortcomings of these two paradigms. In the following, we would like to highlight the differences between our approach and these two approaches.

There is first of all a difference at the very general point of view, regarding the perspective offered by each of the paradigms. NV and SEAA approach the problem from the source code towards application semantics, while DK starts from the scientific context of the application towards its implementation. The first ones aim at adding semantics to the *given code structures*, while DK aims at marking code structures with the *given domain semantics*.

Semantic Entities extended the *Nouns* concept by detaching them from the strict source code data structures. *DK Entities* are closer to the *Nouns* concept, due to their *container* characteristic, as explained above. Nevertheless for the particles simulator described in section 3.6.1, *DK Operations* linked to DK Entities are more suitable to handle the computation of the particles on different faces. We provide the entire example in section 10.3.

Also, SEAA practically eliminated the *Verbs* and *Sentences* concepts original in NV, introducing instead the dynamic *Associations* and *Attributes* to handle the implementation difficulties of the initial *Active Sentences*. As a consequence, from the user point of view, all profiling work/instrumentation has to be done now in a "printf"-manner, by switching on and off the defined dynamic counters. The combination of *DK Operations* and *DK Phases* offers instead a different approach, in the sense that it provides both a *natural* and a well *formalized* means of expressing semantics and domain information.

The main idea of *performance mapping* is to relate performance issues to the code structures responsible for them. NV is rather focused on identifying the data structures and data structures operations, while SEAA tends more towards exact time costs accounting for tasks-based like executions. While these are similar to what our *extension metrics* also offer, both NV and SEAA lack a clear integration of the application semantics within the metrics. Also, the other two types of *DK Metrics*, the *pure metrics* and the *support parameters* do not have a counterpart in either NV or SEAA.

4.6 conclusions

The Berkley's technical report on the evolution of the parallel computing research, published in 2006, states:

"We believe that future successful programming models must be more human-centric. They will be tailored to the human process of productively architecting and efficiently implementing, debugging, and maintaining complex parallel applications on equally complex manycore hardware." [1]

We consider that our approach is conforming to the trend foreseen here. We are extending programming tools to facilitate the deployment of the subjective expertise which humans/scientists have in their research fields, into the process of analysing the performance of their applications on HPC systems.

The key point of *Domain Knowledge* is to offer a framework which provides both a useful guidelines, as well as the necessary flexibility in order to first integrate domain specific elements into applications and then to use them in the analysis process. These two features are achieved through the combination of a familiar syntax for an annotation language on the one side, and three basic constructs on the other.

Moreover, DK fosters a global view of scientific applications as well, supporting metrics which address global performance of programs.

In the following chapters we detail on the implementation and evaluation of DK. We present the markup language LaDoK in chapter 5, an implementation of a framework supporting DK in 7 and real use case results in chapters 8 to 9.

5 LADOK - LANGUAGE FOR DOMAIN KNOWLEDGE

In this chapter we present LaDoK , the *Language for Domain Knowledge*. The purpose of LaDoK is to support developers in expressing the domain specific information and skills they hold, in order to use them further in the *Domain Knowledge* approach.

5.1 LANGUAGE FEATURES

The design of LaDoK is based on five main features, which we identified as important language features in our given context. They resulted from both the analysis of the related work, as well as from common practical experience in the performance measurement and analysis in the HPC field:

- *familiar syntax*: a rather general requirement for languages is to have an easy to learn syntax. We have to take into consideration the fact that LaDoK will be used in addition to common GPLs, and thus the extra learning effort should be kept at minimum.
- *unintrusive relative to source code*: there should be a clear distinction between the source code of the scientific application and the DK related insertions in the code. Also, it should be very clear that LaDoK only marks-up/highlights the information and does not interfere with it.
- *orthogonality*[1]: LaDoK is an instrument used by developers in the DK context in order to make *explicit* the *implicit* domain specific issues. The orthogonality of the language would promote a precise and clear means of expression.
- *composability*: LaDoK addresses scientific code and it is known that scientific applications use overy often special libraries or re-use own application parts developed in related research tracks. LaDoK should support a component-based application design.
- *flexibility* and *extensibility*: the main usage context of LaDoK is the performance analysis process in the broader HPC context. As this is itself a dynamic field, with changing requirements, it is necessary that the new language is prepared to support changes and extensions.

In order to respond to all these requirements, both at the syntactical as well as the semantic level, we designed LaDoK to closely follow the line given by the well known typesetting language LaTeX.

5.1.1 *LaTeX-like language*

LaTeX is by far the first choice, when it comes to scientific papers and books. Hence, most scientists are familiar with the language, its syntax and semantics. These same

1 For a definition of orthogonality for programming languages, see [63]

scientists should apply now LaDoK in the source code, to highlight information which is closely connected to the information which they markup with LaTeX, maybe on the very paper describing the results they obtained with the application. There is an obvious similarity between these two tracks. We aim at keeping it in the actual markup language as well.

Besides the familiar syntax, LaTeX also has the advantage of being very distinct from the common GPLs, and thus the second requirement is also fulfilled.

As with respect to orthogonality, LaTeX is probably not the best candidate, since there are many situations where more than a command exists for the same functionality. Even more, the final typesetting of the document depends on an entire combination of commands and parameters. Often, changing a parameter of a command will influence the result of other commands too. At this point we count on the simplicity of LaDoK for yielding a better orthogonality than LaTeX usually does.

The last three requirements are "naturally" supported, by the way LaTeX is built: composability - one can include other files in the current one; flexibility - there is support for defining new commands; and extensibility - the support for the packaging system.

5.2 LADOK SYNTAX

The LaDoK syntax is mostly identical to the LaTeX syntax, there are only few other keywords and a different order for the optional parameters block, as we describe in this section.

LaDoK is a markup language and, hence, there is no stand-alone LaDoK file, but rather source code files written in common GPLs and enhanced with Domain Knowledge based markups.

The markups are defined through LaDoK commands. The general syntax of one command is:

```
        \commandName{paramSet1}{paramSet2} ... [optParamSet]
```

The parameter sets are comma-separated lists of either single parameter names or pairs of parameter name and parameter values:

```
        paramSet: parameterName1, parameterName2, ...
        paramSet: parameterName1 = parameterValue1, parameterName2 =
            parameterValue2, ...
```

Although following the same syntax, there are two distinct types of commands in LaDoK : *stand-alone* and *coupled* commands. A *stand-alone command* has no other restrictions besides the given syntax and it refers to the global context. See the `\phase` command below.

The *coupled commands* must be given as pairs. They can be nested but not interleaved and function as a container or context for the defined properties and action. They correspond to the environment commands in LaTeX. See, for example, the `entity` definition below.

Listing 7 gives the grammar of LaDoK in the BNF notation.

```
construct:      phase | entity | operation

phase:
                "\phase{" PHASENAME "}" |
                "\phase{" PHASENAME "}[" phaseattrlist "]"
phaseattrlist:
                phaseattrval |
                phaseattrval "," phaseattrlist
phaseattrval:
                "type" "=" ( "communication" | "computation" | "I/O" ) |
                STRING "=" STRING

entity:
                ( "\begin{entity}{"ENTITYNAME"}" |
                "\begin{entity}{"ENTITYNAME"}["entityattrlist"]" )
                statements
                "\end{entity}"
statements:
                statement |
                statement statements
statement:
                construct | GPL-STATEMENT
entityattrlist:
                entityattrval |
                entityattrval "," entityattrlist
entityattrval:
                "type" "=" ( "dynamic" | "static" ) |
                "typevar" "=" VARIABLENAME [    "typefield" "=" STRING] |
                STRING "=" STRING

operation:
                ( "\begin{operation}{" OPERATIONNAME "}" |
                "\begin{operation}{"OPERATIONNAME"}["operationattrlist"]" )
                statements
                "\end{operation}"
operationattrlist:
                operationattrval |
                operationattrval "," operationattrlist
operationattrval:
                "type" "=" ("commcontrol" | "commdata" | "commsend" |
                        "computation" | "initialization") |
                "param" "=" VARIABLENAME |
                "linkedentity" "=" ENTITYNAME |
                "linkedentityset" "=" ENTITYSETNAME ["idx" "=" VARIABLENAME] |
                "iterometervar" "=" VARIABLENAME |
                "dstproc" "=" VARIABLENAME |
                "checkop" "=" ( "SUM" | "PROD" | "AVG" ) |
                STRING "=" STRING
```

Listing 7: Grammar of LaDoK .

```
        program simplevectors
                integer :: i, j
                \begin{entity}{myVectors}
                        real, dimension(10) :: a
                        integer, dimension(:), allocatable :: b
                \end{entity}

                j = 3
                a = (/ (i*2, i=1,10) /)
                allocate(b(10))
                b = j*a;
        end
```

Listing 8: Simple entity definition within a Fortran code fragment.

5.3 LADOK CONSTRUCTS

LaDoK is composed from a small set of constructs, which leverages a good orthogonality. The constructs closely follow the DK Concepts presented in section 4.3, with some extensions imposed by the implementation aspects.

5.3.1 *Entities*

A DK Entity is defined using the following construct:

```
\begin{entity}{entityName}[parameterList]
        % GPL source code containing
        % definitions of variables
        % ...
\end{entity}
```

where `\begin` and `\end` are LaDoK commands and `entity` is a special keyword, which has to appear between the first curly brackets. `entityName` and `parameterList` are to be replaced with the corresponding string values.

Entities regard data structures declaration.

All data structures defined between the `begin` and `end` commands are going to be assigned to the entity identified by `entityName`. Listing 8 gives a minimal example of an entity definition within a Fortran code. For details regarding our solution for implementing the variable-to-entity assignments, as well as the entity identification, please refer to chapter 7.

The semantics of the LaDoK Entity construct complies to the minimal requirement of DK Entities of exposing a container-like characteristic, and even add a few more important features:

CONTAINER-LIKE : first of all, assigning data structures to an entity is as straight forwards as surrounding their definitions between the `\begin{entity}` and `\end{entity}` constructs.

Secondly, more data structures can be added to an existing container by using the same `entityName` at any subsequent entity definition. In this sense, the first definition of a particular entity is treated as such, while any further definitions are treated as extensions to the previous ones.

ROBUSTNESS : a very important feature for a markup language is to be able to cope with various code structures and constraints. Developers should not be forced to rewrite or restructure the code especially for being able to apply the markups.

The entity definition fulfils this requirement, in the sense that it is robust to "foreign" statements appearing inside the `\begin` ... `\end` pair. All programming statements referring to variable definitions are accounted for registration to the current entity, while any other programming statement is being ignored.

HIERARCHIC : many natural, as well as abstract structures, expose a hierarchical characteristic. There are entities which logically belong, or are part of other larger entities. They could be called "sub-entities".

LaDoK does not use an extra definition for "sub-entities". It rather supports this feature through the *nesting* concept. An entity definition which is enclosed inside another entity definition preserves the natural semantics of being a "sub-entity" of the enclosing one.

Any LaDoK implementation thus has to offer support for hierarchical entity identification. Here again, we refer to chapter 7 for details regarding our implementation solution.

Listing 9 illustrates the features above using a C code snippet:

1. *container* $\rightarrow$ the *Collection* entity is defined at lines 2-11 and then extended with 2 more variables at the lines 15-18;
2. *robustness* $\rightarrow$ other programming statements besides the variable declarations are also allowed, like those at the lines 5-6;
3. *hierarchic* $\rightarrow$ the *Index* entity definition at lines 8-10 is nested within the broader *Collection* definition, and thus *Index* is a "sub-entity" of the *Collection*.

DK Entities can receive a list of attributes, as shown in listing 7. The grammar gives a set of predefined attributes, but it also leaves the possibility to freely define new attributes. The only constraint is the attribute-name/attribute-value pairwise declaration. Depending on the actual scientific domain of the application, as well as on the programming tools which use the annotations, there might be new attributes required.

The predefined attributes are as follows:

- *type:* defines a type for the entity. Allowed values are `dynamic` and `static`. Dynamic entities can receive further attributes via DK Operations of type `initialization`.
- *typevar:* defines the variable registered with the current DK Entity whose value is going to represent an identification for the DK Entity at runtime. See section 10.3 for an example.

```
...
\begin{entity}{Collection}
        int* a;
        const int MAX_SIZE = 1000;
        a = (int*)malloc(sizeof(int), MAX_SIZE)
        read_in(a);
        double pi = 3.14;
        \begin{entity}{Index}
                int* idx1;
        \end{entity}
\end{entity}
...

% possibly in another file
\begin{entity}{Collection}
        int* b,c;
        initialize(a, b, c);
\end{entity}
...
```

Listing 9: An entity definition example, featuring extension, nesting and statement interposition within a C code snippet.

- *typefield:* used only in combination with `typevar`, it defines the field of the *identification variable* to be used as the representing value. In order for this to work, the identification variable should have a composed type, like structure or user defined types in Fortran.

5.3.2 *Operations*

The way DK Operations are defined using LaDoK is very similar to the entities definition:

```
\begin{operation}{operationName}[parameterList]
        % GPL source code containing
        % any executable statements,
        % declarations, or definitions
        % ...
\end{operation}
```

where `\begin` and `\end` are LaDoK commands and `operation` is a special keyword, which has to appear between the first curly brackets. `operationName` and `parameterList` are to be replaced with the corresponding string values.

All executable statements enclosed between the `begin` and `end` commands are going to be accounted for the `operationName` operation.

The observations regarding the extension and the nesting of entities definition apply for the operations definition too:

- repeating the same `operationName` in a subsequent definition *extends* the previous one;

- nesting operation definitions results in the inner operation to be treated as *sub-operation*, or part of the outer operation.

There are two important aspects which are different from discussions on entities, and which we would like to present below.

Operations can be linked to entities.

First of all, operations can be *linked* to entities. This is accomplished by adding the attribute `linkedentity` having as value the name of the entity to which the current operation has to be linked with:

```
\begin{operation}{Distribution}[linkedentity=Collection]
        % ... some statements here
\end{operation}
```

The importance of this feature for the performance analysis will be discussed later in the chapters presenting the use cases.

Operations include various types of programming statements.

The second aspect concerns the usability of the actual definition of an operation, very much like the discussion on the robustness of entities definition. While the latter ones target only one type of programming statements, namely the declarations of variables, in the case of operations definition, there is a considerably difference of the required behaviour. One distinguishes here several use cases (note that by function we generally define subprograms - thus also including procedures or methods; and by header files we generally refer to the file containing declarations - these could also be modules or any other types):

- *function declarations*: there are cases where all functions of a given application component should be categorized as being one single operation. In this case, it is desirable that the operation is defined where all functions are declared together, e. g. in the header file, and not necessarily where they are implemented/defined. Hence, operation definitions have to recognize function declaration statements.

```
% function declarations
\begin{operation}{LibraryOperation}
double processDoubles(double* a, int size);
int processInts(int* b, int size);
\end{operation}
```

- *function definitions*: there are also cases where there are no such previous declarations of the functions, or where it is simply more convenient for the developer to just mark the operation in the current working file, than to search for the corresponding header files. For this case, operation definitions have to recognize function definition statements.

```
% function definition
\begin{operation}{ProcessArray}
double processDoubles(double* a, int size){
        for(int i=0; i < size; i++){
                a[i] = % ...
        }
        % ... other statements
}
\end{operation}
```

- *function calls*: finally, there are cases where only a code fragment should be categorized as a particular operation. It follows, that operation definitions also have to recognize function call statements, as well as common execution statements.

```
% code fragments and function calls
double processDoubles(double* a, int size){
        \begin{operation}{Interpolation}
        for(int i=0; i < size; i++){
                a[i] = % ...
        }
        normalize(a); % function call
        \end{operation}
        % ... other statements
}
```

Other attributes which can be set for DK Operations are given in listing 7. Similar to the case of DK Entities, the list of predefined attributes can be extended according to the deployment necessities.

The predefined attributes for DK Operations are:

- *type:* gives the type of the operation. Accepted values are `commcontrol`, `commdata`, `commsend`, `computation`, and `initialization`. All these values were defined in relation to their usage within performance analysis tools. For the semantics of the `commcontrol` and `commdata`, please see chapter 9. The `computation` and `initialization` are well described in section 10.3, while the `commsend` is explained in the implementation section 7.4.2.

- *param:* receives as value the name of a variable. Its semantics is given only in relation with the `type` attribute.

- *checkop:* provides the mathematical operation which will be performed on the variable set by the `param` attribute. This is used in DK Operations of type `commdata`.

- *linkedentityset:* like `linkedentity` described above, but it links an `entityset`. We introduce `entitysets` below, in section 5.4.1.

- *idx:* used only in combination with an `entityset`, it receives as value the name of the variable which is used as the index of the respective `entityset`.

- *iterometervar*: receives as value the name of a variable which is to be considered further as the *Iterometer* of this DK Operation. We defined *Iterometers* in section 4.4.

- *dstproc:* used only for DK Operations of the type `commsend`, it accepts as value the name of the variable which defines the destination process of the communication.

5.3.3 *Phases*

Defining DK Phases in LaDoK is achieved by issuing the `phase` command at the desired place in the source code:

```
% ...
\phase{phaseName}[parameterList]
% ...
```

where `\phase` is a command and `phaseName` has to be replaced with the corresponding string value.

As explained in section 4.3, phases are not container-like concepts as entities and operations, but rather represent a global status of the application. In this sense, the execution context of the application is set to be the `phaseName` phase from the point where this command is encountered. The context will be changed again when execution meets the next `\phase` markup.

Phases cannot be nested. Any new `\phase` just starts a new phase. Nevertheless, the application can enter one particular phase several times during the execution. This can happen if the same `\phase` call is executed several times, or if the same `phaseName` is being used for different `\phase` calls.

The grammar in listing 7 provides one predefined attribute for DK Phases, the `type` attribute. The actual semantics of the given attribute values are actually depending on the particular programming tool where DK is applied. A good example are the profiling tools, which might use such type specification for a better targeted measurement process.

5.4 extensions

In this section we consider two extensions which could be added to LaDoK, namely the *entitysets* and the *data type annotations*. We show that the first one is a natural addition to the DK Concepts, while the latter does not really follow the DK design guidelines.

5.4.1 *Entitysets*

Besides the three main concepts represented by the DK Constructs, LaDoK can also provide a derived concept: `entityset`. An `entitset` is not a "stand-alone" concept in the sense of the semantics described by the DK approach, but it is rather a convenience extension, mainly motivated by the *granularity of the semantics* and the *limitations/characteristics of the implementation*.

In many scientific applications, the actual algorithms or computations are performed on a *group of elements*. Such are, for example, applications working with clusters of atoms, or molecules, with sets of particles, or grid units. In order to apply the DK approach, one is provided in this case mainly with two options: either mark the entire group as one DK Entity, or mark each element of the group as an entity by its own. This is the granularity of the semantics.

At the source code level, when taking into consideration programs written in common GPLs, such groups of elements are represented as arrays. Either as one array of structures/objects, or as several arrays storing different properties of the elements and sharing some index for proper alignment. Hence we have different implementation characteristics.

In LaDoK though, `entities` annotations only allow single entity definitions. Thus, arrays could only be marked together as one large entity

```
\begin{entity}{AllParticles}
        particle_type* particleTypes;
        pos_type* particlePositions;
\end{entity}
```

In order to apply a finer semantics granularity, LaDoK can also provide the derived construct `entityset`. The code snippet above thus becomes:

```
\begin{entityset}{Particles}
        particle_type* particleTypes;
        pos_type* particlePositions;
\end{entityset}
```

Each array declaration between the `\begin` and `\end` commands will be assigned to the given `entityset`. The elements at a given index i from all arrays will be registered as one single entity.

For the example above, `particleTypes` and `particlePositions` represent the `Particles` `entityset` and one entity `Particles[i]` consists of the two elements `particleTypes[i]` and `particlePositions[i]`.

The index of an entityset can be defined at the moment when the entityset is used. One defines a DK Operation and gives the two attributes `linkedentityset` and `idx`, as described above in section 5.3.2.

5.4.2 *Data type annotation*

One question arising when designing LaDoK was: *Should LaDoK also annotate data structure types, or limit only to the data structure declarations?*

Annotating a data type would then look like in the listing below:

```
\begin{entitytype}{Molecule}
        typedef struct{
                oribitals_t* orbitals;
                position_t pos;
        } molecule_t;
\end{entitytype}

molecule_t MoleculeA;
molecule_t MoleculeB;
```

and declaring variables of the annotated type should result in the DK framework registering them as DK Entities.

The listing above would target the same effect as the case below, where the data structure declarations are being annotated instead of the data type declaration:

```
\begin{entity}{MoleculeA}
molecule_t MoleculeA;
\end{entity}
\begin{entity}{MoleculeB}
molecule_t MoleculeB;
\end{entity}
```

There are several benefits which data type annotations could bring to LaDoK:

- there would be less annotations required;
- it complies to the "programming spirit";
- offers the possibility to address substructures like the fields of a custom type.

In the same time, there are more important issues which plead against such annotations:

- it is not following the main DK design guidelines of *directly explaining data*;
- it would be difficult to identify entities, especially if one combines both entity- and entitytype- declarations.

Further investigations using real code implementations as study cases might bring more light into this issue.

6 LADOK FRAMEWORK

In this chapter we present our LaDoK framework implementation. The main goal of the framework is to deliver the DK information associated to an application in such a form that it can be directly used by advanced programming tools. The framework receives as input an application enhanced with LaDoK annotations. As output, it delivers a set of DK objects and a new instrumented version of the application, which should be used together later by the advanced programming tools.

The main components of the framework are:

1. LDK Extractor, including:
    a) LDK Parser;
    b) LDK Directives Processor; and
    c) LDK Instrumenter.
2. LDK Interface Library; and
3. LDK Objects.

Figure 13 presents an overview of these components, along with the data and control flows.

The LDK Extractor is responsible for extracting the DK information from the source code enhanced with LaDoK annotations. The LDK Objects component acts like a database. It stores the DK objects identified and instantiated by the LDK Extractor. The LDK Interface Library provides the necessary functionality for handling the DK objects.

Although not a component by itself, an important part of the framework is represented by the new instrumented source code which is generated in the end, along with the DK objects in the database. These instrumentations are necessary at runtime to provide dynamic information and thus support the management of dynamic DK objects, as well as of some property changes of the static DK objects stored in the database.

6.1 LDK EXTRACTOR

Our implementation of the LDK Extractor is based on *f90inst*, the Fortran instrumenter of the Periscope tool introduced in section 2.4. *f90inst* is implemented in C and uses for the source code parsing the NAGf90 library from the Numerical Algorithms Group (NAG). *f90inst* can parse Fortran files, one at a time, and insert Periscope specific instrumentation calls.

We integrated the LDK Extractor based on the native instrumentation process of *f90inst*, as seen in figure 14.

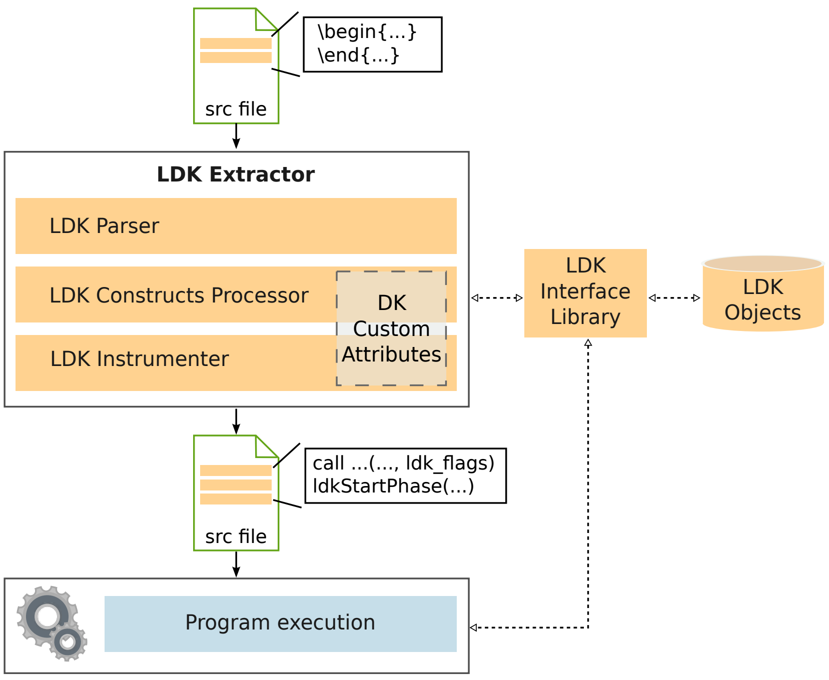


Figure 13: The main components of the LaDoK Framework.

Following the syntax and semantics described above in sections 5.2 and 5.3, developers manually instrument the source code of their application with LaDoK annotations. The framework uses first the NAGf90 library functionality to read-in the file and build the *parse tree (PT)* of the source code.

In the next step, *f90inst* starts traversing the parse tree in order to operate Periscope specific instrumentation. We connect here the LDK Extractor by means of a test command performed on the currently visited node. There are three types of nodes which switch on LDK processing:

1. nodes containing LaDoK annotations;
2. nodes containing routine calls; and
3. nodes containing routine declarations.

If the currently visited node is of one of these three types, then the instrumenting process enters the execution defined by the LDK Extractor. More precisely, there is one particular flow sequence defined within the LDK Extractor for each type of node. See figures 15, 17 and 18, which we describe in more detail in the sections below.

During its execution, the LDK Extractor generates DK objects, or uses the existing ones, depending on the particular flow sequence.

From the LDK Extractor, the execution flow returns back to *f90inst*. Upon finishing the parse tree traversal, source files with new instrumentations are being generated.

This entire process is repeated twice for the entire application, thus we talk about two processing phases:

1. **First processing phase.**
   In the first run, the LaDoK annotations are "consumed" and replaced, where necessary, with LDK Interface Library calls. The DK objects are created as well.

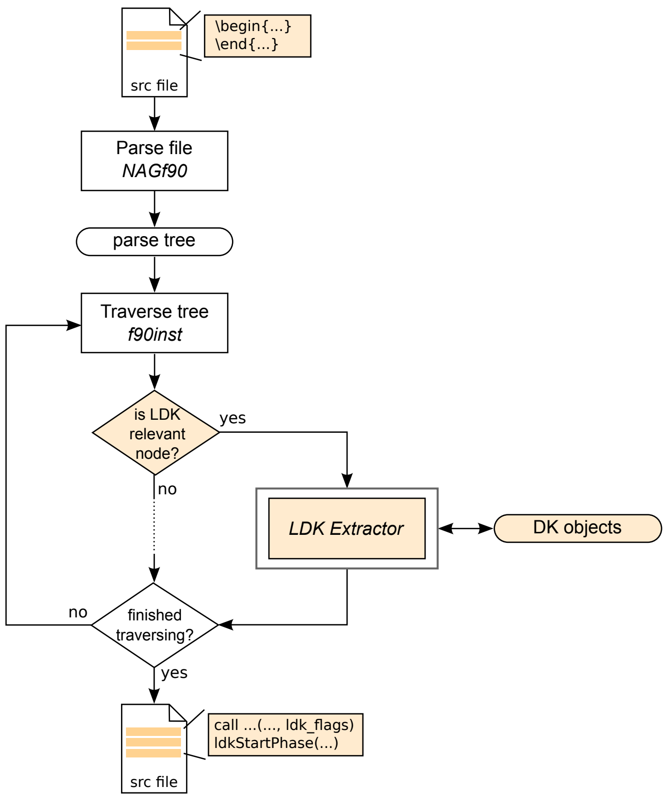


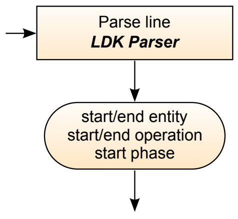
Figure 14: Integration of the LDK Extractor into the *f90inst* execution process.

In this first phase, only the first of the three types of nodes above are used to enable the LDK Extractor (figure 15).

2. **Second processing phase.**
The second run is dedicated to instrumenting the source code with support for dynamic information and dynamic objects. In this second phase, the nodes of the types 2 (figure 17) and 3 (figure 18) are used to switch to the LDK Extractor. The instrumentation also takes into account the information stored by the DK objects generated in the previous phase.

The separation in two processing phases is necessary due to the global character of the DK Objects. Before running any automatic instrumentation of the code in the LDK Instrumenter, the entire DK information needs to be gathered and this is fully accomplished only at the end of the first processing phase.

In the following subsections we present the three components of the LDK Extractor: the LDK Parser, the LDK Directives Processor and the LDK Instrumenter. Starting from the two processing phases described above, we show for each of them the main functionality and implementation strategy.



6.1.1 *LDK Parser*

The LDK Parser comes into play during the first processing phase and it is the first component being called in the execution flow, as seen in figure 15 as well.

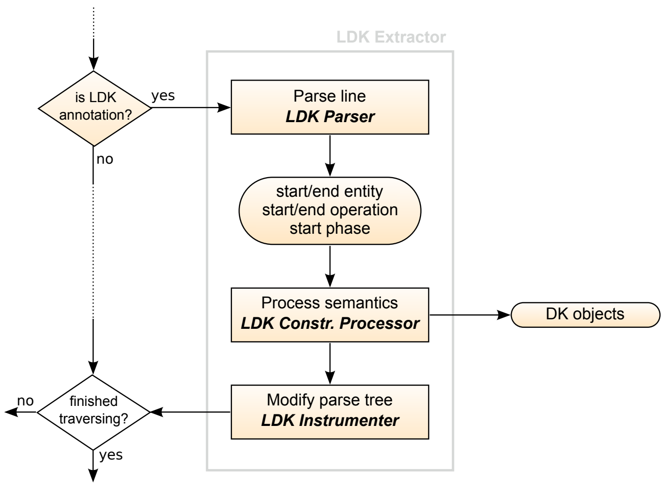


Figure 15: Integration of the LDK Extractor into the *f90inst* execution process. Blue boxes belong to *f90inst*, while orange boxes belong to the LDK components.

As already mentioned, the input file is read-in by the NAGf90 library, which generates the parse tree of the source code. In this tree, all LaDoK annotations are stored as comment nodes. The convention is to prepend every LaDoK annotation line with the identifier `!$LDK`. For example, annotating the start of a new phase in the source code would look like this:

```
!$LDK \phase{MyComputationPhase}[type=computation]
```

If the node is a comment node and it starts with the predefined sequence `!$LDK`, then it is a node which switches to LDK processing. It is the first type of nodes listed above.

The first step in this sequence is parsing the entire annotation line. Being given as a comment line in the source code, the annotation is considered by the NAGf90 parser as one single token and stored within a single node. What the LaDoK parser needs to do now, is to split the line into valid tokens and interpret them based on the LaDoK syntax. This is generally known as *lexical analysis* and *syntactic analysis*, which are the common components of any parser.

The LDK Parser uses Lex and Yacc [9] to automatically generate the appropriate parsing routine for the annotation lines. Lex is a lexical analyzer generator. The generated routine is used to split or "tokenize" input strings into *tokens*. Yacc is a parser generator. The generated function performs a syntactical analysis on a "tokenized" input, based on a defined grammar.

Figure 16 presents a scheme of how `lex` and `yacc` work together. We adapted the initial figure from [9] to depict the particular case of our framework.

The *lexical rules* which `lex` transforms into a callable routine are given in our framework in the file `ladok.l`. It contains the definition of all tokens which have to be identified in the input string line, see listing 10.

The *syntactic rules* are provided in the `yacc.y` file. The syntax of `yacc` is similar to the common BNF grammar definitions. In addition, it allows to declare for each grammar rule a code snippet as well, which is going to be executed every time this rule is applied. For example, listing 11 shows the rules which match the annotation marking the beginning of an entity. Every time the rule is applied to the given anno-

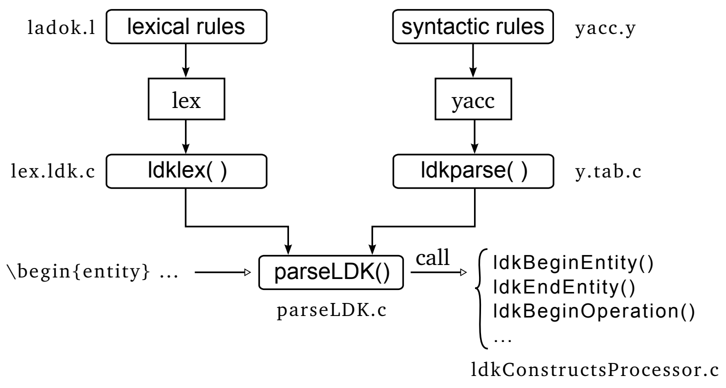


Figure 16: Usage of *lex* and *yacc* to generate the parsing function for processing LaDoK annotations. The two generated functions are used in the central parsing function of LDK Parser to process the input annotation line. The results of this process are calls to corresponding routines in the LDK Directives Processor.

tation line the call to `ldkBeginEntity()` is also going to be executed. As can be seen in the function calls, the tokens which are identified in the input string are available as special variables in the code snippet.

One could observe in listing 11 that we used in the rules definition some aliases. For example, instead of `T_LBRACE` as defined in listing 10 we used `"{"`, and instead of `T_COMMA` we used `","`. This is a convenience option in `yacc` which provides a better readability to the list of rules and can be easily implemented by adding to the list of tokens declarations in `yacc.y` the corresponding aliases:

```
%token T_LBRACE "{"
%token T_RBRACE "}"
%token T_COMMA ","
```

In section 5.2, when we introduced the syntax of LaDoK , we also provided the grammar of the language in BNF format, in listing 7. It is important to observe that the grammar defined here, in the `yacc.y` file, is a derived grammar from the initial one. The reason behind this is the fact that in our framework we are parsing the LaDoK annotations while traversing the source code parse tree node by node. As explained above, we are actually feeding the LDK Parser only one annotation at a time. Hence, the input for our parser in most cases does not represent a valid LaDoK construct, but only one fragment of a construct.

More precisely, we split the rules for *Entity* and *Operation* from the initial definitions including both the `start` and the `end` commands, into pairs of rules, one pair for each construct and one rule for each of the `start` and `end` commands. Listing 12 gives a comparison between the rules used to define an entity in the LaDoK grammar and those used in the syntactic specifications in the `yacc.y` file. It can be seen that the latter ones do not refer to the source code statements enclosed by the entity, the `GPL-STATEMENTS` in the grammar. This is because these statements do not even get to the parser, since only comment lines beginning with the `!$LDK` identifier are being considered here.

The new grammar has the advantage of being able to process annotations line-by-line, as provided by the traversal of the parse tree of the source code. However, it does not cover the entire syntax definition of LaDoK . For example, the new grammar

```
%%
\\phase          return T_PHASE;
\\begin          return T_BEGIN;
\\end            return T_END;
entityset        return T_ENTITYSET;
entity           return T_ENTITY;
operation        return T_OPERATION;
\{               return T_LBRACE;
\}               return T_RBRACE;
\[               return T_LBRACKET;
\]               return T_RBRACKET;
=                return T_EQ;
,                return T_COMMA;
[0-9]+           return T_NUMBER;
[a-zA-Z][a-zA-Z0-9]*  {
                        ldklval.string=strdup(yytext);
                        return T_WORD;
                      }
[ \t]+           /* ignore whitespace */;
\n               /* ignore */;
```

Listing 10: Declaration of tokens used by LDK parser. File: `ladok.l`.

```
/*
rules matching beginning of entity:
\begin{entity}{MyEntity}[refvar=myvar]
*/
beginentity:
        T_BEGIN "{" T_ENTITY "}" "{"T_WORD "}" "[" attributeslist "]"
        {
           ldkBeginEntity(actNode, $6);
        };

attributeslist:
        attributevaluepair "," attributeslist
        |
        attributevaluepair ;

attributevaluepair:
        T_WORD "=" T_WORD
        {
           ldkAddAttribute($1, $3);
        };
```

Listing 11: Definition of rules which match the annotation for the beginning of an entity. File: `yacc.y`.

```
% rules defining a DKEntity in the LaDoK grammar
dkconstruct:
                dkentity |
                dkoperation |
                dkphase ;

dkentity:
                beginentity statements "\end{entity}" ;

beginentity:
                "\begin{entity}{" ENTITYNAME "}[" entityattrlist "]"
                |
                "\begin{entity}{" ENTITYNAME "}" ;

statements:
                statement | statement statements;

statement:
                dkconstruct | GPL-STATEMENT ;



% rules in yacc.y for parsing annotation lines
dkconstruct:
                beginentity | endentity |
                beginoperation | endoperation |
                dkphase ;

beginentity:
                "\begin{entity}{" ENTITYNAME "}[" entityattrlist "]"
                |
                "\begin{entity}{" ENTITYNAME "}" ;

endentity:
                "\end{entity}" ;
```

Listing 12: Comparison between the definition of entities in the LaDoK grammar and the definitions used in the syntactic specification for yacc.

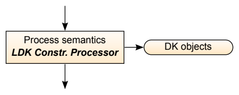
allows that a `beginentity` statement is immediately followed by a `endoperation`. Thus, in the new grammar the following construction is syntactically correct:

```
\begin{entity}{MyEntity}
\end{operation}
```

Seen from the level of the entire source file, the way we process the LaDoK annotations is similar with how an event-based parser works. For example, a SAX parser for XML reports a start or end event each time it encounters a opening or closing tag. In the same way, we report if the annotation line is starting or closing a DK construct. By default, this kind of parsing does not have any "memory" of what happened before. It cannot decide whether a closing event is valid at the place where it is encountered or not.

The solution commonly used in this case is to use a stack data structure to "memorize" past events. We implemented the same solution using a stack of *environments*, `envStack`. Every time a beginning of an entity or operation is encountered, the DK construct is *pushed* into the stack. Every time an ending annotation is encountered, first a test is performed to check whether the current ending construct is matching the construct at the top of the stack. Since the `end` commands do not specify particular instances of DK constructs, it is enough to compare the type of the ending construct with the type of the construct in the stack. If the types match, then there is a syntactically valid ending operation. A *pop* operation is then performed on the stack. In this way we ensure the proper nesting of DK constructs and avoid interleaving.

6.1.2 *LDK Directives Processor*



The LDK Directives Processor is the semantics analysis component of the LDK Extractor. It comes into play in the execution flow described in figure 15, after the parsing action. After having parsed the current LDK annotation line, the next step to be fulfilled is to apply the corresponding semantics.

We described the semantics of the DK concepts in section 5.3 and we showed in the previous section, that the output of the LDK Parser component is represented by three types of events:

1. start phase;
2. start/end entity; and
3. start/end operation.

The LDK Directives Processor receives these events and applies a series of actions such that in the end the semantics of the corresponding DK Concepts are fulfilled. It is important to observe here that the semantics of the LDK Directives is an adaptation of the semantics of the DK Concepts, based on the implementation restrictions. This is closely related to the fact that the actual implementation grammar of the LDK Directives is also an adaptation of the original LaDoK grammar, as we already described in the previous section.

For example, the LaDoK semantics gives one definition of a *DK Entity*, while the set of events passed to the LDK Directives Processor contains two types of events related to entities: `start entity` events and `end entity` events. The semantics of the LDK

Directives is thus deriving from and fulfil the overall semantics of the DK Concepts, but is not identical.

In general, the `start` events are related to generating the DK Objects and storing the declarative attributes, like the name and the type. The `end` events are then related to storing the structural information, like the enclosed code variables or routines.

DK Phase = generate object + insert phase trigger.

The DK Phase concept is the most simple concept available in LaDoK. Its semantics is that of a markup signalling that the execution flow entered a defined particular phase.

The DK Phase has only a corresponding `start` LDK Directive. Upon encountering such an event, the LDK Directives Processor generates a new DK Object for the phase, if this is its first occurrence, and it stores along the attributes declared within the LDK Directive.

It then passes the `id` of the phase to the LDK Instrumenter, which inserts the corresponding call in the parse tree for triggering the start of this phase at runtime.

DK Entity = generate object + gather variables.

The semantics of the DK Entities is more complex than the one of DK Phases. Besides the declarative attributes, like name and type, DK Entities also have the structural related meaning, namely that all variables declared between the begin and end markups of a DK Entity are assigned to this entity.

We have two corresponding events triggered by the LDK Parser for entities: a `start entity` and an `end entity` event.

The semantics associated with the `start` event is similar to that for DK Phases: the new DK Object for an entity needs to be created, if it does not already exists, and the name and all the declared attributes have to be stored along in the DK Object.

The remaining part of the semantics of the DK Entities, namely the one referring to the enclosed variables, is then applied when the `end entity` event is encountered. This is a more complex aspect than the one regarding the declared attributes. The complexity comes from the fact that one needs to analyse here the actual structure of the source code, whereas in the previous case the attributes of an entity were directly provided through the LDK Directive and passed along in the event call.

Given now that the LDK Directives Processor encountered an `end entity` event, how does it find all the variable declarations included between the start and the end directives for the current DK Entity? Regardless of the programming language of the source code, this information can be retrieved from the parse tree. It is sufficient to know the node corresponding to the `start` markup and the node corresponding to the `end` markup.

In our framework, the parse tree is already provided by the Periscope instrumenter. Since we use an event-based parsing method for the LDK annotations, we need again a mechanism to memorize the information provided in past events. For example, upon encountering an `end entity` event, we need to know the node in the parse tree at which the corresponding `begin` event was encountered.

We therefore make use again of the `envStack` data structure introduced in the parsing process of the LDK Parse component. When a `start` event is encountered, the stack is fed not only with the kind of the directive, needed for the syntactic analysis described above, but also with the reference to the begin node. The listing below shows the definition of the type of elements which are stored in the `envStack`:

```
typedef struct {
    char*       name;
```

```
        enum EnvType        type;
        Inode       beginnode;
        UT_array* attributes;
    } EnvElem;
```

For DK Entities, the `beginnode` field is initialized when the `start entity` event is encountered. When an `end entity` event is encountered afterwards, it means that the DK Entity which is at the top of the stack has to be processed from a semantic point of view. More precisely, one needs to identify the variables declarations in the source code and register them to the current DK Entity.

We use in our implementation the low-level operations provided by the *NAGf90* library to move within the parse tree:

- NEXT(): returns the right sibling of the given node, or 0 if there is no sibling;
- DOWN(): returns the left most child of the given node, or 0 if there is no child;
- UP(): returns the parent node of the given node, or 0 if this is the root.

As already mentioned, our implementation supports Fortran codes. For declarations of variables, we considered the following four cases and all combinations of them:

- simple variable declaration:
  `INTEGER :: simplevar, anothervar;`
- declaration of arrays:
  `INTEGER, DIM(:), ALLOCATABLE :: array;`
- declaration with initial values:
  `INTEGER, DIM(:):: initarray = (/1,2,3/)`
- declaration of pointers:
  `POINTER, INTEGER :: mypointer;`

Each type of declaration generates a different parse sub-tree which requires a particular implementation in the tree traversal.

DK Operation = generate object + gather routines.

The semantics of the DK Operations is very similar to the one of the DK Entities. First of all, there are the type, the name and other attributes declared in the `begin operation` directive and passed directly to the LDK Directives Processor through the `start operation` event.

A particular aspect is given here by the *reference attributes*. We call reference attributes, those attributes which receive as value a reference to another element or object, usually the name of that element or object. For example, the `type` attribute is not a reference attribute, as its value does not represent a source code element or DK Object. In contrast, the value assigned to the attribute `dstproc` in listing 13, refers to the actual variable `j`, which is an element in the source code. We thus call the `dstsrc` attribute a *reference attribute*. The same holds for the `linkedentity` attribute in the same listing, which points to a DK Entity, which is a DK Object.

Processing the semantics of such attributes extends the simple DK Object creation and storage with an extra validation which has to be performed in forehand. The LDK Directives Processor thus has to run one of the two types of tests to semantically validate this kind of attributes:

```
j = get_worker()

\begin{operation}{CommExample}{type=Comm_Send, dstproc=j, linkedentity=MyEntity}

call MPI_Send(myArray, length, INTEGER, j, myTag, comm, ierror)

\end{operation}
```

Listing 13: Semantics of DK operations covering dynamic aspects.

1. validate source code references; or
2. validate DK Objects references.

The first type of tests has to check whether the referenced variable exists, and if so, whether it is defined in the scope where the DK Operation is declared.

In our implementation, we perform this test using the symbol table provided by the *NAGf90* parser. The test ensures only that the variable is defined within the current Fortran subprogram. It does not ensure though, that the variable is also accessible in the scope where the DK Operation is declared, if there is such a particular scope defined. Nevertheless, since in Fortran the variable declarations are to be found at the beginning of a subprogram, our implementation covers the major cases found in applications. For all other cases, a compilation error will be thrown for the undefined variable, since it will be used in the instrumentation calls inserted for the DK Operation.

For the second type of test, one needs to validate whether the referenced DK Object is defined in the context of the current application. Although this check could be easily accomplished with a query to the LDK Objects component, the overall semantics of the DK Objects and the current parsing implementation makes the things a bit more complicated. First of all, we explained in section 4.3 that the DK Concepts have global semantics, which means that the DK Objects are defined in the global scope, as well. This means that the DK Object referenced by the current attribute of a DK Operation does not necessarily need to be defined in the scope of the currently processed source file. We recall that the LDK Parser processes only one file at a time.

We thus cannot fully validate semantically the attributes referencing a DK Object during the first processing phase. This can only be accomplished after the first processing phase finished and all DK Objects were generated. Our solution is to store during the first processing phase either the id of the DK Object, if that exists already, or otherwise to store the name of the DK Object in a temporary attribute of the DK Operation, which can then be processed when the entire application ran once through the LDK Extractor.

Besides the attributes, we also need to apply the semantics referring to the routines enclosed between the `begin operation` and `end operation` directives. Similarly to processing the variables assigned to DK Entities, we need to deploy again the mechanism used for memorizing the node in the parse tree corresponding to the `begin operation` directive. Upon encountering an `end operation` event, we traverse the parse tree looking for routine declarations.

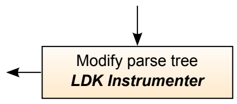
In the particular case of Fortran codes, one has to distinguish between several types of routine declarations. First of all, there are function declarations and subroutine declarations. Then, the routines can also be nested, with functions having, for example, other locally declared functions. And finally, there are also routines declared within modules and thus have a different semantic scope.

Our framework considers the following cases of declarations, with examples given in listing 14:

- function declarations;
- subroutines declarations;
- declarations of nested functions and subroutines; and
- declarations of module functions and subroutines.

All these different types of declarations result in slightly different parse sub-trees which have to be traversed accordingly. We use in this case the low-level functions provided by the `NAGf90` library.

6.1.3 *LDK Instrumenter*



The LDK Instrumenter is the component responsible for performing the modifications in the parse tree. This generally means adding extra nodes which define either routine calls or variable declarations.

The LDK Instrumenter is being used in the both processing phases of an application. In the first phase it is responsible for inserting the so called *event triggers*, as demanded by the LDK Directives Processor. In the second phase it runs in a more autonomous mode, enhancing the source code with support for *exposing dynamic information*, based on an extra `ldk_flags` argument which is added to routines.

6.1.3.1 *Event triggers*

In the first processing phase, the LDK Instrumenter is modifying the parse tree according to the information received from the LDK Directives Processor. There are two cases where such an action is required:

1. when entering a DK Phase; and
2. when a code region (inside a routine or program) is declared as a DK Operation.

In both cases, the LDK Instrumenter needs to insert a call to the corresponding function of the LDK Interface Library. This is an event trigger to announce that either a given DK Phase was entered, or that a given DK Operation either started or ended.

These events are exposed through the LDK Interface Library to the programming tools using the LDK Framework. These can register specialized callback functions to handle the specific events.

The modifications of the parse tree are accomplished in our framework by means of the two low-level functions provided by the NAGf90 library:

```fortran
PROGRAM exampleroutines
! program body
! ...
CONTAINS
        ! function declarations
        INTEGER FUNCTION myfunc()
                ! function body
                ! ...
        END FUNCTION

        ! SUBROUTINE declaration
        SUBROUTINE mysubroutine
                ! SUBROUTINE body
                ! ...
        END SUBROUTINE

        ! routine in another routine
        ! (function or subroutine)
        INTEGER FUNCTION parentfunc()
                ! function body
                ! ...

                CONTAINS
                        INTEGER FUNCTION childfunc()
                                ! function body
                                ! ...
                        END FUNCTION
        END FUNCTION

END

MODULE examplemodule
! module body
! ...
CONTAINS
        ! routines declarations in modules
        INTEGER FUNCTION mymodulefunc()
                ! function body
                ! ...
        END FUNCTION
END MODULE
```

Listing 14: Declarations of functions and subroutines in Fortran supported by our framework.

- `PT_addprev(pt_node, new_node)` - inserts in the parse tree the new node `new_node` as a sibling to the left of the already existing node `pt_node`; and
- `PT_addnext(pt_node, new_node)` - inserts in the parse tree the new node `new_node` as a sibling to the right of the already existing node `pt_node`.

6.1.3.2 *Expose dynamic information: `ldk_flags` array*

One great characteristic of the DK semantics is the fact that, based on *static* annotations it extracts *dynamic* information at runtime. Formulated from the domain level point of view, this means that data and actual values take precedence in front of the solution specific structural organization. Or, from an implementation oriented point of view, the DK objects have a global scope and lifetime.

For example, the code in listing 15 comprises two entity declarations, two subroutine calls on variables assigned to entities and other routine calls on formal arguments. Following the two possible approaches above, the dynamic characteristic is given by the following:

- The set of matrix processing operations are structurally enclosed into one subroutine, but the results obtained herein refer to a concrete entity. Which of the two entities exactly, is defined only at runtime.
- The definition of the two entities includes at the first level the local variables enclosed between the markups. Due to the global semantics of the entities though, all formal parameters within routines calls, which map to the initially assigned values, regardless of the nesting level of the call, are also semantically assigned to the entities. Execution branches and recursive calls make some of the formal parameters identifiable only at runtime.

Dynamic information is definitely a requirement for any DK enhanced framework. Moreover, it has to be provided in both directions: from the outer-most to the inner-most context and back from inside outwards. Outer-most context would be the entity declaration, while the inner-most context is the last level of nested subroutine calls. The solution which we provide in our framework uses the call stack as the transfer channel of the dynamic information, along with the global identification system of the DK Objects.

We automatically instrument routines by extending their signatures with an extra argument which is added at the end of the list of existing arguments. The extra argument is called `ldk_flags` and is an array with one integer entry per formal argument of the routine. `ldk_flags` stores at runtime the id of the entity to which an argument is assigned to, if such a mapping exists. Otherwise, it is initialized to `-1`. We thus have at runtime, within the body of a routine:

$$\mathtt{ldk\_flags}[i] = \begin{cases} j \in \mathbb{N}, & \text{if } i^{th} \text{ argument in the formal list of arguments is} \\ & \text{mapped to the entity with id } j; \\ -1, & \text{otherwise.} \end{cases}$$

```
PROGRAM dynamicsemantics
...
!$LDK \begin{entity}{A}
REAL, DIMENSION(:), ALLOCATABLE :: matrixA
!$LDK \end{entitiy}

!$LDK \begin{entity}{B}
REAL, DIMENSION(:), ALLOCATABLE :: matrixB
!$LDK \end{entitiy}

CALL process_matrix(matrixA)
CALL process_matrix(matrixB)

...
CONTAINS
        SUBROUTINE process_matrix(m)
                REAL, DIMENSION(:), ALLOCATABLE, INTENT(INOUT) :: m
                ...
                ! matrix operations
                m = ...

                ! data dependent execution branch
                IF (test_matrix(m)) THEN
                        CALL adjust_matrix(m)
                ENDIF

        END SUBROUTINE

        LOGICAL FUNCTION test_matrix(test_m)
                ! return value based on values in test_m
                test_matrix = ...
        END

        SUBROUTINE adjust_matrix(h)
                REAL, DIMENSION(:), ALLOCATABLE, INTENT(INOUT) :: h
                ! other operations
                h = ...
        END
END
```

Listing 15: Short example of entities declaration. The dynamic semantics is given through the subroutine calls.

The call statements of such routines have to be instrumented as well. In this case, the dynamic information, if it is provided in the enclosing routine, has to be passed further:

$$\mathtt{ldk\_flags}'[i] = \begin{cases} j \in \mathbb{N}, & \text{if } i^{th} \text{ argument in the call list is a variable} \\ & \text{mapped to the entity with id } j; \\ \mathtt{ldk\_flags}''[\mathtt{idx}[i]], & \text{if } i^{th} \text{ argument in the call list is the } \mathtt{idx}[i]\text{-th} \\ & \text{local argument of the enclosing routine and} \\ & \text{that routine is instrumented;} \\ -1, & \text{otherwise.} \end{cases}$$

In our framework, the instrumentation of the code is taking place while the Periscope instrumenter is processing for the second time the entire source code of the application, one file at a time. For each file, the parse tree is built and then it is traversed in a pre-order fashion.

The call to the LDK Instrumenter is integrated as shown in figures 17 and 18. The diagrams present the logical processes which are executed for the different types of nodes. Please note that we present them in separate diagrams only for a better comprehension, otherwise they are part of the same instrumentation process.

CALLS TO ROUTINES OR INTERFACES

The first diagram presents the automatic instrumentation of calls of routines and interfaces. The goal is to insert as a last argument of the call, the `ldk_flags` array, initialized for each of the arguments of the routine or interface with either:

- entity IDs;
- values of the local `ldk_flags`; or
- `-1`.

Before the actual modification of the parse tree, there are a couple of preparation steps, which also depend on the programming language of the source file. For our Fortran implementation, we first run four test cases:

1. **is internal**: if the routine is not *internal*, it means that it belongs to an external library or module, which is not compiled along with the current application. This means that the definition of the routine cannot be instrumented and, hence, the extra argument cannot be added to the signature of the routine. Instrumenting only the call of the routine would result in an incorrect code.

   In our framework, we generate during the first instrumentation phase, when the DK annotations are processed, a list with all internal routines of the application. This list is loaded in the second instrumentation phase and can be used to check whether a routine is internal or not.

2. **has arguments**: if the routine has no arguments, then there is no information to store in the `ldk_flags` array at all. Instrumentation is needed only if the routine has at least one argument.

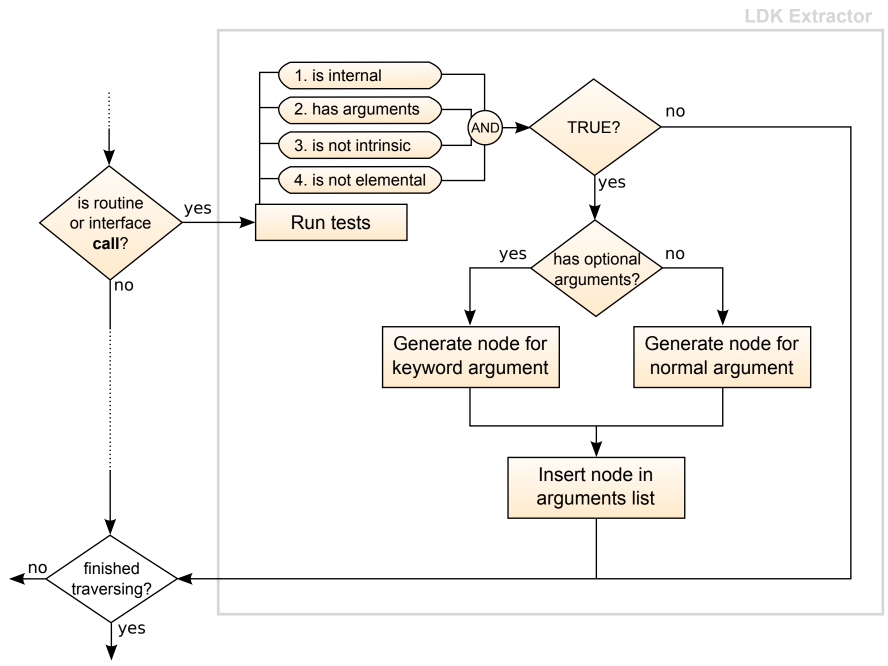


Figure 17: Automatic instrumentation of calls to routines and interfaces.

3. **is not intrinsic**: this is similar to the test for internal routines. Intrinsic routines cannot be instrumented.

4. **is not elemental**: elemental routines are a particular type of Fortran routines. They accept only one argument and are usually used as operators. Since it is not allowed to extend the list of arguments, these routines are also not instrumented.

If the current routine passes all tests, then the call is going to be instrumented.

As a general discussion, there might be cases where some call statements are not considered in the instrumentation process, or, if so, some other limitations might occur and the respective call remains uninstrumented. In this case, we could have the situation that for the same routine we have both instrumented and uninstrumented calls.

Fortran offers a great feature to support such behaviour: optional arguments. Given, for example, the code in listing 16, where `MyRoutine` has two optional arguments, one can use call statements like the first two in the code, omitting optional arguments from the call list.

For our purposes, it would be useful to have `ldk_flags` declared optional, as this allows to also have some uninstrumented calls of the routines. For these cases, the extra optional argument is omitted, i.e. not inserted in the call statement, leaving only the initial arguments, which is a valid call.

There are some few considerations which need to be made here for some particular situations. Listing 16 shows several possible call statements for the case where there are already some optional arguments declared for this routine.

While the calls 1-3 are correct, the 4th and 5th calls bring along some difficulties. In the call number 4 the optional argument `c` is left out from the call statement. This is in essence, the same as in call 2. If, however, this call needs to be instrumented,

```
PROGRAM hardinstrumentation

INTEGER :: x,y,z
...

! 1. uninstrumented call --> OK
CALL MyRoutine(x,y,z)

! 2. uninstrumented call omitting argument c --> OK
CALL MyRoutine(x,y)

! 3. instrumented call --> OK
CALL MyRoutine(x,y,z, (/-1, -1, -1/) )

! 4. instrumented call omitting argument c --> NOT OK
! * incorrect computations or runtime errors are
! * produced due to erroneously assigning the array
! * to the omitted argument c
CALL MyRoutine(x,y, (/-1, -1, -1/))

! 5. instrumented call omitting argument c --> NOT OK
! * compiler error: type mismatch
CALL MyInternalRoutine(x,y, (/-1, -1, -1/))

! 6. instrumented call omitting argument and providing keywords --> NOT OK
! * compiler error: keywords can only be used if the routine
! * has an explicit interface
CALL MyRoutine(x,y, ldk_flags=(/-1, -1, -1/))

! 7. instrumented call omitting argument and providing keywords --> OK
! * the compiler generates interfaces for internal routines
! * and thus keywords are accepted
CALL MyInternalRoutine(x,y, ldk_flags=(/-1, -1, -1/))

CONTAINS
        SUBROUTINE MyInternalRoutine(a, b, c, ldk_flags)
                INTEGER :: a, b
                INTEGER, OPTIONAL :: c
                INTEGER, DIM(3), OPTIONAL :: ldk_flags
                ...
        END SUBROUTINE
END

SUBROUTINE MyRoutine(a, b, c, ldk_flags)
        INTEGER :: a, b
        INTEGER, OPTIONAL :: c
        INTEGER, DIM(3), OPTIONAL :: ldk_flags
        ...
END
```

Listing 16: Examples of call statements for routines which have optional initial arguments and could be instrumented or not with the optional extra argument `ldk_flags`.

then the array constructor is added at the end of the call for the special `ldk_flags` argument. In the given example, the code will compile and the first element in the array will erroneously be attributed to argument `c`. If the routine is an internal or module routine, i. e.declared within `CONTAINS` or module file, then the compiler is able to report a type mismatch between the actual and the formal arguments. This can be seen in the listing in call 5 for the internal routine `MyInternalRoutine`

The solution is to add a keyword to correctly identify the optional argument. Call number 4 is then adjusted like in call number 6, but this produces again a compiler error, since keywords can only be used if there is an explicit interface defined for the routine.

A correct example is given in call 7. Although this looks like a duplicate of call 6, there is one important difference: the `MyInternalRoutine` is an internal routine. Compilers automatically generate interfaces for internal and module routines and thus keywords can be used in this case.

To summaries, one can always declare the `ldk_flags` argument as optional, but using keywords is only possible if one of the three cases apply:

- the routine has an explicit interface defined in the code, e. g.:

```
INTERFACE
        SUBROUTINE MyRoutine(a, b, c, ldk_flags)
                INTEGER :: a, b
                INTEGER, OPTIONAL :: c
                INTEGER, DIM(3), OPTIONAL :: ldk_flags
        END
END
```

- the routine is an internal routine, defined in the `CONTAINS` section; or

- the routine is a module routine, defined inside a module.

The goal is to cover with the instrumentation strategy as many cases as possible. If the routine is internal or module routine, then using keywords is always allowed, but only mandatory for routines which already have optional arguments. If, instead, the routine is an external routine, then we can differentiate between routines with and without optional arguments. For those without optional arguments, it is not necessary to use keywords at all and thus the calls will always be valid, no matter whether the routine has an explicit interface or not. For the routines which do have optional arguments, the keywords need to be used to insure valid calls. It is common that such routines also have an interface declared, since keywords might already be used in other calls. If this is the case, then the instrumented call with keyword arguments will also be valid. If the routine does not have an explicit interface, though, then the instrumentation will not be successful. A quick solution would be to just add the respective interface in the source code.

We consider that this strategy covers the most cases met in real world codes.

Fortran syntax allows the initialization of arrays with array constructors, as seen in the calls above. For other programming languages it might be necessary to insert some more nodes in the parse tree for declaring and initializing the `ldk_flags` variable and then including it in the call statement.

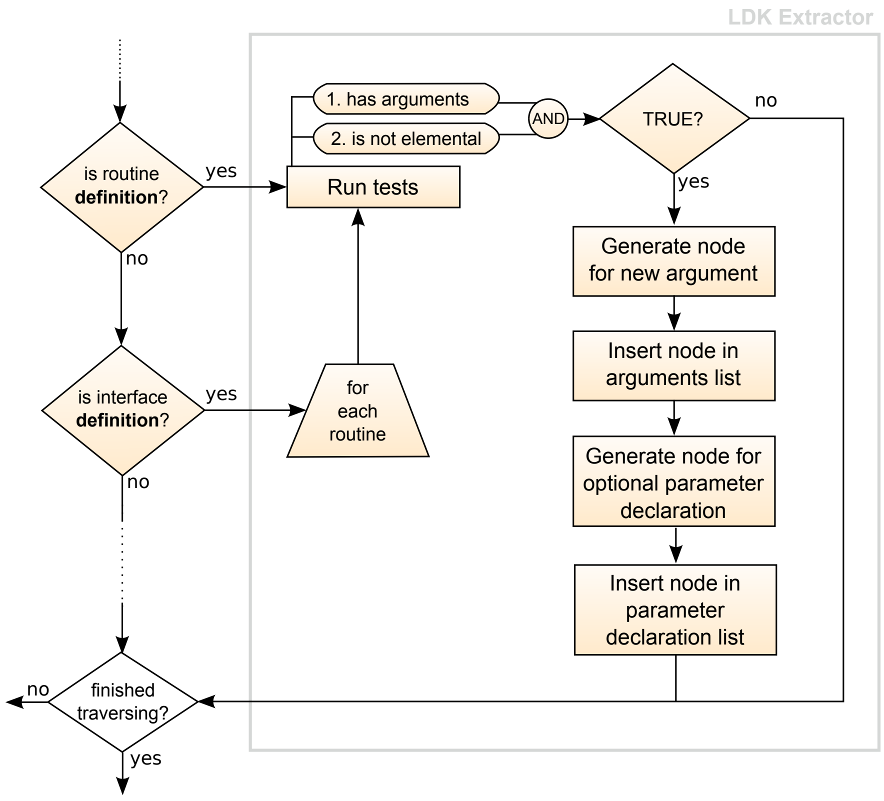


Figure 18: Automatic instrumentation of definitions of routines and interfaces.

When building the list of values for the extra argument, there are several aspects which have to be taken into consideration.

First of all, the call arguments list may contain not only variables, but also constants, expressions, or function calls. In DK terms, though, only the variables, including array elements, can be attributed to a DK Entity.

Secondly, the order of the actual arguments might be different from the order in the formal list of arguments of the routine and some arguments might be missing from the call statement altogether. The order is important because in the `ldk_flags` array, the elements are associated with the arguments based on their index. It is thus important to make sure that the `ldk_flags` array is built based on the indexes in the formal arguments list.

Lastly, some of the arguments in the call list might be local arguments of the enclosing routine. In this case, if the enclosing routine is also instrumented, i. e.it contains an `ldk_flags` array in its signature, then the corresponding index of this local `ldk_flags` array has to be inserted in the list of values given to the call statement.

DEFINITIONS OF ROUTINES OR INTERFACES

The other type of parse tree nodes which are subject to automatic instrumentation are the definition nodes, both routines and interfaces definitions. The goal of the instrumentation is to extend the signature of the routines with the extra argument `ldk_flags`.

The logical sequence is presented in figure 18. The steps are similar to those in the instrumentation of the call nodes with some few differences.

First of all, there is no need to test whether the routine is intrinsic or if it does not have a definition in the source code. The fact that the current node contains the definition of the routine excludes these possibilities.

The next step is the insertion of the argument in the list of formal arguments. Besides generating the corresponding node for the parse tree, the `ldk_flags` array must also be registered in the symbol table. In order to do so, we again use in our framework the low-level functions provided by the NAGf90 library. We declare the string to be used in the name of the symbol, then create the symbol and set the type and kind and the flags for formal optional argument and `IN` intent. In the end, a node for this argument is generated and inserted in the parse tree at the end of the formal arguments list.

For Fortran implementations, in addition to the insertion in the arguments list, the extra argument must also be declared along with all other parameters and variables in the body of the routine. We thus generate a node for the argument declaration as well. An important detail is that the `ldk_flags` array has a fixed length. The declaration of the array thus contains besides the `OPTIONAL` property explained above, also a `DIMENSION(<n>)`, with `n` being the number of formal arguments of the routine.

6.2 LDK OBJECTS

The LDK Objects component is the *database* storing the actual instances of the DK Objects generated for the current application. The term *database* refers here only to the functionality provided by such a component, as we do not deploy any DB specific technology, such as MySQL or DB2. Our implementation is based on a set of advanced pure C data structures.

For the data structures we rely on the functionalities provided by the *uthash* library. Uthash offers a macros-based implementation of some of the common data structures used in programming, but which do not have built-in support in C. Such are hashes, arrays, lists, and strings.

We use hashes to store the instantiations of three DK Concepts: *DK Entities*, *DK Operations* and *DK Phases*. In addition to these three type of elements, we also store in the LDK Objects database two more types of elements: *Var*s and *Func*s. These are the counterpart of the variables and routines which are found in the source code and which are being assigned to one of the DK Objects. They are thus no DK Objects by themselves, but are stored in the database for an easier management.

Figure 19 gives an overview of the hashes used in the LDK Objects component.

An element in a `UT_hash` is a C structure. We defined in total five structures: three for the DK Objects - entities, operations and phases -, and two more for the variables and routines which are assigned to entities and operations, respectively.

The *DKEntity*, *DKOperation*, and *DKPhase* structures have in common the `id`, `name`, and `type` fields:

- `id`: a positive integer value which uniquely identifies the corresponding DK Object within the set of objects of the same kind; e.g. the `id` of an entity is unique in the set of `id`s of all declared entities;

- `name`: a string containing the name of the object. We make use here of a name mangling/decoration strategy and prepend the initial name in the case of enti-

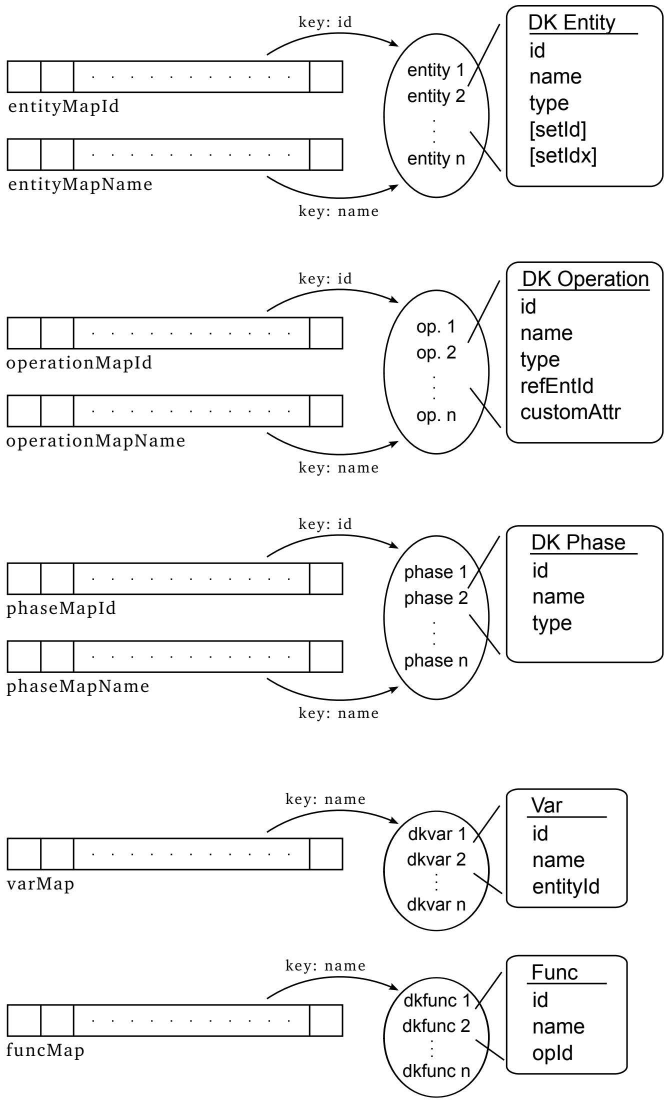


Figure 19: Data structures used to store the DK Objects. The *Map* denotes a `uthash` implementation, while the element boxes on the right are of type `struct` with corresponding fields.

ties and operations with a sequence of dot-delimited ids. The sequence of ids represents the *namespace* of the current object. Entities and operations can be nested and the namespace contains the IDs of all parent objects which enclose the current object.

For example, given the declaration of entities in listing 17, the `id` and `name` fields will be initialized as follows:

```
A.id = 0
A.name = ".A"
B.id = 1
B.name = "0.B"
C.id = 2
C.name = "0.1.C"
```

```
        \begin{entity}{A}
                INTEGER :: varA
                \begin{entity}{B}
                    INTEGER :: varB
                        \begin{entity}{C}
                                INTEGER :: varC
                        \end{entity}
                \end{entity}
        \end{entity}
```

Listing 17: Declaration of three nested entities.

Since DK Phases are not block constructs, it does not make sense to discuss about nesting of phases. In this case, the `name` field stores the name of the phase, which is also global.

- `type`: an integer value representing the index of the predefined type .

The DKEnity structure requires two more fields used for implementing the *entityset* concept:

- `setId`: the ID of the entityset to which this entity belong to; and
- `setOffset`: the offset of this entity within the entityset.

The DKOperation structure has two specific fields, as well:

- `linkedEntId`: a positive integer value representing the ID of the entity to which this operation is linked to, if any. This is set by means of the `linkedentity` attribute explained in section 4.3; and
- `customAttr`: a `UT_hash` containing the name/value pairs of all other defined attributes.

The *Var* and *Func* structures have each three fields:

- `id`: a positive integer value representing the ID of the object;
- `name`: a string value containing the name of this object. This includes the full identification of the scope where the variable or routine is declared, thus providing a global unique name for each object; and
- `entityId` or `opId`: the entity or operation to which the variable or, respectively, the routine is assigned to.

The syntax of the DK annotations include the name of the respective DK Object as a mandatory string parameter. This is the human-readable identifier of the respective DK Object. On the other hand, the management system and, much more, the common tracking systems work with number identifiers. This is common practice due to the more efficient indexing and handling operations for numeric identifiers, as well as the less memory required for storing the ids.

We thus need to index our sets of DK Objects both, on their `ids`, and on their `names`. This is accomplished within the LDK Objects component by means of pairs of hashes:

one hash per index type and one pair of hashes per DK Object type. In figure 19, for example, the `entityMapId` is a hash of DKEntities using as hash key the id field, and the `entityMapName` is another hash of DKEntities using as hash key the name field.

Every two hashes in a pair point to the same set of DK Objects. This is indeed the case in our implementation. The `UT_hash` data structure only stores the references to the actual element structures. When creating a new DKEntity, for example, we allocate once the necessary memory for storing the fields in the structure and then we feed the reference to this new DKEntity to both the `entityMapId` and `entityMapName` hashes. Thus, the maps provide two different access indexes for the same set of objects.

For the variables and routines we only need the name hashes, since these structures are not exposed further to the programming tools using the LDK Framework. Internally, throughout the execution of the LDK Extractor components, we only need to access these elements based on their names.

6.3 LDK INTERFACE LIBRARY

The LDK Interface Library provides support for query operations and management of the objects in LDK Objects. It is a stand-alone C implementation, which, together with the LDK Objects component, is designed to interact with both the LDK Extractor, as well as the advanced programming tools which will use the DK information during the program execution.

There are four types of functions:

1. functions which declare new objects and add them to the database;
2. functions which search objects in the database;
3. functions which store and load the database entries; and
4. utility functions for querying objects properties.

The first type of functions implement the same algorithm for each of the DK Objects. They are called either by the LDK Directives Processor, when the DK annotations are processed, or by the advanced programming tools which use the DK information, when they initialize the database or when dynamic objects are created. They receive as a mandatory argument the name of the DK Object. An optional argument is the `id` of the object: if no id is given, then a new object with a new id is created. Otherwise, the new created object is assigned the given id. This is useful for the case when the objects are loaded from a dump file, for example, as compared to the objects generated at instrumentation or runtime.

When a declare function is called, the first step is to check whether there already exists an object of the given type with the given name. If this is not the case, then this is the declaration of a new object. The necessary memory is allocated and the eventually provided attributes are initialized. If an `id` is given as an argument, then it is assigned to the object, otherwise a new unique id is generated for this type of object. In the end, the object is pushed to the corresponding hash structures and the reference to the object is returned by the function.

If there already exists an object with the given name and type, then according to the DK semantics, this means either that:

- an entity is being extended with new variable declarations;
- an operation is being extended with new routine declarations; or
- a phase is begin re-entered.

Thus, for these objects there are no further actions to be taken at this point.

On the other hand, if it is the declaration of a Var or Func structure and such a structure with the same name already exists, then an error is generated. It would mean, for example, that the same variable is going to be registered for two different DK Entities, which is not allowed.

As an implementation detail, there is one particular case which is treated separately, namely the declaration of variables in subentities. In section 6.1.2 we showed that the variable declarations are collected only at the `end entity` directive. This means that, given a declaration of nested entities as in listing 17, the declaration of the variable `varC` will be identified and registered to entity `C` when the first `\end{entity}` directive is encountered. The next `\end{entity}` is closing the declaration of the entity `B`. The entire parse tree between the beginning and ending of entity B is searched now for variable declarations. First of all, variable `varB` is found and added to the database. Then the declaration of the `varC` variable is found again. The call to `declareDKVar()` for this variable will find it already declared in the database and assigned to entity `C`. This is obviously not a semantic error, but an implementation issue. Hence, we do not directly throw an error when finding a variable with the same name already in the database. First we check instead whether the entity, to which the variable is assigned to, is a subentity of the currently processed entity. In the given example, we check whether entity `C` is a subentity of entity `B`.

Checking such a relation between entities is an example of the utility functions within the LDK Interface, mentioned in the enumeration above at point 4. Theoretically, the function should simply test whether the `id` of the parent entity is found in the namespace definition of the subentity. As explained in section 6.2, the names of entities are decorated with the dot separated list of `ids` of their parent entities, in their nesting order. Since these are strings of chars, it is more efficient for our C implementation to actually compare entire strings. We first extend the namespace of the parent with the id of the parent and then issue a `strncmp()` between the extended namespace of the parent entity and the namespace of the subentity, but only for the length of the first one.

```
if (strncmp(extendedParentNS, subentityNS, extendedParentNS_length ) == 0){
        return 1; // it is subentity
}
```

The second type of functions in LDK Interface are the search functions. Their implementation is based on the interface provided by the `UThash` library. For example, looking up an entity in the name-keyed hash is accomplished with a call to `HASH_FIND_STR()`, like in listing 18.

The standard complexities for the operations on hashes apply to the UThash implementations as well: searching, adding and deleting objects from the hashes are in average constant-time operations, *O(1)*.

DKEntities, DKOperations and DKPhases, can be looked up either by their name or their id, while variables (Var) and routines (Func) are indexed only after their names.

```
DKEntity* findEntityByName( char *entityName ){
        DKEntity* currEntity = NULL;
        HASH_FIND_STR( entityMapName, entityName, currEntity);
        return currEntity;
}
```

Listing 18: Looking up an entity by its name. Implementation based on the *UThash* functions.

As explained above, the names stored for entities and operations are already decorated with the corresponding namespace. This assures both the uniqueness of the name identifier, as well as the representation of the parent-child relations. Nevertheless, there are also cases where the *short name* of an entity or operation is required. For example, when the `linkedentity` attribute of operations is used:
`\begin{operation}{MyOperation}[linkedentity=MyEntity]`
`linkedentity` specifies to which entity the current operation applies or refers to. The DK annotation will contain only the *short name* of that entity, namely the one used when declaring it with `\begin{entity}{MyEntity}`. Such a search operation reduces at traversing the entities hash and checking whether the given *short name* matches the last part of the name of the current entity. In this particular case, searching is of linear complexity, *O(n)*, with *n* being the total number of entities.

The third type of functions in LDK Interface are the functions for storing and loading the database entries. As expected, this set of functions is not the result of some DK semantics, but it is rather a particular implementation aspect of our framework. Storing and re-loading of the DK objects is required in two cases:

1. when advancing with the instrumentation process from one source file to the other; and

2. at runtime, when the database is used by the advanced programming tools using the LDK Framework.

The first case appears due to the fact that the DK Objects have global declarations, but the Periscope instrumenter only instruments one file at a time. Thus we use the store and load function from the LDK Interface to dump all entries from the LDK Objects when the instrumentation of a file is finished and then to re-load them when starting the instrumentation for the next file.

The second case will be covered in more detail in the next chapter, when presenting the DK-enhanced performance analysis framework.

The store functions simply dump all entries from the object hashes of LDK Objects into an ASCII file. Each object is written on a line, starting with the object type, then the name and the id, and then the rest of the fields, as defined in the corresponding C structure. For example, in listing 19, there are two entities stored: *integrals* and *uniqueatoms*, respectively. The excerpt also shows three variables registered to the first entity and other two variables registered to the second entity. The field after the id of a variable gives the id of the entity to which this variable is registered to. In the end there are also two phases defined, *initmaster* and *computemaster*, with `ids` 1 and, respectively, 2, but without a given phase type - hence `0` in the last field.

```
ENTITY .integrals 1 -1 -1
ENTITY .uniqueatoms 2 -1 -1
...
VAR INTEGRALSTORE_MODULE.INTEGRALSTORE_3C_FIELD 1 1
VAR INTEGRALSTORE_MODULE.INTEGRALSTORE_3C_POTEN 2 1
VAR INTEGRALSTORE_MODULE.INTEGRALSTORE_3C_CO 3 1
...
VAR UNIQUE_ATOM_MODULE.MOVING_UNIQUE_ATOM_INDEX 15 2
VAR UNIQUE_ATOM_MODULE.UNIQUE_ATOM_GRAD_INFO 16 2
...
PHASE initmaster 1 0
PHASE computemaster 2 0
...
END
```

Listing 19: Excerpt from an ASCII file storing DK objects.

6.4 DK CUSTOM ATTRIBUTES

DK Custom attributes offer the flexibility necessary for adapting to different application domains. The attributes are to be stored within the DK Objects. Any particular instrumentation strategy can be added as an extension to the *LDK Instrumenter*. Further instrumentation strategies may also be needed in the advanced programming tools using the DK Objects database.

7 A FRAMEWORK FOR DK-ENHANCED PERFORMANCE ANALYSIS

In chapter 2 we already described the common process of performance measurement and analysis. We saw that all existing tools generally follow the same steps in the performance analysis flow, presented in figure 2 as well. In order to prove the results which can be achieved when applying the Domain Knowledge (DK) approach, we developed a prototype framework for performance analysis, which complies with the mentioned workflow.

In this chapter we present our DK-enhanced profiling framework. We start by giving a detailed overview of the performance analysis flow. We focus at each step on distinguishing between the actions which are required from users and those which are accomplished by tools, and we show, as well, which are the outputs of these actions, mostly different types of files. We also include in each step examples from the main performance tools: Periscope, Scalasca and Vampir, which were briefly introduced in section 2.4.

We then go on and present first an overview of our framework with its main components, their interconnections and main functionalities. More insights are then provided in the following sections, which are structured based on the steps of the performance flow: instrumentation, measurement and analysis.

The main goal is to show how we used the DK in each of these steps and, in particular, how we integrated and used the LDK Framework to build the new DK-enhanced profiling framework.

The chapter concludes with the potential extensions of the framework. We consider here possibilities for extending the supported functionalities, for widening the support to other technologies, as well as for improving the framework usability.

In the remainder of this chapter we are going to use interchangeably the terms *performance measurement and analysis* and *profiling*. Based on the classification in section 2.1, our framework is in fact a *tracing tool*, but we keep the convention above for the sake of simplicity.

7.1 THE DETAILED PERFORMANCE ANALYSIS FLOW

Before delving into the implementation insights of the DK-enhanced framework, a more detailed view of the performance analysis flow is required. We iterate here over the representation of the performance analysis flow from section 2.1, figure 2 and extend that flow diagram into an activity diagram like in figure 20. Our goal is to provide an understanding of the performance analysis process as seen from the point of view of the application *developers*, which now become profiling tools *users*. In the next sections, we will also add the point of view of the *profiling tool developers*, when we consider the design and implementation details of the components of our DK-enhanced profiling framework.

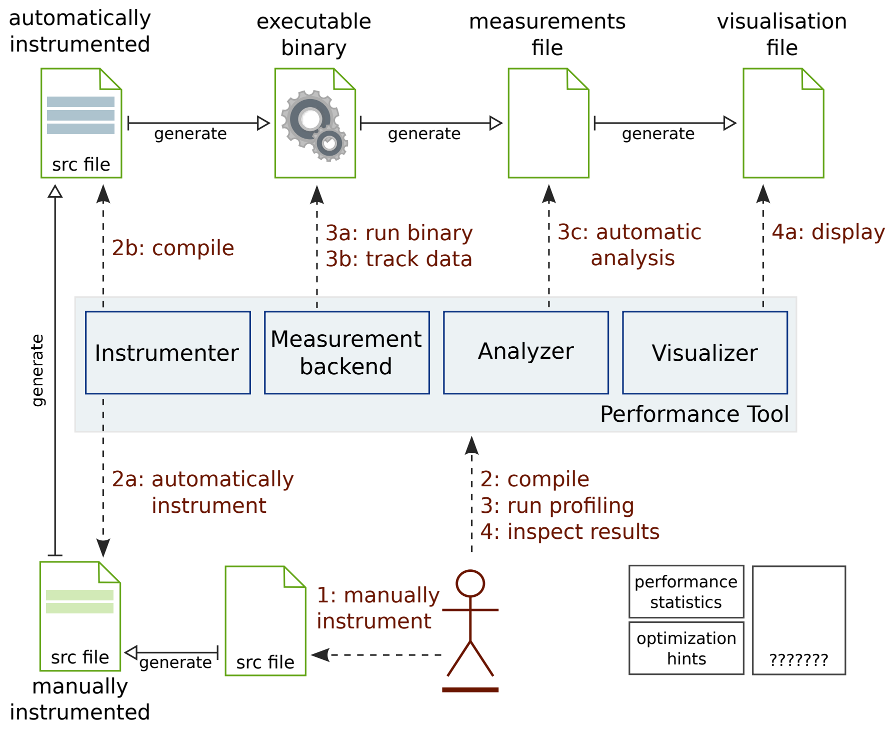


Figure 20: The detailed steps of the performance measurement and analysis flow.

In the performance tuning process, given is the source code of an application and the ultimate goal is to obtain a new version of the application, necessarily with improved performance. In order to achieve the final goal, two intermediate results are produced: characterization of the application performance and optimization hints.

There is one main *actor* in this process: the developer of the application, depicted in figure 20 in the lower center. To some extent, the performance tool could also be considered an actor for the steps of the process where automatic actions take place, like *3b: track data* and *3c: automatic analysis*.

1: manually instrument
2a: automatically instrument

The flow is started with the instrumentation of the application. As described in chapter 2, both *manual* and *automatic instrumentation* are possible. If present, the manual instrumentation has to be performed first. This is an action applied by the user directly to the source code of the application, see action marked with 1 in the diagram. The automatic instrumentation is usually performed along with the compilation of the application - actions 2 and 2a.

Most tools have defined a command to be prepended to the call of the compiler itself. For example, the common way of automatically instrumenting applications which are built using a `Makefile` is to declare an instrumentation variable depending on the tool which is being used:

```
# Instrumenter
# Periscope
INST = psc_instrument
# Scalasca
INST = scalasca -instrument
# Score-P
INST = scorep
# VampirTrace
```

```
INST = vtcc -vt:cc
```

and then prepend the common declaration of the compiler with the previously defined instrumenter call:

```
# C Compiler
CC = gcc -O3
```

```
# C Compiler, prepended with instrumentation call
CC = $(INST) gcc -O3
```

The automatically instrumented source file generated at this step is then passed to the compiler itself. This is action 2b in the diagram and has as a result the executable binary of the application. At this point, the necessary libraries for the performance measurements get linked as well. It is common practice that the performance tools have dedicated libraries which are used in the measurements step.

2b: compile

The actual performance analysis starts when the user issues the commands for running and profiling the application. This is action *3: run profiling* in our diagram, which divides in three sub-actions on the tool side: *3a: run binary*, *3b: track data* and *3c: automatic analysis*.

First of all, starting the application has its own particularities, depending on the specific tool which is used. Thus, action 3a in the diagram could assume:

3a: run binary

- starting the performance tool and providing the usual command line call of the application as an argument for the tool, along with other settings. This is the case for Periscope and Scalasca:

  ```
  $ psc_frontend --apprun="<CMDLINE>" --mpinumprocs=16 --force-localhost
  ```

  ```
  $ scalasca -analyze  mpiexec -n 16 <CMDLINE>
  ```

- starting the application normally, with the performance tool running in the background, like with Score-P and VampirTrace.

In both situations the user has in addition the opportunity to use a wide range of predefined environment variables to configure and control the execution of each particular performance tool.

The next step is collecting the measurements data, action 3b in the diagram. Which type of data is being collected and how it is stored or processed, this is again very specific to each performance tool. As already described in section 2.4.5, Score-P was a joint effort to provide a standardization for all these different strategies and, as a result, the Score-P framework for instrumentation and measurements can be used in combination with tools like Scalasca, Vampir, Periscope and TAU. Regardless of the specifics of each measurement backend or library, this step of the performance profiling is completely transparent to the user.

3b: track data

The next step depicted in the diagram, *3c: automatic analysis*, has again some particularities. We used a very simplified representation of this step, in order to underline the concept of using automatic analysis of measured data and to roughly localize it in the overall performance flow. In practice, there are different approaches to automatic analysis, based on the main orientation of each of the performance tools.

3c: automatic analysis

Periscope, for example, uses *online analysis*, which means that the measurements are processed at runtime and, based on the results of this analysis, new measurements are scheduled. Thus, for Periscope, step 3c is combined with 3b and the analysis is not applied to some generated measurements file, but directly to the measurements available at given points in the execution of the tool. The analysis is also transparent to the user.

In contrast, Scalasca provides a specialized tool for post-mortem processing of measurement files. This can be applied by the user to retrieve performance statistics and, eventually, to re-run the measurements after configuring the tool to target the hotspots which were identified, or to filter out the less interesting application parts. In this case, the automatic analysis step is already part of the next action represented in our diagram, *4: inspect results*, and the output is represented by one of the boxes on the lower right of the diagram, namely the *performance statistics*.

4: inspect results

The last action in the performance flow is the inspection of the performance results, or the *manual analysis* of the performance characteristics. This step always takes place after the measurements process finished and it is therefore also being called the *post-mortem analysis*.

There are mainly two ways of performing the manual analysis: either at command-line or by means of a GUI. In the first case, the user mostly uses some scripts or small tools or functionalities provided by the performance tool in order to generate appropriate statistics and profile numbers from the gathered measurements. This is basically, as already mentioned above, a mix between manual and automatic analysis.

In most cases though, the measurements are going to be inspected by means of a GUI provided by the performance tool. VampirTrace measurements are usually visualised with Vampir, Scalasca measurements with Cube, and Periscope measurements with the Eclipse-based GUI. Besides the GUI visualisation, all tools also provide command line methods to display information about the measured performance.

This last step concludes the performance measurement and analysis flow. In the lower right corner of the diagram we represented the actual response which is expected by users at the end of this flow:

- performance statistics;
- optimization hints; and, as a final goal,
- good performance of the application.

The *performance statistics* can usually be retrieved either from the automatic analysis or by generating them from the provided GUIs and command-line scripts.

For *optimization hints* and *good performance* though, it is usually up to the experience and skills of the user to harvest them from the provided measurements. This is where DK also comes into play. It helps users instruct the tools what is that they need to measure and then it supports the inspection of the results by embedding users' experience from their own research domains into the performance metrics.

There is also some effort towards automatic optimization of applications. One approach is to specialize on some particular implementation aspects, like, for example, MPI communication or execution loops. The tools implementing this approach usually combine static with dynamic analysis of applications. One could also call them

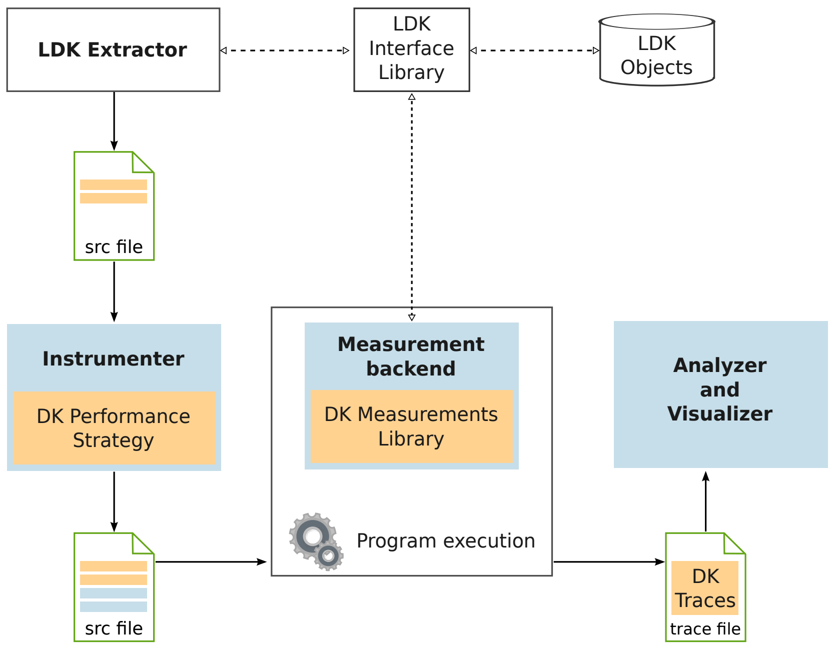


Figure 21: The main components of an DK-enhanced framework for performance measurement and analysis.

as *semi-automatic optimization* tools, as they only analyse the application and provide optimization hints, but do not actually modify the source code.

The other approach is also defined as *tuning* and it targets the improvement of performance by adjusting the values of different environment parameters or application internal variables. Such are, for example, the number of MPI processes used to start the application, the frequency of the CPU cores running the application threads or processes, or the specific compiler flags used to build the executable of the application. The main strategy is to define a set of possible configurations and to run the application with these different configurations, providing at the end the best one as a tuning solution. Thus, the resulting optimization hints do not refer to changes to the source code itself, but only to the values to be used for particular parameters.

7.2 FRAMEWORK OVERVIEW

In the overall structure of our DK-enhanced profiling framework we can distinguish two main implementation layers. On the one side, there is the LDK Framework, which we already described in chapter 6. Represented in the upper part in figure 21, the LDK Framework is responsible for generating the LDK Objects database and for preparing or instrumenting the initial source code with support for DK specific dynamic information.

The second implementation layer represents the components and extensions responsible for performing the performance measurement and analysis. The blue boxes in the lower range of the same figure represent the initial components of the profiling tool. These components can perform a *pure* performance analysis, without any DK considerations. From a general point of view, these components could be any components belonging to a performance analysis tool. We use here the generally agreed terms to define them as *Instrumenter*, *Measurement backend*, *Analyzer* and *Visualizer*.

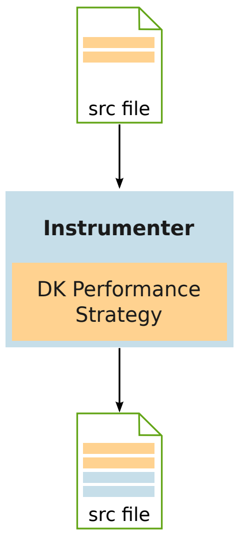
In our particular implementation we use the following:

- **Instrumenter:** the *f90inst* Fortran instrumenter from Periscope;
- **Measurement backend:** the *OTF2* tracing library from Score-P; and
- **Analyzer and Visualizer:** the *Vampir* GUI from the tool with the same name.

The orange boxes depicted inside the blue boxes represent the extensions which we implemented in order to enable the support for Domain Knowledge. We integrated in the *f90inst* instrumenter our own automatic instrumentation strategy, which we call *DK Performance Strategy*. As measurement backend, we developed our own library based on the *OTF2* for tracing support. There was no need to extend the *Vampir* GUI, but we designed particular *DK Traces*.

The *DK Measurements Library* processes at runtime the function calls inserted previously by the DK Performance Strategy component. It stores the results in the DK Traces, using the functionalities provided by the Measurement backend. The library also acquires the necessary information via the LDK Interface Library in order to store within the DK Traces, besides the values of the measured metrics, also the description of the DK Objects. The Visualizer relies on these descriptions to properly display the data. In this way we eliminate any direct dependency between the Visualizer and the LDK Framework.

7.3 INSTRUMENTATION



The main role of the instrumenter in the performance analysis flow is that of preparing the application by inserting appropriate calls to the measurement backend. We use in our framework the *f90inst* Fortran source code instrumenter of the Periscope online analysis tool. We extended *f90inst* with a *DK Performance Strategy* component, which automatically inserts appropriate code instrumentation in order enable capture performance information relevant for DK.

For our profiling framework we implemented two instrumentation strategies for two of the DK metrics introduced in section 4.4:

- Memory per Entity (MPE) metric; and
- Iterometers.

7.3.1 *MPE metric instrumentation support*

In section 4.4 we defined MPE as the amount of memory used by an entity throughout the execution of an application. In this section we show which are the instrumentation strategies applied in our framework for supporting the deployment of MPE analysis.

There are basically three questions with respect to MPE, which need to be answered in order to design the instrumentation:

A. **what** is the object of the metric?

B. **when** is the metric applied?

C. **how** is the metric evaluated?

The answers are given in the following paragraphs, along with the implementation specific details.

A. Dynamic arrays

A DK Entity can have any kind of variables assigned to it. All these variables occupy some memory attributed in the end to the entity. From all variables, though, it is natural that the most amount of memory is occupied by arrays. We use here the Fortran term *array* to refer to all data types which, in general, represent a set of elements: lists, maps, vectors, matrices, etc.

In most common programming languages there are two concepts with respect to memory allocation for arrays: static memory allocation and dynamic memory allocation. In static memory allocation the array is given a fixed size already at compile time, while in dynamic memory allocation the size is decided during the execution of the application.

Theoretically, both types of arrays could occupy a considerable amount of memory, but it is common practice nowadays that the larger arrays are dynamically allocated. This is due to the fact that these arrays mostly contain data which is read or computed at runtime and thus the exact size is only available during the execution.

Therefore we set the focus in our framework on dynamic arrays, but the implementation could be extended at any time to also consider the size of static arrays.

In Fortran, there are two types of dynamic arrays:

- allocatable arrays; and
- automatic arrays.

We do not consider pointers here, although they are classified sometimes as dynamic arrays as well.

Allocatable arrays are dynamic arrays for which memory can be allocated and deallocated at runtime by using the `ALLOCATE` and `DEALLOCATE` statements.

```
REAL, DIMENSION(:), ALLOCATABLE :: A
INTEGER :: n
...
ALLOCATE(A(n))
...
DEALLOCATE(A)
```

Automatic arrays are local arrays in routines which sizes are declared based on the value of some formal argument. The memory of such arrays is allocated automatically upon entering the routine and is deallocated automatically upon leaving it.

```
SUBROUTINE myRoutine(n)
        INTEGER, INTENT(IN) :: n
        REAL, DIMENSION(n) :: A
        ...
END
```

In our framework we demonstrate how both kinds of dynamic arrays can be considered in the MPE metric.

B. Code- and user-driven instrumentation

We distinguish two types of instrumentation strategies for MPE, based on the trigger which enforces the instrumentation: code- and user-driven instrumentation.

By *code-driven instrumentation* we refer to instrumentation actions which are enforced by particular statements or structure of the source code. In the case of MPE, these statements are the `ALLOCATE` and `DEALLOCATE` statements.

We extended the instrumentation process of the Periscope instrumenter to also integrate this strategy. The connection point is the same as in the case of the LDK Instrumenter, presented in section 6.1.3: during the second phase of the instrumentation, a new test condition is introduced to check whether the current node is corresponding to an ALLOCATE or DEALLOCATE statement. If this is the case, then each of the variables enclosed in the statement are checked whether they are assigned to any DK entity. If so, then a call to the `ldk_alter_mpe()` function of the DK Measurements Library is inserted in the parse tree for that specific variable. The function receives as arguments the id of the entity to which the variable is assigned to and the amount of memory which should be registered.

The other strategy type is *user-driven instrumentation*. Although it might sound like *manual* instrumentation, it is not the case. This instrumentation strategy is still automatic instrumentation, but it is carried out on the basis of the hints and information specified by the user.

More precisely, in manual instrumentation, the user edits the source and inserts either markups or even direct routine calls. For DK, the action through which the user inserts the DK annotations into the code is an example of manual instrumentation.

In user-driven instrumentation, the tool interprets the markups inserted manually by the user and then it automatically instruments the code, based on the information provided by these markups.

For MPE in our framework, this strategy is used to instrument the dynamic automatic arrays. We already mentioned that these arrays are local variables of routines. If the user declares such a routine as a DK Operation and it also specifies that this DK Operation is linked to one specific entity, then we can account the memory used by the automatic arrays to that particular entity.

For example, in listing 20, the routine `MyRoutine` is assigned to the operation `MyOperation` which is linked to the entity `MyEntity`. The local arrays `A` and `B` are not assigned to `MyEntity`, which contains only the variable `myArray`. Nevertheless, since all computations in `MyRoutine` are semantically linked to `MyEntity`, then it follows that the memory allocated for this computations should also be accounted for in the MPE of `MyEntity`.

Since we are using the same instrumenter component for both the DK-enhanced profiling framework, as well as for the LDK Framework, in our implementation, the instrumentation actions described in this section are carried out along with the `ldk_flags` instrumentation for the given routine. The diagram in figure 18 is thus extended with a second branch containing two steps: one for inserting the calls to `ldk_alter_mpe()` at the beginning of the routine and another one for inserting the matching calls at the end of the routine.

```
...
\begin{entity}{MyEntity}
REAL, DIMENSION(:,:), ALLOCATABLE :: myArray
\end{entity}

INTEGER :: m, k


ALLOCATE(myArray(m))
CALL ldk_alter_mpe(0, size(myArray), storage_size(myArray)/8) ! <--- inserted
    by instrumenter

CALL MyRoutine(k)
...
CALL ldk_alter_mpe(0, -size(myArray), storage_size(myArray)/8) ! <--- inserted
    by instrumenter
DEALLOCATE(myArray)

\begin{operation}{MyOperation}[linkedentity=MyEntity]
SUBROUTINE MyRoutine(n)
        INTEGER, INTENT(IN) :: n
        REAL, DIMENSION(n) :: A, B

        CALL ldk_alter_mpe(0, size(A), storage_size(A)/8) ! <--- inserted by
            instrumenter
        CALL ldk_alter_mpe(0, size(B), storage_size(B)/8) ! <--- inserted by
            instrumenter
        ...
        CALL ldk_alter_mpe(0, -size(A), storage_size(A)/8) ! <--- inserted by
            instrumenter
        CALL ldk_alter_mpe(0, -size(B), storage_size(B)/8) ! <--- inserted by
            instrumenter
END
\end{operation}
```

Listing 20: Instrumentation for MPE support.

C. Runtime size of arrays

The values of the MPE metric represent an amount of memory expressed in bytes. Determining the actual memory occupied by a dynamic array at runtime is different in each programming language. In C, for example, one can use the `sizeof()` function to find out exactly how much memory does an array occupy. Fortran also provides a similar function, the `size()` intrinsic function, but which only returns the total number of elements in the array and not the total memory.

We thus defined in our Fortran implementation the memory occupied by a dynamic array as the product between the number of elements in the array and the size of one element.

$$memory_{array} = |elem \in array| * memory_{elem}$$

The size of an element depends first of all on its data type, but it may also depend on the compiler, the machine architecture and the operating system. Since Fortran2008, there is an intrinsic function which can be used to query the memory footprint of variables. The `storage_size` function returns the number of bits occupied by the given argument. It works on both standard and custom types of variables. If the argument is an array, it returns the size of one element in the array.

The formula used for gathering MPE information in bytes is thus:

$$memory = size(array) * storage\_size(array)/8$$

In the instrumentation process, this is deployed in the calls to the LDK Measurements library, which are inserted as described in the previous paragraph. Listing 20 shows several examples of the inserted calls to `ldk_alter_mpe()`. It is also worth observing that, in order to receive the correct results, the `size()` and `storage_size()` functions must be used at the appropriate point in time. One can see that when it comes to an `ALLOCATE` statement, the instrumented calls are added after the allocation, while for `DEALLOCATE` statement, the calls are inserted before the deallocation. This is important because the `size()` function, for example, could not return the number of elements in `myArray`, but only after these elements were allocated. Similarly, calling `size()` after `DEALLOCATE` would not be able any more to say how many elements were in `myArray` before the deallocation.

If, however, the compiler does not support the newer Fortran2008 standard, and hence the `storage_size()` function is not available, or if the programming language in which the application is implemented does not provide such a function at all, then another solution for determining the size of elements needs to be applied. We present here the alternative approach which we tested in our framework as well. Although it is implemented for Fortran, it can be adapted for other languages as well.

First of all, we defined a set of constants to hold the size in bytes of the standard types. The constants are given as parameters in a separate module, which can be changed and adapted by the user, if it is needed on the used platform.

```
module ldkparams

! size in bytes
integer, parameter :: LDK_Integer_SIZE = 4
```

```
integer, parameter :: LDK_Real_SIZE =  4
integer, parameter :: LDK_Double_SIZE = 8
integer, parameter :: LDK_Complex_SIZE = 8
integer, parameter :: LDK_DComplex_SIZE = 16
integer, parameter :: LDK_Char_SIZE = 1
integer, parameter :: LDK_Integer4_SIZE = 8
...
end module ldkparams
```

Then we designed a function which accepts as a parameter the node of a variable and returns a string representing summations of string literals like LDK_Complex_SIZE, LDK_Integer_SIZE, which are defined in the module mentioned above. The function first identifies in the symbol table the symbol of the given variable. It then recursively traverses the type definition in the symbol table building the return string. For example, given a variable of a custom type with two integer fields and one real field, the function will return the string `LDK_Integer_SIZE + LDK_Integer_SIZE + LDK_Real_SIZE`. If the argument is of a standard type, the function just returns the corresponding size parameter.

Although this solution works for most cases, there might also occur situations where the computed size might not accurately represent the actual size of variables. In Fortran, for example, the compilers are allowed to use *padding* in order to generate structures with aligned fields. Padding means adding bytes of unused memory, either between the fields, or at the end of the structure. This increases performance at the expense of memory usage. Table 3 demonstrates the effect of padding comparing the size computed by our function and the actual size including padding.

Although this solution does not always deliver the exact values for MPE, it could still be used as a workaround, focusing on the qualitative analysis of the results rather than the quantitative one.

7.3.2 *Iterometers instrumentation support*

In section 4.4 we defined *Iterometers* as a class of metrics which target the analysis of iteration processes within applications.

Following the same implementation questions as for MPE, we could state the following:

A. **what** makes the object of the metric?
Answer: iteration parameters.

It is important to differentiate here between the variables in the source code representing iteration indexes and the possible DK Objects or other code data structures which might be semantically understood as the *iteration parameters*. An iteration step can be identified by means of an index, but it could also be identified by means of the data which is being processed in that particular step.

B. **when** is the metric applied?
Answer: user-driven instrumentation.

Iterometers record values of indicated iteration parameters when and where these parameters are declared in DK annotations.

Type definition	Size string	Computed size	Actual size (with padding)

`TYPE intreal_t`
`integer :: a`
`real :: b`
`END TYPE intreal_t`
	LDK_Integer_SIZE + LDK_Real_SIZE	8	8

`TYPE int8real_t`
`integer(8) :: a`
`real :: b`
`END TYPE int8real_t`
	LDK_Integer4_SIZE + LDK_Real_SIZE	12	16

`TYPE int8realreal_t`
`integer(8) :: a`
`real :: b, c`
`END TYPE int8realreal_t`
	LDK_Integer4_SIZE + LDK_Real_SIZE + LDK_Real_SIZE	16	16

`TYPE int8int8real_t`
`integer(8) :: a, b`
`real :: c`
`END TYPE int8int8real_t`
	LDK_Integer4_SIZE + LDK_Integer4_SIZE + LDK_Real_SIZE	20	24

Table 3: Comparison between the computed size and the actual size of some variables of custom types, when padding is used on a 64-bit platform.

The needed instrumentation can be carried out already in the first instrumentation phase, when the LDK directives are processed. Iteration parameters are specified as attributes of DK Operations. There are two attributes which can be used to specify iteration parameters:

- `iterent`: the name of a DK Entity; and
- `iterparam`: the name of a formal argument or local variable.

When such an attribute is encountered in the definition of a DK Operation, a call to the `ldk_alter_iterometer()` function of the DK Measurements Library needs to be inserted in the parse tree. This DK Operation is a region-based operation and this ensures that the code is already instrumented with proper calls to markup the start and end of the operation, as explained in the event triggers instrumentation in section 6.1.3.1.

As in the case of inserting MPE calls for `ALLOCATE` and `DEALLOCATE` statements, there are some particular cases where the positioning of the `ldk_alter_iterometer()` call might be important. In the general cases, this call is inserted right at the beginning of the operation. If, however, the operation is of the `Comm_Recv` type, then the call needs to be inserted at the end of the operation. `Comm_Recv` operations represent communication actions where there is some data to be received. For example, it might include a `MPI_recv()` call. In this case, the iteration parameters should be traced only at the end of the operation, i. e. after their values were initialized in the receive call.

c. **how** is the metric evaluated?
Answer: variable values or DK objects ids.

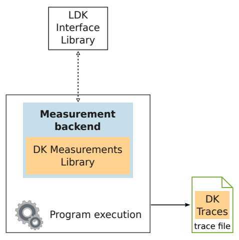
If the iteration parameter is a source code variable, declared with `iterparam`, then the value of the Iterometer is given by the value of that variable.

If, instead, the iteration parameter is a DK Entity, declared with `iterent`, then the value of the Iterometer is given by the id of the entity.

7.4 MEASUREMENT

There are mainly two components of our framework which are specific to the measurements step of the performance flow:

- the DK Measurement Library; and
- the DK Traces.



While the DK Traces could also be considered part of the *Visualisation* step, we include it here, as part of the tracing approach in the measurement library.

Unlike the DK Performance Strategy component, which is tightly integrated within the Periscope instrumenter, the two measurements components represent stand-alone implementations. Their functionality is based however on the API provided by the OTF2 library.

7.4.1 *OTF2 basics*

OTF2 [52] stands for *Open Trace Format* version 2 and is the follower of OTF. OTF was developed at the Technische Universität Dresden, in collaboration with the University of Oregon and the Lawrence Livermore National Lab. OTF2 was developed in the context of the Score-P project. OTF was producing ASCII trace files, while OTF2 is using the binary format.

The target of OTF2 is to support an efficient working process with very large trace files in serial and parallel environment.

At the core of the tracing format are the events, the locations, and the local and global definitions.

EVENTS

At the core of the tracing technique there are the different events, which are necessarily associated with a timestamp. In OTF2 one can register several types of events, each with corresponding associated properties:

- code region enter or leave;
- metric values;
- MPI operations;
- OMP operations;
- parameter values;
- RMA operations;
- thread operations.

LOCATIONS

OTF2 supports parallel I/O, providing separate writing streams for each separate location. Locations are usually associated with execution processes or threads.

The merging of the data gathered by each collection is taking place automatically through the functionality offered by the OTF2 library.

DEFINITIONS

Some event types are associated with different code or analysis elements like code regions, metrics or parameters. When such an event is recorded, only event specific information is begin stored, like the timestamp or the location of the occurrence. The properties of the referenced elements are given separately in the so called *definitions*.

OTF2 supports local and global definitions. The local definitions are given per location and can be mapped to global definitions, if such a mapping is provided.

An important aspect for the tools which use OTF2 as a writing library, is the fact that all these definitions can be written at the very end of the tracing processes. This is particularly convenient, science many of the information to be recorded in definitions is only given at runtime through the instrumentation calls.

7.4.2 *DK Measurements Library*

The DK Measurements Library comprises of a set of functions which build up the tracing interface of the framework. Calls to these functions are inserted in the instrumented version of the application and they are executed at the runtime of the application.

The component is implemented in C and can be built as a library, separately from the rest of the framework. It is then linked to the instrumented application and produces trace files at execution time.

The integration with Fortran applications is natively supported. Function calls from Fortran are linked to the implementations provided in the C library. The only sensible aspect is data types of arguments. To be on the safe side, the trace functions use only integer arguments.

The functions included in our implementation are as follows:

`void ldk_start_phase_(int* phaseId, int* phaseType):`

Registers the start of the phase with the given Id. The phase type is also provided and could be used for more specialized tracing. One could, for example suspend a given type of trace events until the application leaves the current phase.

Phases are implemented as OTF2 regions. Starting a phase means entering a new region. Since phases do not have ending commands, before actually registering the start of a new phase, first the event for leaving the current region, i. e. phase, is registered.

`void ldk_alter_mpe_(int* entId, int64_t* elemCnt, int* elemSize)`

Registers changes to the MPE metric. The id of the targeted entity is provided, along with the value which has to be computed as the product between the `elemCnt` and `elemSize`.

MPE is stored as a metric in OTF2, as described below in the section regarding the LDK Traces component.

`void ldk_alter_iterometer_(int* type, int* opId, int* value)`

Registers the changes to a given Iterometer. Iterometers are associated to an operation, which is provided by the `opId` argument. The type specifies whether it is an entity-valued Iterometer, or a variable-valued.

`void ldk_comm_entity_(int* type, * entId, int* srcProc, int* destProc)`

Registers communication or transfers of entities between application processes. The `type` specifies whether it is a *send* or a *receive*. The source process and the destination process are also being given and registered.

This function is not currently associated with an instrumentation strategy in LDK Strategy or LaDoK implementation. Its functionality was evaluated based on manually instrumented source files. The visualisation of the results also requires a post-mortem processing step of the trace files, which we implemented as a command-line tool.

7.4.3 *DK Traces*

The DK Traces is represented by the OTF2 trace files generated by our framework at the end of the performance measurement flow. Besides the registration of the events themselves, an important aspect is represented by the definitions we use in the OTF2 format in order to express our specialized metrics.

Throughout the entire tracing process, the information with respect to the DK Objects is retrieved via the LDK Interface Library of the LDK Framework.

MPE STORAGE IN OTF2

In order to store the MPE metric, the correct references to the DK Entities and the amount of the memory allocated or deallocated have to be stored.

1. **DK Entity:** In the LDK Traces each DK Entity has associated one OTF2 metric with:
   - metric identifier: the ID of the DK Entity;
   - metric name: the name of the DK Entity;
   - metric mode: OTF2_METRIC_RELATIVE_LAST, which means that values are added/subtracted to/from the last value of the metric.

   The DK Entities definitions are received from the LDK Interface component, which, in turn, first needs to load them into the LDK Objects component, as described in section 6.3.

2. **Amount of memory:** The amount of memory is passed as the value of OTF2 metric events `OTF2_EvtWriter_Metric()` with signed 64-bit integers `OTF2_TYPE_INT64` type.

DK PHASE STORAGE IN OTF2

DK Phases are defined using the OTF2 code region elements. For each DK Phase stored in the LDK Objects, one OTF2 code region is defined using:

- region identifier: the ID of the DK Phase;
- region name: the name of the DK Phase;
- region type: `OTF2_REGION_FLAG_PHASE`.

OTF2 TRACES OF DK ENTITIES TRANSFERS

In order to store the communication or transfers of DK Entities, the OTF2 code region is used again. This time, there are only two regions with given predefined semantics: one region where entities are sent and another one where entities are received. The two regions have a custom chosen identifier and the custom names `SENDENT_Region` and `RECVENT_Region`.

The actual relevant information, namely the id of the DK Entity involved in the transfer, as well as the identification of the sending and receiving processes are stored as attributes of the region entering event.

The attributes are defined in the definition section of the LDK Traces. There are in this case three attributes with hard-coded identifiers:

```
enum TraceAttrT_Type{
        TraceAttrT_ENTID = 0, /* the id of the entity */
        TraceAttrT_SRCPROC = 1, /* the source process */
        TraceAttrT_DESTPROC = 2, /* the destination process */
        TRACEATTRTYPE_CNT
};
```

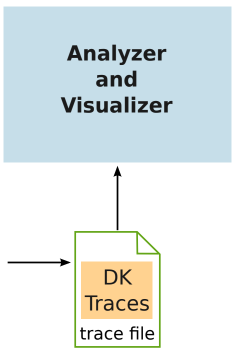
Upon calling the `ldk_comm_entity_` function, the values retrieved as arguments are assigned to the attributes. These are then given as the current attributes list to the enter event for the corresponding region. and added to an attributes list.

```
OTF2_EvtWriter_Enter(evt_writer, attribute_list, get_time(), regionRef);
OTF2_EvtWriter_Leave(evt_writer, NULL, get_time(), regionRef);
```

Since the region is only needed to signal the type of the transfer, wither send or receive, and to store the corresponding parameters, after issuing the enter event, the region is also begin left immediately.

7.5 ANALYSIS AND VISUALISATION

There are two main aspects to be considered for the analysis and visualisation component of our DK-enhanced profiling framework.



First of all, the DK Traces generated during the measurements phase are using the open-source OTF2 format. Hence, any tool capable of reading OTF2 files could be used to visualise the results.

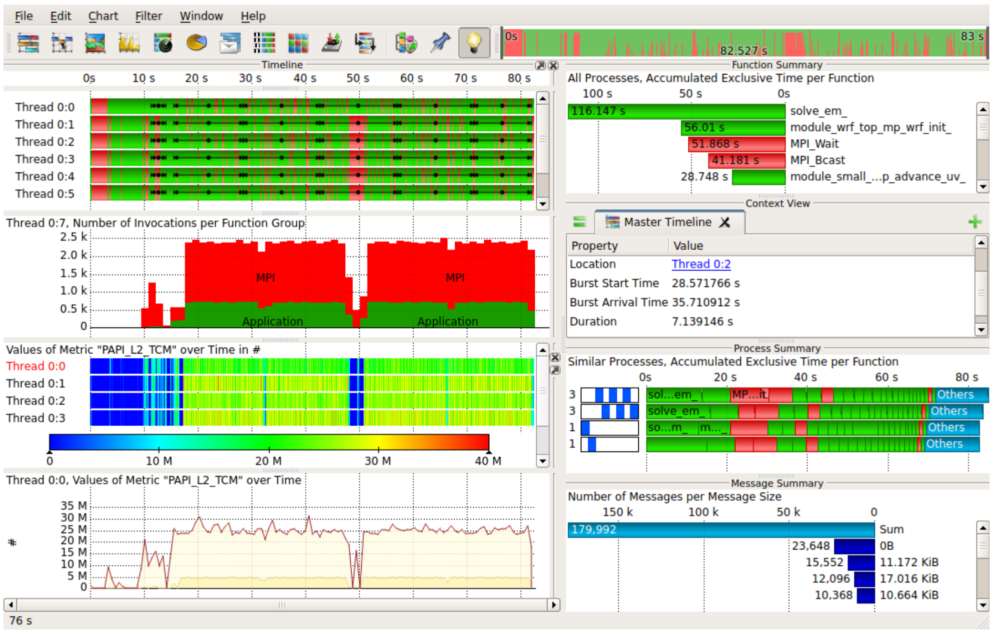
Secondly, all information regarding the DK approach is already encoded in the DK Traces through the specialized design of the registered events and metrics. Thus, the visualisation tool does not need to interact in any way with the LDK Framework.

These two facts allowed us to use for analysis and visualisation within our framework a tool which is not open-source and for which we do not have access to the source code. The *Vampir* GUI is able to display out-of-the-box the DK Traces produced in the measurements phase.

Figure 22 presents the Vampir screen with many views enabled, for one of the standard example files delivered along with the software, namely the *Large_ScoreP/traces.otf2* file.

On the left hand side of the screen there are the trace views displaying different events in their chronological order, from the left to the right. The most common is the *Timeline* view, displayed here at the very top, with one horizontal bar per process. The next view below is an example of a *Summary Timeline*. In this case, the number of invocations for each function is being displayed. The third chart is a *Performance radar* view, displaying the values of the `PAPI_L2_TCM` counter, which is the PAPI counter for the total cache misses. At the bottom one can see another view for this same counter. This time the chart is including the results for only one process.

On the right hand side of the screen there are the profiling views and contextual information. The first diagram on top is the *Functions Summary*, displaying the total time spent in each function. The view below this is reserved for the contextual information. Upon clicking anywhere on the Vampir screen, the corresponding available information will be displayed here. Further below there is the *Process Summary* chart displaying the accumulated exclusive time per function. Finally, the last view is the


File
Edit
Chart
Filter
Window
Help
Timeline
Function Summary
All Processes, Accumulated Exclusive Time per Function
116.147 s solve_em_
56.01 s module_wrf_top_mp_wrf_init_
51.868 s MPI_Wait
41.181 s MPI_Bcast
28.748 s module_small_...p_advance_uv_
Thread 0:0
Thread 0:1
Thread 0:2
Thread 0:3
Thread 0:4
Thread 0:5
Thread 0:7, Number of Invocations per Function Group
MPI
Application
Values of Metric "PAPI_L2_TCM" over Time in #
Thread 0:0, Values of Metric "PAPI_L2_TCM" over Time
Context View
Master Timeline
Process Summary
Similar Processes, Accumulated Exclusive Time per Function
Others
Message Summary
Number of Messages per Message Size
76 s

Property	Value
Location	Thread 0:2
Burst Start Time	28.571766 s
Burst Arrival Time	35.710912 s
Duration	7.139146 s

179,992	Sum
23,648	0B
15,552	11.172 KiB
12,096	17.016 KiB
10,368	10.664 KiB

Figure 22: Vampir screen for an MPI application.

Message Summary chart with a histogram displaying the number of MPI messages per message size.

All views are connected and responding to the selections performed on the timeline-based views on the left hand side. One can zoom in and out both on horizontal and vertical directions.

Vampir manages without a problem very large trace files, scaling well both for a large number of execution processes, as well as for very long execution times. Here the capability of aggregating several events, like for example MPI messages, is of a crucial importance.

7.6 EXTENSIONS OF THE DK FRAMEWORK

Through its structure and design, the DK-enhanced framework easily supports the addition of other custom metrics, for example. Once the corresponding instrumentation strategy is added to the *DK Performance Strategy* components, the measurements library can be adapted to trace the necessary information.

Some special visualisation and analysis extensions could also be considered. These depend though on the visualisation tool, as presented in the previous section.

8 USE CASE: MEMORY-INTENSIVE APPLICATIONS

The term *memory-intensive* received some few slightly different interpretations within the application performance community. In this chapter we refer by memory-intensive to that class of applications, which use throughout their execution a large amount of *intermediary memory*, required by the many temporary data structures which are generated.

This is different from the *data-intensive* applications which either need to process a large amount of *input data*, or which generate large amounts of *output data*.

Also, *memory-intensive* is different from *memory-bound*. The first one is a classification strictly referring to the evolution of the memory footprint throughout the execution of applications, while the second classification refers to the *memory access profile* in relation to the computing operations.

The three classifications do not exclude each other, they just refer to different aspects when characterizing applications.

Examples of common applications for each classification include [75]:

- memory-intensive: graph-based algorithms, quantum chemistry, (de novo) DNA sequence assembly;
- data-intensive
  large input: data-mining, visualisation applications, genome computation;
  large output: climate modelling, molecular dynamics, structural mechanics;
- memory-bound: PDE solvers using stencil-based methods, sparse-matrix algorithms.

Due to the dynamic aspect of the temporary data structures used in memory-intensive computation, it is often difficult to predict the amount of memory used by the application throughout the execution. The size of the temporary data structures usually depends on the input data and sometimes also on the computed data.

Estimations of the amount of required memory exist, of course, as the size complexity analysis of scientific algorithms is common practice, like the time complexity as well. There are two aspects which need to be considered with respect to size complexity, though. First of all, these are usually only *upper limit* asymptotic estimations, expressed mostly using the big $\mathcal{O}$ notation. Secondly, the size complexity analysis only expresses the *behaviour* or the *scalability* of the algorithm with respect to some characteristics of the input dataset. It does not, however, give an estimation of the actual size of the required memory, e.g. number of bytes or MB. The reason why size complexity fails in these two aspects, is that it only considers the theoretical formulation of algorithms. In practice, the choice of the implementation approach, as well as the implementation language do influence the overall behaviour of an algorithm with respect to the memory requirements.

Some performance tools, like Vampir and data-centric tools [66], offer the possibility to track memory usage, either at system-level or at single variables-level. For large simulations, the first solution is too coarse-grained to be able to deduce problematic data structures, while the second one usually proves to be too fine-grained and also lacking context information.

In this chapter we provide a solution to these challenges, based on the three steps of the DK approach presented in section 4.2. We first introduce the quantum chemistry application ParaGauss. Then we proceed with the first step of the DK approach: *identifying* the relevant knowledge (Domain Knowledge) with respect to the memory-intensive aspect. We then show how to *express* the identified knowledge by inserting LaDoK annotations in the source code. In the third step we focus on the *usage* of the expressed knowledge for an improved performance analysis, namely we present the results obtained using our DK-enhanced profiling framework and the MPE DK Metric. Finally, we draw conclusions regarding the benefits of DK for this use case and suggest further performance analysis for the code.

8.1 QUANTUM CHEMISTRY APPLICATION: PARAGAUSS

ParaGauss [6] is an application developed at Technische Universität München within the Theoretical Chemistry research group. It is a "parallel DFT code for solving challenging electronic structure problems in chemistry, surface science, and the field of nanostructured materials" [46]. In chapter 1, we already mentioned that, in the HPC context, DFT (Density Functional Theory) is the most widely applied electronic structure approximation in computational chemistry and computational materials science. ParaGauss thus subscribes to the main stream of HPC chemistry applications.

ParaGauss is targeting highly symmetric structures for which matrix blocking methods can be applied. It is also exposing a good performance for the complex computations of metallic/conductive systems.

8.1.1 *Density Functional Theory (DFT)*

Density functional theory is a reformulation of quantum mechanics which, in the case of electronic ground states, offers an alternative to the Schrödinger equation. DFT is based on the electron density instead of the complicated many-body wave functions. Actual DFT approximations provide a computable expression for the total energy E whose variational treatment leads to the electronic structure of the system under study.

In the Schrödinger formulation, the total energy is given by:

$$E = \frac{\langle\Psi|H|\Psi\rangle}{\langle\Psi|\Psi\rangle}, \tag{1}$$

where Ψ is the electronic wave function depending on the spatial coordinates of all electrons of the system, and H is the Hamiltonian operator.

HK: from 3n arguments to only 3.

On the other hand, DFT is based on the Hohenberg-Kohn theorem which expresses E as a functional[1] of the electron density ρ. The electronic density is the probability of finding any of the n electrons of a system within volume element $d\vec{r}$. Unlike the wave function, ρ is depending only on three arguments, namely the three coordinates

1 A functional is a function which has as argument another function, while a common function has as argument a variable.

of the system and it is actually a physical observable. The connection between ρ and the many-body wave function in Schrödinger's equation is given by:

$$\rho(\vec{r}) = n\int \dots \int |\Psi(\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n)|^2 d\vec{x}_2 \dots d\vec{x}_n, \tag{2}$$

where x_i has both a position and a spin component: $x_i = \{r_i, \sigma_i\}$. In the following though, we are going to use the spin-restricted formulation only.

Most realizations of DFT rely on the Kohn-Sham formalism, which, in turn, uses the following partitioning of the total energy:

$$E[\rho] = E_{ext} + E_{kin} + E_{coul} + E_{xc} \tag{3}$$

- E_{ext}: the potential energy arising from the atomic nuclei of the system;

  $$E_{ext} = \int \sum_{A}^{M} \frac{Z_A}{|\vec{r}_{1A}|} \rho(\vec{r}) d\vec{r} \tag{4}$$

  where

  M : number of atoms in the electronic systems;

  Z_A : atomic number of atom A;

  $\vec{r}_A$: position vector of nucleus A;

- E_{kin}: the kinetic energy of all electrons within the system;

- E_{coul}: the pairwise classical electrostatic interaction energy between any two electrons of the system;

  $$E_{coul} = \frac{1}{2} \int \int \rho(\vec{r}_1) \frac{1}{|\vec{r}_{12}|} \rho(\vec{r}_2) d\vec{r}_1 d\vec{r}_2 \tag{5}$$

  where $\vec{r}_1$ and $\vec{r}_2$ are the positions of the two different electrons.

- E_{xc}: the exchange-correlation energy including all non-classical quantum mechanical electron-electron interactions.

The exact form of the functionals representing the kinetic E_{kin} and the exchange-correlation E_{xc} energies are not known. In order to express the kinetic energy term E_{kin}, the Kohn-Sham formalism introduces a set of single-electron orbitals under the assumption:

$$\rho(r) = \sum_{i}^{n} |\psi_i(\vec{r})|^2 \tag{6}$$

where

n : number of electrons;

ψ_i : a one-electron wave function (as opposed to Ψ which is the all-electron wave function in equation 1).

In principle, this is only a mathematical approach, which does not change any of the physical or chemical semantics. The largest part of the kinetic term E_{kin} can be expressed now in terms of the one-electron wave functions ψ:

$$E_{kin} = -\frac{1}{2}\sum_{i}^{n} \langle \psi_i | \nabla^2 | \psi_i \rangle \tag{7}$$

KS: from many-electron to one-electron equations.

In DFT one seeks to find the ground-state energy, from which many properties of the system can be derived afterwards. Following the variational principal, the ground-state energy is given by minimizing the energy expression. Thus every ψ_i emerges as a solution of the *Kohn-Sham equation:*

$$\hat{h}_{KS}\psi_i = \epsilon_i \psi_i \tag{8}$$

with the one-electron KS operator given by:

$$\hat{h}_{KS} = h_{kin} + h_{ext} + h_{coul} + h_{xc} \tag{9}$$

$$= -\frac{1}{2}\nabla^2 - \sum_{A}^{M} \frac{Z_A}{|\vec{r}_{1A}|} + \int \frac{\rho(\vec{r}_2)}{|\vec{r}_{12}|} d\vec{r}_2 + \nu_{xc}(\vec{r}_1) \tag{10}$$

Since $\hat{h}_{KS}$ depends on ρ, which is computed from ψ_i, the equations 8 cannot be computed directly and hence an iterative self-consistent field (SCF) approach needs to be applied.

LCAO: from analytic to algebraic representation.

In order to do so, the equations 8 are adapted for the computer-based calculations by means of the LCAO (Linear Combination of Atomic Orbitals) machinery. LCAO is used to transform the analytic formulations of the formulae into a linear algebra formulation. The unknown one-electron wave functions are replaced by a linear combination of known basis functions, called *orbital basis functions*. The coefficients by which the basis functions are combined become the unknowns in the new system of equations. The unknowns are gathered in matrix- or vector-like representations, which are suitable for computer-based calculations.

There are several types of basis functions which can be used in DFT implementations, such as Gaussian-type-orbitals (GTO), Slater-type-orbitals (STO), and plane waves. ParaGauss uses products of Gaussian radial functions and solid spherical harmonics [49]. The one-electron orbital functions are thus expressed as:

$$\psi_i = \sum_{\mu}^{N} c_{\mu i}\chi_{\mu} \tag{11}$$

with

$$\chi_{\mu} \propto Y_l^m(r-A)e^{-\alpha(r-A)^2} \tag{12}$$

where:

χ_{μ} : atom-centered orbital basis functions;

Y_l^m : polynomial harmonic function;

l : angular momentum;

m : magnetic number;

α : exponent of the gaussian radial function;

N : number of basis functions.

An important observation is that the orbital basis functions χ_μ are atom-centered. This means that their values depend on the (known) positions of the nuclei and thus they can be computed in forehand.

Replacing ψ_i in equation 8 with the linear expression from equation 11, multiplying afterwards to the left with an arbitrary orbital basis function χ_ν and then integrating over the space, leads to the new form of the equations [13]:

$$\sum_\mu c_{\mu i} \int \chi_\nu(\vec{r}_1)\hat{h}_{KS}\chi_\mu(\vec{r}_1)d\vec{r}_1 = \epsilon_i \sum_\mu c_{\mu i} \int \chi_\nu(\vec{r}_1)\chi_\mu(\vec{r}_1)d\vec{r}_1 \tag{13}$$

This system of N equations can be rewritten in algebraic form as a generalized eigenvalue problem:

$$\hat{H}^{KS}\hat{C} = \hat{S}\hat{C}\hat{\epsilon} \tag{14}$$

where all matrices are of size NxN:

$\hat{H}^{KS}$ is called the Kohn-Sham matrix;

$\hat{S}$ is called the overlap matrix; and

$\hat{C}$ is the matrix of the unknown coefficients $c_{\mu i}$.

and the matrix elements are given by:

$$\hat{H}^{KS}_{\nu\mu} = \int \chi_\nu(\vec{r}_1)\hat{h}_{KS}\chi_\mu(\vec{r}_1)d\vec{r}_1 \qquad \hat{S} = \int \chi_\nu(\vec{r}_1)\chi_\mu(\vec{r}_1)d\vec{r}_1 \tag{15}$$

Expanding $\hat{h}_{KS}$ as in formula 9 and grouping the kinetic and external terms together, leads to the two groups of integrals specific to DFT:

A. one-electron integrals:

$$h_{\nu\mu} = \int \chi_\nu(\vec{r}_1)(-\frac{1}{2}\nabla^2 - \sum_A^M \frac{Z_A}{|\vec{r}_{1A}|})\chi_\mu(\vec{r}_1)d\vec{r}_1 \tag{16}$$

B. two-electron integrals:

$$J_{\nu\mu} = \sum_\lambda \sum_\sigma P_{\lambda\sigma} \int\int \chi_\nu(\vec{r}_1)\chi_\mu(\vec{r}_1)\frac{1}{|\vec{r}_{12}|}\chi_\lambda(\vec{r}_2)\chi_\sigma(\vec{r}_2)d\vec{r}_1 d\vec{r}_2 \tag{17}$$

where P is called the density matrix and is given by

$$P_{\lambda\sigma} = \sum_i^N c_{\nu i}c_{\mu i} \tag{18}$$

The two-electron integrals 17 are also called four-center integrals, since there are four different atom-centered orbital basis functions involved: χ_ν, χ_μ, χ_λ and χ_σ. Four-center integrals are very computation and space demanding and, hence, there are often replaced by three-center integrals. The latter are obtained by introducing a new approximation of the electronic density via an additional set of Gaussian basis functions f_k, also called *fitting functions* [6]. The three-center integrals are thus given by:

$$J'_{\nu\mu} = \sum_k d_k \int\int \chi_\nu(\vec{r}_1)\chi_\mu(\vec{r}_2)\frac{1}{|\vec{r}_{ij}|} f_k(\vec{r}_2) d\vec{r}_1 d\vec{r}_2 \tag{19}$$

The exchange-correlation term ν_{xc} is not included in equations 16 and 17. Due to its complex form, it is usually computed via numerical integration [6]:

$$h^{xc}_{\nu\mu} = \sum_n \omega_n \chi_\nu(r_n)\nu_{xc}(r_n)\chi\mu(r_n), \quad r_n \in \text{ grid} \tag{20}$$

The self-consistent field (SCF) method is used to iteratively solve the system of equations 14. One starts with an initial guess of the solution, the density matrix P in equation 18. This initial guess is inserted in equation 17 in order to compute the Kohn-Sham matrix H^{KS} given in equation 15. The next step is to solve equation 13, expressed as generalized eigenvalue problem in equation 14. The new solution is compared with the previous one and, if necessary, a new iteration step is started.

8.1.2 *Implementation*

ParaGauss is a Fortran 95 parallel application, using MPI and a C wrapper for communication.

The general execution path of ParaGauss is represented in figure 23 and it also represents the common approach for DFT implementations.

For the purpose of this chapter we concentrate here only on the *Integrals* step of the execution. Further details with respect to the implementation and parallelization strategies used in ParaGauss can be found in [6], [49] and [64]. Also, a very good survey on parallel implementations for quantum chemistry is given in [35].

In the Integrals step, the set of one- and two-electron integrals are (pre-)computed and stored. These are the integrals given by the formulae 16 and 17. They are computed here once and then they are reused in each step of the SCF iteration.

As mentioned above, ParaGauss uses two-center integrals for the 1-electron terms and three-center integrals for the 2-electron term. All necessary data is stored in dedicated data structures which are grouped in separate Fortran modules. The computation is parallelized based on batches or groups of integrals, called *quadrupels*. The name *quadrupel* comes from the four (*quad*) indices which identify each such batch: (ua_1, l_1, ua_2, l_2). ua_1 and ua_2 identify the two atoms involved in the current computation within all unique atoms defining the electronic system. l_1 and l_2 are the corresponding angular momentum numbers, as given in equation 12 as well.

The quadrupels are distributed by the master process and the computations are performed by the slave processes.

8.2 RELEVANT DOMAIN KNOWLEDGE

Throughout this thesis we pointed out that one of the key characteristics with respect to DK is that it is usually only *implicitly* present within the source code. In order to

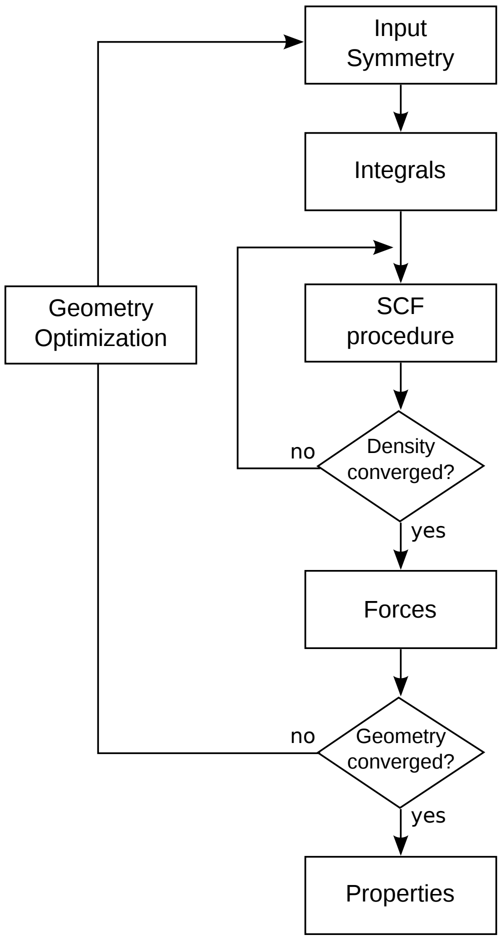


Figure 23: ParaGauss general execution flow [6].

make it *explicit*, one first needs to recognize its traces in the multitude of available information.

In our current use case we could identify three different sources of such information.

First of all, we have all the standard information which is present in the quantum chemistry textbooks when discussing the computation of integrals for DFT implementations. For example:

- one-electron integrals are easy to compute and do not require a lot of space;
- two-electron integrals are the bottleneck: require huge amount of space and are very many (up to $O(N^4)$);
- vanishing integrals exist, but they cannot be estimated exactly in forehand;
- both computation and storage costs can be reduced by considering symmetry properties of systems[2];
- primitives can be reused within integrals of the same shell.

2 Some electronic systems expose symmetries which are used to group atoms into given types, referred to as *unique atoms*. The computations, including integrals evaluation, can then be performed for this smaller number of *unique atoms* alone.

Secondly, there is the knowledge gathered within the development of ParaGauss, knowledge which is partially documented in the form of source code comments or published papers. Three examples are:

- for the two-electron integrals Paragauss uses either 3- or 4-center integrals;
- due to the parallelization of the computation of the integrals, after the computation of each quadrupel, the results need to be scattered among the slave processes; and
- in the implementation of the relativistic correlation part of the E_{xc} term in equation 3, large uncontracted basis functions are used, requiring significant memory space.

Lastly, there are results gathered from performance analysis experiments ran by the developers of ParaGauss. The software has a built-in measurement system for execution time, but no support for memory measurements. Examples of acquired information are:

- the execution time of the Integrals step strongly varies with the input electronic system and the computation setup and could take anything from 10%, up to 40% of the total execution time; and
- the ratio with respect to execution time of all other computation phases strongly vary with the input electronic system as well.

8.3 LADOK ANNOTATIONS

We are now interested on how we can insert LaDoK annotations to the ParaGauss source code, given the DK elements listed above. We bare in mind that our goal is to gain an insight into the memory requirements of the Integrals step within ParaGauss.

To demonstrate the application of LaDoK we chose three statements from the lists above:

A. *Two-electron integrals require huge amount of space.*
   There is one Fortran module in ParaGauss dedicated to storing the computed two-electron integrals: `integralstore_module.f90`. We simply enclose the entire section for variable declaration at the beginning of the module by one DK Entity definition:

```
module integralstore_module
\begin{entity}{Integrals}
 real(kind=r8_kind), allocatable, target, dimension(:), public  :: &
      integralstore_2cob_kin, &
      integralstore_2cob_nuc, &
      integralstore_2cob_efield, &
          ...
\end{entity}
contains
...
end module integralstore_module
```

The LDK Framework reported afterwards at compilation time, that there were 14 variables registered to the `Integrals` entity.

B. *Unique atoms are used when symmetry is considered.*
In the file `unique_atom_module.f90` we find the data structures designed to store the unique atoms. Here, again, we enclose all variable declarations within one DK Entity:

```
module unique_atom_module
...
type, public ::  unique_atom_type
...
end type unique_atom_type

\begin{entity}{UniqueAtoms}
 integer(kind=i4_kind), public           :: N_unique_atoms = 0
    ! Number of unique atoms
 integer(kind=i4_kind), public           :: N_moving_unique_atoms
    ! Number of non-fixed unique atoms
 type(unique_atom_type), pointer, public :: unique_atoms(:)
    ! unique_atom(N_unique_atoms)
 type(unique_atom_type), pointer, public :: unique_atoms_eperef(:)
    ! unique_atoms(N_unique_atoms_eperef)
           ...
\end{entity}
end module unique_atom_module
```

The `UniqueAtoms` entity is assigned a total number of 15 variables, out of which 4 have complex user-defined types.

C. *Significant memory is required for the computation of the relativistic correlations.*
In ParaGauss, the main computations for the relativistic correlation contributions to the E_{xc} take place in the `do_one_block()` subroutine declared in the `relgrads.f90` file and called in a loop over the total number of irreducible representation blocks. In this subroutine, there are a set of local matrices which are used for the computations. We annotate them as follows:

```
subroutine do_one_block(irr,n,n_c)
   ...

\begin{entity}{UFandUB}
    type(rmatrix)  :: UF,UB
\end{entity}

\begin{entity}{Nuc}
    type(rmatrix)  :: Nuc
\end{entity}

\begin{entity}{Vrel}
    type(rmatrix)  :: V_rel
\end{entity}

\begin{entity}{NucInMomSpace}
    type(rmatrix)  :: VP
```

```
\end{entity}

\begin{entity}{UncontractedMatrices}
    \begin{entity}{UncontractedOverlap}
        type(rmatrix)  :: S
    \end{entity}
    \begin{entity}{UncontractedKin}
        type(rmatrix)  :: T
    \end{entity}
    \begin{entity}{UncontractedPVSP}
        type(rmatrix)  :: PVSP
    \end{entity}
\end{entity}

\begin{entity}{ContractedMatrices}
    \begin{entity}{ContractedOverlap}
        type(rmatrix)  :: S_c
    \end{entity}
    \begin{entity}{ContractedKin}
        type(rmatrix)  :: T_c
    \end{entity}
    \begin{entity}{ContractedVrel}
        type(rmatrix)  :: V_rel_c
    \end{entity}
\end{entity}

\begin{entity}{DiagT}
    type(rdmatrix) :: t_diag
\end{entity}

\begin{entity}{DiagTp}
    type(rdmatrix) :: Tp
\end{entity}
...
end subroutine do_one_block
```

Note that we intentionally used only DK Entity annotations, as these are sufficient for performing MPE measurements. DK Operations and DK Phases could be used as well for annotating the code, leading to other types of measurements, as explained later in chapter 9.

8.4 DK METRIC: MEMORY PER ENTITY

According to the definition given in section 4.4, MPE is a pure DK metric which shows the memory usage of entities throughout the execution of an application.

Given the LaDoK annotations above, we are interested in analysing the memory usage of the given DK Entities. We expect to see large amounts of data allocated for the *Integrals* and *UncontractedMatrices* entity and small to medium amounts for the *DiagonalMatrices* and *UniqueAtoms*. For the entities of the relative correlation computation we also expect more variations due to the temporary allocations and deallocations throughout the computation.

The MPE analysis should give some clear figures as regarding the quantitative aspect of the memory usage. How large are the entities in terms of MB or GB, for different electronic structures? How does the application and memory usage scale with increasing number of computing processors?

For this experiment we used the frameworks presented in chapters 6 and 7 to process the LaDoK annotations and insert the automatic instrumentations in ParaGauss. For the visualisation of the trace files we used Vampir 8.5.0.

The input systems are a cluster of 38 Palladium atoms and a smaller one, of only 6 Palladium atoms. For each electronic system we considered combinations of the two different computation options:

- **symmetry**: we used either OH or C1. The first one leads to using, for example, only 3 unique atoms for Pd38, while the second one preserves all 38 atoms as unique atoms for the same case;
- **pseudo-potential** or **relativistic**: labelled with `pp` and `ar`, respectively, these are two mutually exclusive setups. In `pp` only the valence electrons are used in computations, while the rest, closer to the atomic nucleus are replaced by an effective core potential. In contrast, in `ar` all electrons are used, adding the relativistic correlation contribution as well.

In the remainder of this section we first prove the success of the combination of the dynamic information tracking mechanism presented in section 6.1.3.2 with the entity tracing feature implemented in our DK-enhanced performance framework. We then proceed with a quantitative analysis provided by MPE for the entities annotated above. Lastly, we look at the memory usage behaviour by using a qualitative approach to the MPE measurements.

8.4.1 *Result: proof of the DK dynamic information feature*

We consider the DK measurements for one ParaGauss run using the Pd38-ar-OH as input system and configuration.

The implementation of the `do_one_loop()` routine which we annotated above, presents some difficulties for data oriented profiling:

1. the memory allocation calls are encapsulated in separate routines, outside the scope of the actual variables;
2. all subroutines responsible for allocations are grouped under a unique interface definition, interface which is used for all calls targeting memory allocation for variables;

```
interface alloc
     module procedure ralloc
     module procedure ralloc_12
     module procedure ralloc_many
     module procedure calloc
     module procedure calloc_12
     module procedure calloc_many
     !
     ! rdmatrix:
        module procedure alloc_rd
        module procedure alloc_many_rd
        !
        ! chmatrix:
        module procedure challoc
    end interface
```

3. there are multiple levels in the callpath from the call involving the targeted variable until the actual memory allocation call;

```
! relgrads.f90
call alloc(n, S, T, Nuc, PVSP)

! matrix_methods        .f90
subroutine ralloc_many(n,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9)
    implicit none
    integer(IK),intent(in)      :: n
    type(rmatrix),intent(inout) :: a0,a1,a2,a3,a4,a5,a6,a7,a8,a9
    optional a2,a3,a4,a5,a6,a7,a8,a9
    !** End of interface ***

    call alloc(n,a0)
    call alloc(n,a1)
    if(present(a2)) call alloc(n,a2)
    if(present(a3)) call alloc(n,a3)
    if(present(a4)) call alloc(n,a4)
    if(present(a5)) call alloc(n,a5)
    if(present(a6)) call alloc(n,a6)
    if(present(a7)) call alloc(n,a7)
    if(present(a8)) call alloc(n,a8)
    if(present(a9)) call alloc(n,a9)
end subroutine ralloc_many

! matrix_methods        .f90
subroutine ralloc(n,a)
    integer(IK),intent(in)      :: n
    type(rmatrix),intent(inout) :: a
    !** End of interface ***

    ! ...
    a%n1 = n
    a%n2 = n
    allocate(a%m(n,n))
    ...
  end subroutine ralloc
```

4. interface calls can be executed on up to 10 different variables at a time.

Besides these callpath related issues, there is also worth to notice that the variables are not declared as simple Fortran arrays. Instead, they are of custom defined types, each of these types containing then at least one field of array type.

Due to the dynamic information tracking capabilities provided by the LaDoK framework and the specialized entity tracing feature of the DK-enhanced framework, correct assignment of the memory allocation for the defined entities was obtained.

```
\begin{entity}{Overlap}
   type(rmatrix)  :: S
\end{entity}
\begin{entity}{Kinetic}
   type(rmatrix)  :: T
\end{entity}
...
call alloc(n, S, T)
```

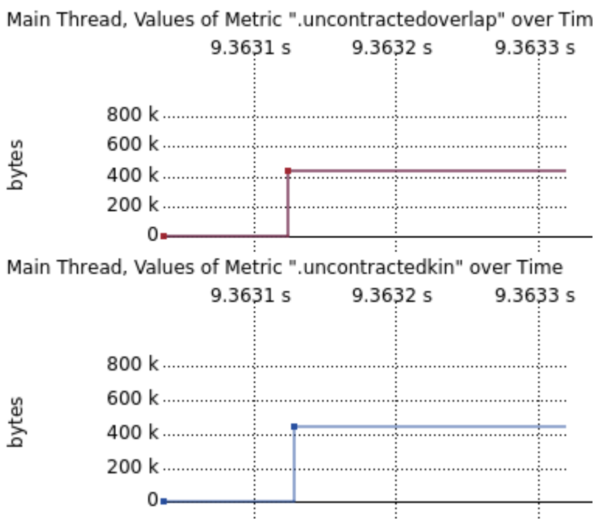


Figure 24: Left: One call single call using two different entities. Right: Memory allocations registered separately for each entity.

```
\begin{entity}{UFandUB}
type(rmatrix) :: UF,UB
\end{entity}
...
call alloc(n,UF,UB)
...
call pgfree(UF)
...
call pgfree(UB)
```

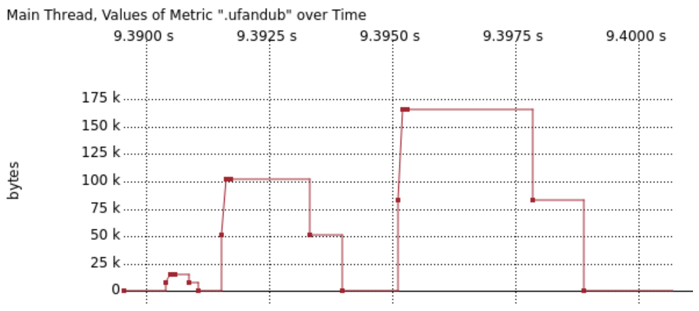


Figure 25: Left: Multiple calls for same entity. Right: 3 executions of the code on the left: allocation and deallocations cumulated correctly to the same entity.

We would like to point out, that no other user instrumentation besides the LaDoK annotations of the entities was required.

Figures 24 to 26 highlight three examples of tracing results. The diagrams were exported from the corresponding metric views in Vampir.

In figure 24 there is one call of the `alloc` interface for the `S` and `T` variables. The two variables were assigned to different entities via the LaDoK annotations. On the right hand side, one can see the MPE diagram for each entity. The memory allocation was registered correctly for each entity.

In figure 25 there is one call of the `alloc` interface for two variables belonging to the same entity, followed by consecutive single calls for memory deallocation. On the right hand side the MPE measurements for the entity and for three consecutive runs of the `do_one_block()` routine are displayed. One can see that, in each run there is one large memory allocation cumulating both variables. The deallocations are registered correctly in the account of the same entity as well.

Another example we provide here is related to the initialisation phase, at the very beginning of any ParaGauss run. Upon reading the input file, the necessary data structures to hold the information for the electronic system are initialized. In sec-

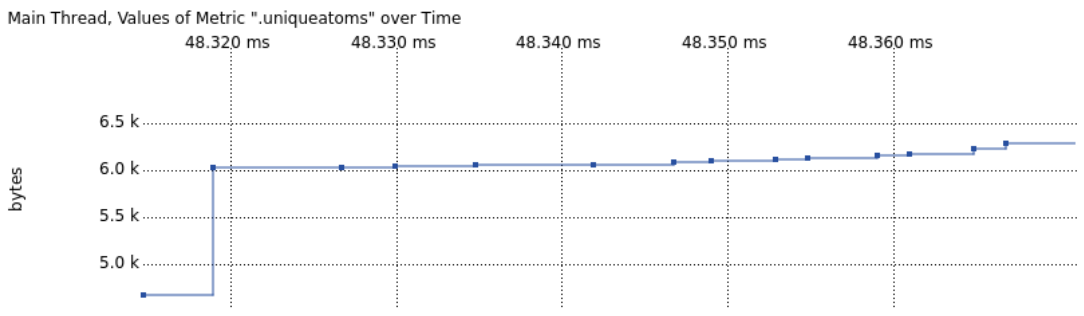


Figure 26: An initial memory allocation of 4.6 KB for an array with only 3 elements.

tion 8.3 we have defined the entity *UniqueAtoms*. One variable belonging to this entity is an array of elements of the type `unique_atom_type`.

Due to the OH symmetry, the input Pd38-ar-OH defines only 3 unique atoms. Looking in figure 26 at the traces for the *UniqueAtoms* entity, though, one can observe right at the beginning a large memory allocation of about 4.5 KB. Indeed, checking the definition of the `unique_atom_type` type in the source code brought to light a very complex and large custom type. The detailed logs output by the framework also report: there were 3 elements allocated, each of 1552 bytes. Thus, our tracing systems proved to be successful also when working with complex custom types.

8.4.2 *Result: quantitative assessment of memory usage*

Quantitative assessment refers in our context to the conclusions which can be drawn from a set of measurements with respect to the *quantity* represented by the measured metric values.

We use as input the Pd38 atomic cluster. We already know that, in ParaGauss, the storage complexity of the two-electron integrals scales with $\mathcal{O}(N^3)$, with N being the number of basis functions. For the Pd38-ar input system there are in total $N = 1710$ basis functions and $N_{ff} = 2394$ fitting functions. This results in a total number of 3.502.194.570 integrals for the C1 symmetry and at least 350.219.457 integrals for the OH symmetry version.

The practical question is now: do they fit in the main memory? How many MB or GB do these numbers mean?

Using the simple annotations for *Integrals* in section 8.3, we ran ParaGauss with three configurations:

1. Configuration 1: Pd38-ar-OH on 4 processes;
2. Configuration 2: Pd38-ar-C1 on 4 processes; and
3. Configuration 3: Pd38-ar-C1 on 8 processes.

The results in figure 27 clearly show that Configuration 1 could fit on any common machine, the *Integrals* entity occupying only about 9 MB of memory. On the other hand, Configuration 2 has a very large memory requirement of about 7 GB. Running it on more processes though, like in Configuration 3, reduces the size of the *Integrals* entity on each process. Running with an appropriate number of processes, the C1 symmetry might thus also fit into the main memory of a common machine.

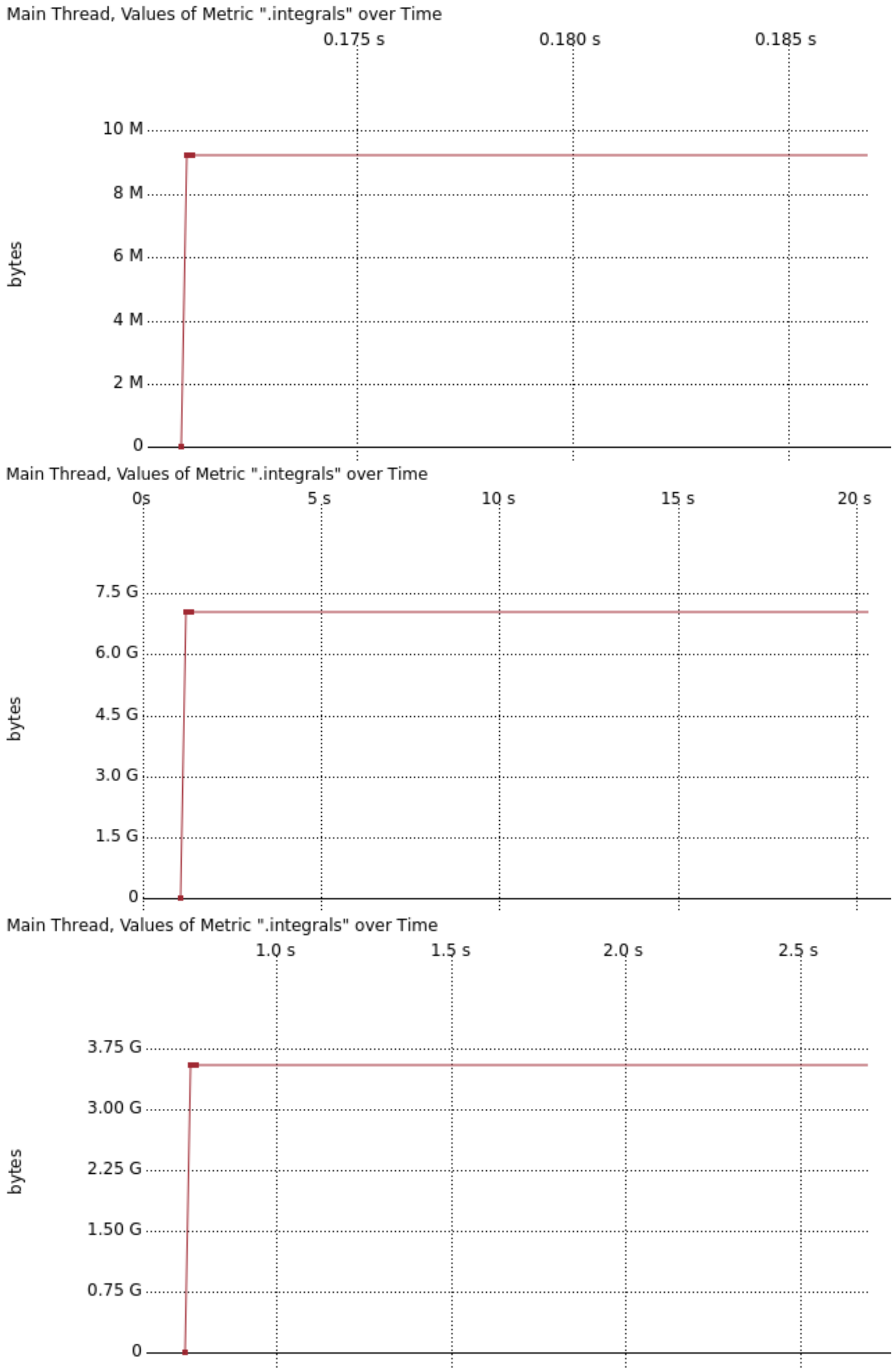


Figure 27: MPE for the Integrals entity. Top: Pd38-ar-OH configuration on 4 processes. Center: Pd38-ar-C1 configuration on 4 processes. Bottom: Pd38-ar-C1 configuration on 8 processes.

```
\begin{entity}{Overlap}
   type(rmatrix)  :: S
\end{entity}
\begin{entity}{DiagT}
   type(rdmatrix) ::
       t_diag
\end{entity}
...
call alloc(n, S, ..)
call alloc(n, t_diag)
```

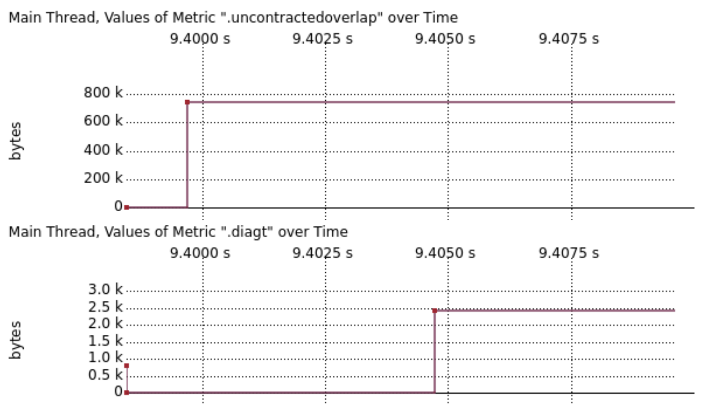


Figure 28: Left: Similar calls for variables of different types. Right: Correct tracing of the allocated memory considering the different sizes of the elements of the two matrices.

The example in figure 25 also points out the visual aid brought by the quantitative approach of MPE. The diagram displays three consecutive runs of the `do_one_block()` routine. The amount of memory occupied by the same entity is very different from run to run.

Another example of the MPE quantitative analysis is given in figure 28. There are two calls to the `alloc` interface, each for variables of different types. At a fast source code overview, one would expect the two entities to have comparable MPE values, given the same size `n` passed in the call. The traces on the right hand side, though, report a huge difference. With more than 700 KB, the overlap matrix uses by two orders of magnitude more space than the diagonal matrix. The explanation lies, on the different datastructure types of the variables. the `rdmatrix` is in fact only a one-dimensional array, while the `rmatrix` is a full two-dimensional matrix. The definitions of the custom types are usually grouped in a separate file, far away from the current place of usage and hence easy to oversee when reading the implementation of an algorithm. MPE quantitative approach supports thus a better code understanding.

8.4.3 *Result: qualitative assessment of memory usage - evolution*

Qualitative assessment refers in our context to the conclusions which can be drawn from a set of measurements with respect to the *quality* of the measured objects, or to their evolution in time with respect to a given metric.

The first and rather simplistic way in which MPE can be used to asses the evolution of the memory usage is the visual assessment of "hills and valleys" in the MPE diagram. A high frequency of ups and downs of the MPE values would signal possible suboptimal allocation and deallocation calls. Whether this can be improved, it strongly depends on the problem and implementation algorithm.

Another example includes higher-level information which can be acquired by combining MPE with DK Operation annotations. Figure 30 presents an overview of MPE values for some of the entities defined in section 8.3 in the `do_one_loop()` routine. The diagrams cover one execution of the routine.

One can see that the MPE profile over the entire routine execution is quite different for each entity. Some of them are being allocated memory from the very beginning,

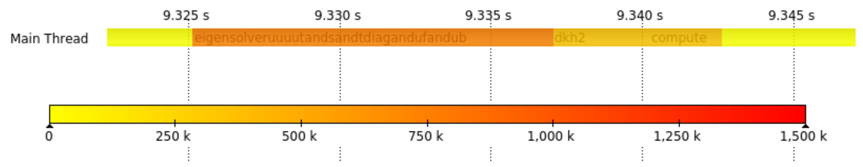


Figure 29: MPE overlay view on timeline. One execution of the `do_one_block()` routine with the color-mapped size of the *UFandUB* entity as an overlay.

like the uncontracted matrices, others are only being allocated at the very end, like the ContractedKin matrix. Some of them occupy memory for more than half of the execution time, while others are only short-lived allocations.

While this is already a valuable information which adds support in performance analysis, it could be extended even more by using DK Operations as well. One could annotate the important computation calls within `do_one_block()` as DK Operations, and link those operations with the corresponding entities on which the computations are actually applied. We show how this can be done later, in chapter 9. The visualisation would then include in the timeline along with the names of the operations being executed, also the entities involved in these operations. The MPE view of those specific entities would then reveal whether there were some too early allocations of the memory, or too late deallocations.

Vampir already provides a combined view of the timeline with an overlay of the values of a chosen metric. In figure 29 we chose to visualise MPE for UFandUB entity.An extension to this display mode picturing the actual used entities as well, would provide a powerful visual assessment method for the efficiency with which the allocated memory is being used.

8.5 CONCLUSIONS

In this chapter we have shown the benefits which the DK approach brings when it comes to analysing the performance of a specific class of applications, namely memory-intensive applications.

Quantum chemistry codes are a classic example of memory-intensive applications. The common computation method used in quantum chemistry is DFT, which following the Kohn-Sham approach results in a large amount of integrals having to be evaluated and stored. We used ParaGauss as our test application.

Based on the relevant DK-related statements with respect to the integrals computation, LaDoK annotations with DK Entities are easily inserted in the source code. For memory-related issues, one performs MPE measurements. MPE is the DK metric showing the amount of memory used at each point in time by a given entity.

A first analysis on a chosen block of code proved the correctness of the MPE measurements visualised with Vampir. Several implementation aspects which necessarily require dynamic runtime information can be traced by our DK-enhanced performance analysis tool.

The MPE results have a two-folded impact on performance analysis. On the one hand, the actual amount of memory occupied by entities is an aid in estimating the right input configurations which can be computed on given machines.

On the other hand, the evolution of the memory usage of an entity is a helpful information for optimisation analysis, especially when combined with other DK features, like the DK Operations.

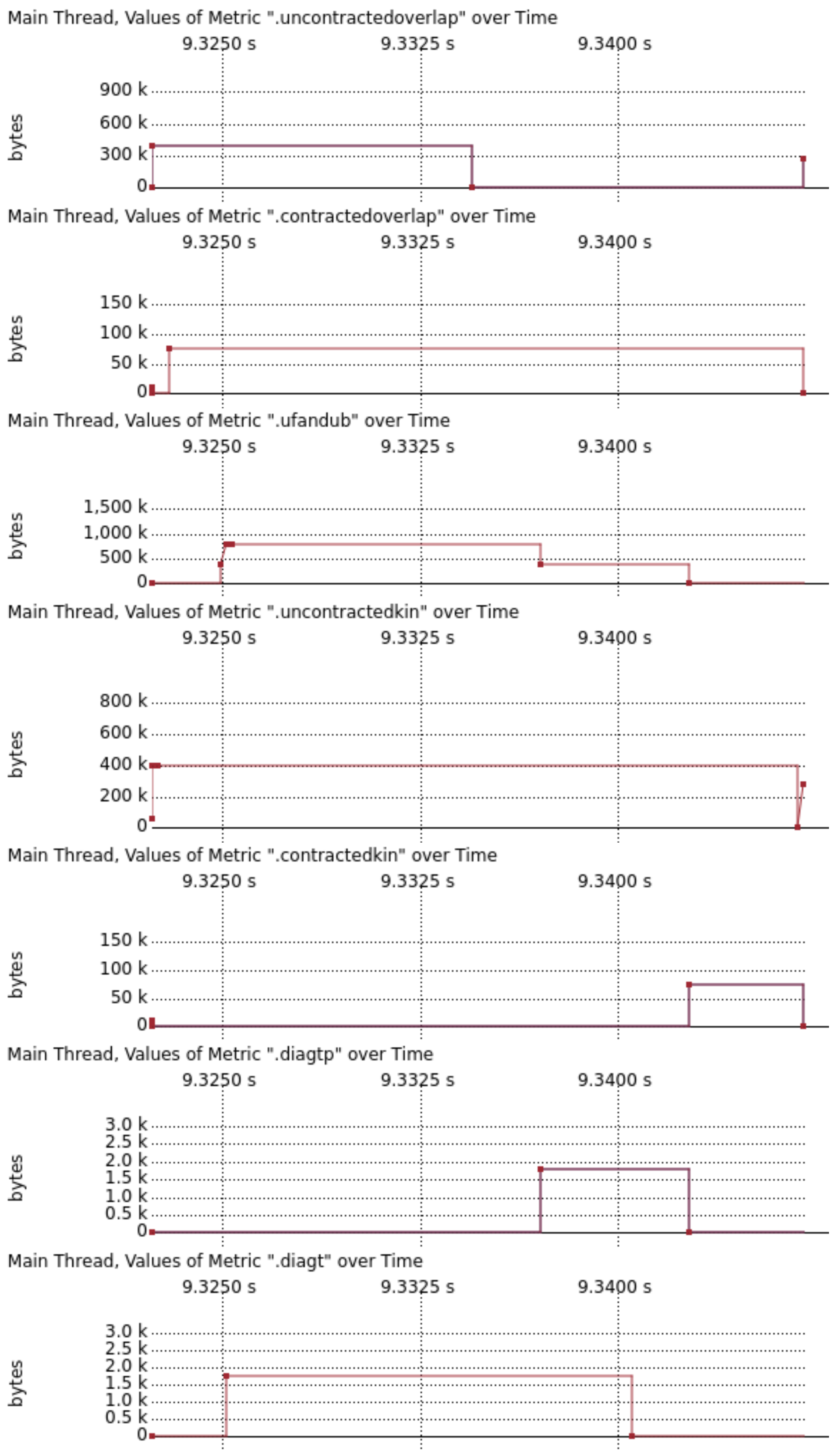


Figure 30: MPE behaviour for one run of the `do_one_block()` routine and several entities.

9 USE CASE: LOAD IMBALANCE PROBLEMS

One main issue which needs to be considered for parallel implementations in general, is the scalability of applications. In HPC, this is even more a crucial issue, as the main computation power arises from the many computation units which can and should be used for running applications.

Strong scalability refers to the performance of an application when maintaining the same input data and increasing the number of used computing units - cores or processors. *Weak scalability* allows increasing the size of the input data along with the number of computing units. Good scalability means, in the case of strong scalability for example, that the execution time of an application decreases proportionally with the increasing number of used processes or cores. The metric which is commonly used in this context is the *speedup*. The speedup gives the measure of how many times the execution of an application using `p` processes is faster than when using a single process. Ideally, the speedup is linear, or even superlinear for some cases[1].

A good speedup of an application running on `p` processes naturally implies that the work is well distributed among all `p` processes. One talks about *work or computation load* to refer to the amount of work which needs to be performed by each of the processes. A poor distribution of the work among processes results in uneven computation loads, usually labelled as *load imbalance*. This is a common performance bottleneck in HPC which can be caused by different factors, for example:

- the computation domain cannot be partitioned in even parts;
- the computation domain is too small;
- the computation tasks have very different computation costs;
- the computation tasks or domain dynamically changes with the computation;
- the costs of the computations cannot be predicted in forehand;
- there are data dependencies between the computation tasks, requiring synchronization points.

Without being exhaustive, the list above shows the variety and complexity of the situations which could lead to load imbalance problems. Solving such a bottleneck is in most cases not a trivial task and it commonly implies a good understanding of the implementation and the application domain itself. This is why our Domain Knowledge approach has a good potential for supporting the developers in dealing with this type of bottlenecks.

In the remainder of this chapter we present the dynamic load balance algorithm implemented within ParaGauss, the quantum chemistry application which we introduced in chapter 8. We then highlight the relevant Domain Knowledge for the load balancing process, followed by the corresponding LaDoK annotations based on the

1 For example, when the data needed for computations starts fitting inside the cache memory.

pseudocode of the algorithm. In the end we show which metrics and information could be obtained with the given annotations and we conclude with the benefits which DK brings in the current use case.

9.1 LOAD BALANCING IN PARAGAUSS

In section 8.1 we presented ParaGauss and the basics of DFT, which is the most common approach for quantum chemistry.

One important step in the implementations of DFT is the evaluation of the two-electron integrals, equation 17. The importance of the these integrals is given, from the chemical point of view, by the fact that they are used to compute the most part of the kinetic energy E_{kin}, equation 7, which again represents a significant amount within the total energy.

Moreover, from the computational point of view, the evaluation of the two-electron integrals enjoyed great attention, due to the fact that they are very computation expensive, the asymptotic complexity raising formally up to $\mathcal{O}(N^4)$.

There was a lot of work carried out for finding good parallelization schemes and corresponding efficient implementations for this step [35]. One common issue is the one referring to the storage of the necessary data for computations, in our case the density matrix P in equation 18, and the Kohn-Sham matrix $\hat{H}^{KS}$ in equation 15. For the purpose of this chapter though, we concentrate on the other commonly encountered issue: the load balancing of the computation of the integrals distributed among the application processes.

Finding a balanced work distribution for the two-electron integrals is not a trivial task. There are several factors which contribute to the complexity of this problem.

First of all, the number of integrals is very large, even for the medium sized electronic structure. For example, for a cluster of only 38 atoms of Palladium (Pd), there are more than 10^{12} integrals.

Secondly, the computational effort varies from one integral to the other, since it depends on the angular momentum l and the contraction[2] of the basis functions involved within that specific integral, as the equation 12 shows as well.

Thirdly, it is a common practice to perform batch computations of the integrals, grouping several of them into one single computation task. The criterion upon which the groups are built is very important for the general performance of the computation. Usually, the batches are built based on the shells of the atoms, or the angular momentum. The advantage is that a part of the primitive integrals[3] can be computed once and then used for all the integrals within the batch. The disadvantage is that this makes it even more difficult to exactly estimate the computation costs for each batch.

Lastly, when integrals screening is applied, some of the two-electron integrals are not computed at all. This is based on the principle that the interaction intensity between two electrons decays with the distance and thus, for a pair of electrons far away from each other, the integrals vanish as well.

In ParaGauss there are two load balancing approaches available:

2 Linear combination

3 The primitives are the basic components of the contractions of the basis functions.

1. **adjusted Round-robin distribution**: this is a static distribution method of the integral batches. The batches are labelled first with a cost class, based on the estimated computation effort. The Round-Robin method starts distributing batches from the computationally most expensive class, continuing in descending order until the least expensive one. The difference in cost between two consecutive classes is of an order of magnitude [76].

2. **work-stealing approach**: this is a dynamic enhancement of the Round-robin distribution. The batches are assigned initially similarly as in the approach above. Upon finishing evaluating its own set of batches, a processes is allowed to "steal" remaining work from the other processes [49].

9.2 RELEVANT DOMAIN KNOWLEDGE

In the context of scientific applications, *load imbalance* is a concept which, by default, belongs to the *computational field*, or, more precisely, to the performance optimization. The main concept behind is the *workload*, though, which necessarily resides within the specific *natural science field*. This is different, for example, from one of the other common application field for load balancing, namely the internet-based services. There, the workload is dynamically defined by user behaviour and software response time characteristics, whereas in our case, the workload is outlined by (strict) natural laws, expressed in algebraic structures or analytic representations. The field or domain within which the workload resides is of great importance when it comes to correctly treating load imbalance issues.

ParaGauss workload = batches of integrals.

For our current study case, the workload is given by the integrals batches which have to be evaluated by each of the processes. In ParaGauss such a batch is called a *quadrupel* and it is defined by a tuple of four indices: (ua_1, l_1, ua_2, l_2).

The two indices ua_1 and ua_2 identify the two (unique) atoms which are involved in the computation. In equation 17, these are the atoms on which the basis functions χ_ν and χ_μ and, respectively, χ_λ and χ_σ are centred on.

l_1 and l_2 define the angular momentum values for each of the two atoms. This further selects only a part from all integrals which could be evaluated for the two atoms. The mathematical expression is to be followed from equation 12. For every angular momentum l there are $2l + 1$ possible values for the magnetic number m. The number of values for the exponent of the gaussian radial function α is defined by the corresponding chosen set of basis functions.

An important observation here is that, theoretically, integrals could have been grouped based on any of their defining indices. The choice for the current quadrupel definition is based on the knowledge that those integrals "belong together" due the way basis functions are defined.

Another important aspect along with the workload discussed above is the so called *load balancer*. That is, besides the logic leading the workload distribution, it is important how this distribution is actually accomplished. For scientific applications, for example, it is important whether the data necessary for computation is already accessible by the process, or whether it needs to be received as part of the work package, bringing to the workload some extra time expenses.

No data fetching for batches, but scatter of computed results.

In ParaGauss, there are two aspects which need to be considered with respect to time costs due to data transfers for integrals batches computation:

- all data necessary for the evaluation of the integrals is already distributed to each process before beginning the Integrals step. Hence, there is no initial setup time which would need to be accounted for in the total time costs of a batch;
- at the end of each batch computation, the results are scattered to a subset of processes. This is related to another substep within the two-electron integrals computation, namely the evaluation of the fit functions f_k which are used in ParaGauss in the 3-center integrals as shown in equation 19.

9.3 LADOK ANNOTATIONS

In order to obtain DK-level views of the load balancing status of ParaGauss runs, the DK Operation concept can be used. There are three main keypoints which are of interest for the current study case.

9.3.1 *Distribution of quadrupels*

The distribution of the quadrupels is implemented in ParaGauss as a pure *control communication* operation. This means that the main goal of such an operation is to setup and trigger some action on the target process, usually a more time consuming one. Consequently, a control communication is also characterized by a relatively short message size, containing a relatively small set of control parameters.

In ParaGauss, the control parameters are the four indices which we described for a quadrupel. One can annotate such an operation as:

```
\begin{operation}{QuadrupelsDistribution}[type=commcontrol,
    param=q.ua1, param=q.ua2, param=q.l1, param=q.l2]
        ...
\end{operation}
```

9.3.2 *Computation of quadrupels*

The actual evaluation of the integrals within one quadrupel is a common computation operation. The data structures which are used in this step are actually DK Entities and can be linked to the operation correspondingly:

```
\begin{operation}{QuadrupelComputation}[type=computation,
    linkedentity=Hamiltonian, linkedentity=Overlapp, linkedentity=Basis]
        ...
\end{operation}
```

9.3.3 *Scattering of results*

Scattering the results of the computation is a *data communication*. Unlike the control communication for the quadrupels distribution where only some parameters were sent, here the transmitted messages contain data which is directly involved in the further computations. A consequence is that the amount of data is also considerably larger. In our case, the computed parts of the Hamiltonian matrix are scattered to a subset of the processes.

```
\begin{operation}{ResultsScattering}[type=commdata, param=H,
    checkop=SUM]
        ...
\end{operation}
```

9.4 DK TRACES

There are several ways in which DK can be used for performance analysis. In section 8.4 we showed the benefits of using a specialised DK metric, the MPE. In this current chapter, we concentrate more on how DK can augment with high-level or dynamic information the common measurements and visualisation provided by performance tools. We use as a context the investigation of the load imbalance issues.

9.4.1 *Work packages*

A common issue when analysing load imbalance situations is the *identification* of the exact work packages which were being processed by each of the working units. Given an MPI implementation, one could pin-point, for example, on the timeline view of the Vampir tool, which one is the problematic work package, either by being too short or too long in execution. The difficulty occurs in really identifying the characteristics or parameters which define this particular work package.

In the case of ParaGauss, we already explained that the work packages are completely defined by the indices in the quadrupels. In the timeline view, though, one could only see the MPI call from the master to that specific processes, together with the corresponding low-level information like the number of transmitted bytes, or the duration of the transfer.

The meaningful information can be provided via the DK approach. Annotating the communication operation like in section 9, instructs the measurement backend to also trace the values of the given variables and assign them to the corresponding execution of the operation. These values are then provided in the context view whenever an operation is selected in the timeline. This is a high-level dynamic information which DK provides for the identification of work packages.

9.4.2 *Detailed processing costs*

For the actual computation of the given work packages, one is usually interested in the execution time. This performance metric might be combined in some cases with the analysis of the hardware counters values, like FLOPs or cache misses. The latter, though, are hard to map to the actual source code and thus the root cause for any observed metrics or behaviour is hard to detect.

DK offers the possibility to add an extra level of information in this case as well. The DK Operations can be linked to DK Entities, as we showed above. The timeline view can thus be extended to provide hints related to the actual targeted data structures within the current execution segment.

For the computation of a quadrupel, for example, one can interrogate not only the elapsed time, but also the total memory needed for the entities involved in the current execution step. This high-level information is a plus for interpreting the commonly provided low-level information. For example, a high cache miss rate for a relatively small entity could mean that the storage pattern does not fit the computation/access pattern. Given the small size of the entity, one solution might also be the partial replication of the stored data.

9.4.3 *Results checks*

In scientific applications it is sometime the case that the current position or status within the entire simulation or computation processes could be determined by means of some representative value like, for example, the sum over the elements of an array.

For a scattering process like the one described for the results of the quadrupel computations in ParaGauss, common performance tools would only be able to provide information regarding the size of the transmitted data. Using the DK approach, one is able to gain insight in the content of the transmitted data as well.

The main characteristic of *data communication*, as opposed to *control communication*, is that the amount of transmitted data is large. It is impossible to trace the content of a data communication like we proceeded for control communication. We rather instruct the DK-enhanced framework to apply and track only the result of a given check operation, for example, only the sum over the elements of an array. This value can then be visualised as a property of the annotated operation.

9.5 DK PROFILES

One key mechanism for performance analysis is represented by the profiling approach. Profiling offers a summarisation over the entire execution of the measured values. Besides raw data display for gathered metrics measurements, profiling also uses histograms and bar charts.

9.5.1 *Workpackage grouping per execution time*

One example of a histogram in Vampir, is the *Message summary* view, which displays the total number of messages per message size, as seen in figure 31. A similar view could be generated for DK high-level information. In our context we would be interested to see the number of quadrupels for given computation time intervals. As explained in section 9.1, the quadrupels distribution algorithm in ParaGauss is based on the classes of quadrupels defined by the estimated computation time. The estimations could be checked using the view of the actual execution times and number of quadrupels. This is basically an adaptation of the CEC metric which we described in section 4.4. There, the workpackages were considered identical, while here they have different execution times.

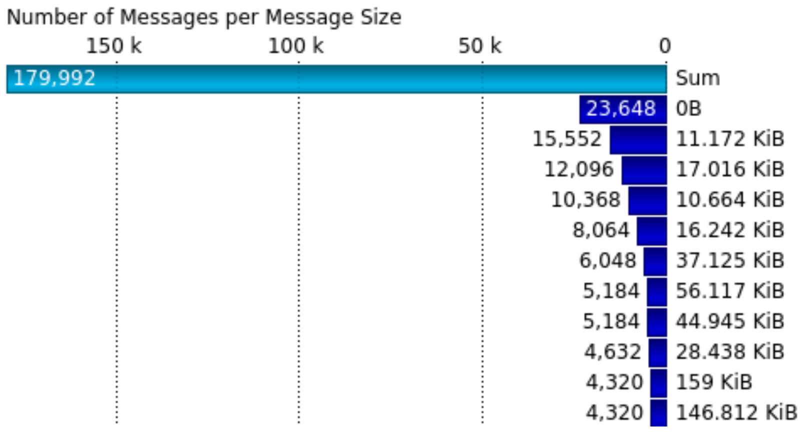


Figure 31: Message Summary view in Vampir for the default Score-P tracefiles.

LaDoK already provides support for such information and the necessary profiling measurements could be easily integrated in the DK-enhanced framework. It is sufficient to add to the definition of the computation operation in section 9.3.2, the `param=q.ua1, param=q.l1, ...` list of parameters already given for the control communication operation in section 9.3.1. The measurements library would then trace the quadrupel for every execution of the operation.

A minimal extension of the Vampir tool would be needed in order to generate histograms with the number of executions of the operation for given execution time intervals. If the runs were stacked on each other to build the bars in the graph, then one could also select each run and inspect the corresponding parameters registered with the annotation we presented above.

9.5.2 *Execution time per entity type*

One could even get one step further in customizing the views in Vampir . When analysing the distribution method of the quadrupels in ParaGauss, for example, one could use a histogram of the total execution time per atom type, or per unique atom. We consider an adapted annotation version of the unique atoms from chapter 8 as entityset:

```
module unique_atom_module
type, public ::  unique_atom_type
   character*12 :: name
   ...
end type unique_atom_type
...
\begin{entityset}{UniqueAtom}[typevar=unique_atoms, typefield=name]
   type(unique_atom_type), pointer, public :: unique_atoms(:)
   ...
\end{entityset}
...
end module unique_atom_module

...
```

```
\begin{operation}{QuadrupelComputation}[type=computation,
    linkedentityset=UniqueAtom, idx=q.ua1, idx=q.ua2]
                ...
\end{operation}
```

The `typevar` and `typefield` attributes represent the variable and eventually the field in the custom type which define the type of an entity, or of each of the entities in the current entityset. This will be the grouping parameter for the histogram view in Vampir. If there is more than one field needed to identify the type, one can add several `typevar/typefield` attributes.

The `linkedentityset` attribute together with the `idx` attribute instruct the DK-enhanced framework, that the current operation is running computations for the two entities identified by the indices `q.ua1` and `q.ua2` in the entityset `UniqueAtom`.

The tracing library should record this information with respect to the current operation, as well as the value of the `typevar/typefield` for the currently used entities. All this information should be sufficient for a visualisation tool, for example Vampir with some small extensions, to produce afterwards the histogram with the total execution time per unique atom.

9.5.3 *Execution time per operations group*

We have seen already that for ParaGauss, the time for processing one quadrupel does not include only the effective computation time, but it also includes the control and data communication before and after the computation. For a correct load balancing analysis one usually has to consider several operations which are executed for the same workpackage.

Common profilers are able to identify several calls of the same routine and display, for example, the total execution time. But they are not able to group single calls of different routines which work on the same dataset.

The annotations presented above with the identification possibility for DK Entities and the connection of DK Operations to those specific DK Entities, introduces the possibility for profiling tools to group routines, or, better, DK Operations, on the basis of the datasets they use.

As an example, the *Process Summary* view of Vampir, figure 32, displays the accumulated execution time per routine and per process. It is hard to identify the actual time spent for computing a given dataset, since the routines are ordered based on the total execution time. Moreover, there is no information as whether all calls of a given routine are within the same callpath. Some general routines might be called from very different context. Given our DK enhancement, correct execution times would be reported for the computation of single workpackages. In our load balancing use case, correct execution times will include the computation operation of a quadrupel, but also the quadrupels distribution and the results scattering steps.

9.6 CONCLUSIONS

Load imbalance problems are difficult to handle with common performance analysis approaches. The factors which come into play in such contexts are tightly connected

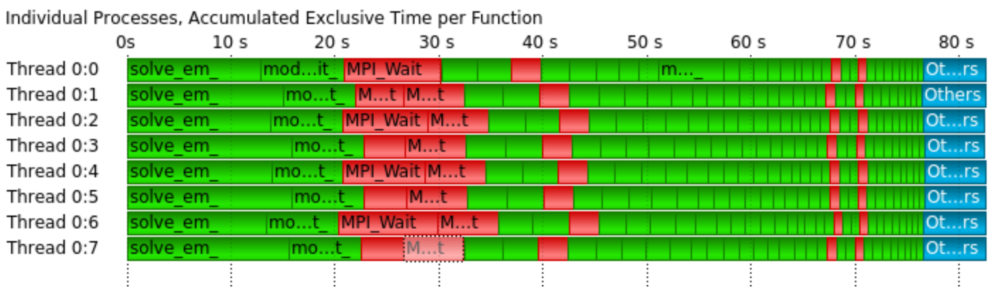


Figure 32: Process Summary view in Vampir for the default Score-P tracefiles.

to the scientific domain of the application. This is why, the low-level performance information needs to be enhanced with DK specific higher-level information.

Using different types of DK Operations, one can better guide the performance measurement and analysis process. The resulting DK specific information can be used on top of the common profiling methods, improving their usability.

DK based visualisation enhancements can be used for both tracing and profiling views. Timelines and histograms can be adapted to include the DK specific information, providing better support for performance analysis.

10 FURTHER USE CASES

In this chapter we present three further use cases for DK-enhanced performance analysis. We showcase some features of LaDoK which were not covered in the previous use cases, but also highlight the importance of already presented ones for some classic application types. We start with an example of how *linked operations and entities* can help in profiling applications based on multigrid solvers. We then highlight the practicality of the *DK Phases* within the context of performance dynamics. Finally, we look at the example used in the related work of SEAA [73] and show how indexed *entitiysets* and custom attributes can yield the same results as in the cited work. Further deployment examples can also be found in [7].

10.1 MULTIGRID-BASED SIMULATIONS

Multigrid methods are prevalent in applications using PDEs (Partial Differential Equations), as they are suited for solving $Ax = b$ sparse linear systems [86]. Such systems are typically generated in the discretization of PDEs. Multigrid methods are able to solve such systems in $\mathcal{O}(N)$ time and storage space, where N is the number of the unknowns.

Multigrid methods use a sequence of successively refined grids. The goal is to solve the system discretized on the finest grid. In order to do so, one approximates the error of an initial guess for this solution. The approximations are done recursively in a fixed number of steps, from the finest to the coarsest grid. Using the Fourier transformation concepts, we say that the smooth part of the error is approximated on the coarser grid, while the non-smooth part is solved with a basic iterative method on the fine grid [86]. A typical pseudocode for the main recursive function is given in listing 21.

Using DK for a multigrid implementation in a specific application domain is tempting from many points of view. Nevertheless, for the current study we would like to drop our attention only on a single aspect, namely the recursive call pattern of the main function.

One particular challenge for performance analysis in this context is to clearly identify the current grid level. One could use, of course, either a callpath view or a process timeline view as provided by Vampir in order to count the number of recursive calls and thus identify the current refinement. This might become a tedious job, since there are multigrid implementations which go beyond the simple "V" cycle, producing, for example, "W", "M" or "F" cycle types.

What is actually needed, is a means to characterise the iteration process. We defined in section 4.4, along with the other DK metrics, also the *Iterometers*. Since most scientific implementations involve some kind of a loop or iteration, it is clearly necessary to also have a means to characterize such processes.

For multigrid, there is no explicit loop, but one could consider the cycles through the grids as successive iterations towards the solution.

```
function MultiGrid(..., size)

        smooth()
        residual()
        restrict()

        if (coarsest grid){
                /* solve error equation */
        } else {
                MultiGrid(.., size_next)
        }

        interpolate()
        correction()
        smooth()
end function
```

Listing 21: Typical pseudocode for the main function of a multigrid method.

The LaDoK annotation to be added to the pseudocode above is rather simple:

```
\begin{operation}{MultigridOneLevel}[iterometervar=size]
        function MultiGrid(..., size)
        ...
        end function
\end{operation}
```

We defined the entire function as one operation and we added the `iterometervar` attribute to signal the variable which identifies or characterizes the current execution of the operation.

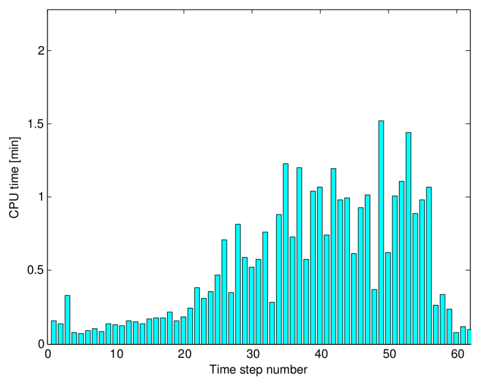
Extending a DK-enhanced framework to support this feature is straightforward. For the visualisation one could use either a timeline based display, adding to the operation also the information regarding the iterometer, or a profiling view style, with aggregated measurements per grid level.

10.2 APPLICATIONS WITH DYNAMIC PERFORMANCE

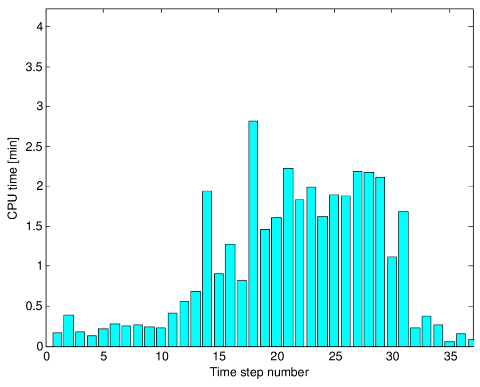
Performance dynamics refers to the variations over time of the performance of applications. Reasons for such variations come from multiple factors and the variations themselves can be reflected on multiple dimensions of performance.

On the one side, the reasons might be related to the actual hardware on which the application is running, from the network limitations to the cache memory architecture, or even some integrated energy saving approaches. On the other side, the application itself might include different execution phases, like the classic input-compute-output, or some adaptive algorithms, like mesh refinement or multigrids.

The variations could be observed on any of the measured performance metrics. For example, the CPU Time for single interation steps, the execution time of particular MPI calls, or the number of FLOPS per memory access.



(a) Time step adaptation strategy 1.



(b) Time step adaptation strategy 2.

Figure 33: Performance of INDEED, measured as CPU time per iteration.

#	Iteration diapason	Sim.time diapason [sim. time units]	Cumulative CPU time [min.]	Simulation speed [sim. time units / min.]
1	[1,25]	(0,23.5]	4.72	5
2	[26,55]	(23.5,51.22]	25.77	1.08
3	[56,62]	(51.22,56.81]	2.19	2.56

Figure 34: Performance properties reported by the automatic analysis method of the time step adaptation strategy 1 [50].

In the HPC context, where the number of processes executing the application tends to be very large, and also the total execution time of applications tends to be very long, analysing performance dynamics easily becomes a tedious task for human users. Oleynik [50] proposes an automatic approach for detecting and analysing performance dynamics, providing results for some real-world use cases. We shortly refer here to one of these use cases and highlight the benefits which DK could bring.

INDEED [28] is a simulation application, based on highly accurate finite elements implementations for the computation of sheet metal forming, developed by GNS mbH. It is common for simulations in the mechanical computation field to use adaptive methods for more accurate computations, based on the evolution of the simulated dataset. INDEED uses different adaptation strategies for the time step size used in solving some stiffness systems. Large time steps imply less iterations, but each iteration is computation expensive. Small time steps result in more iterations, but with less CPU time per iteration. The goal of the adaptation strategies is to find the sequence of time step widths which generate the shortest overall execution time.

The performance of the code, expressed as CPU time per iteration, exposes some clear dynamics, as can be seen in the two histograms in figure 33.

The automatic analysis of the performance dynamics detects three convergence phases, as well [50]. Figure 34 shows the three different properties detected for the first adaptation strategy.

While this is already valuable information, one could reach even more insight into the problem, if some higher-level, domain specific information were provided as well. We know, for example, that the performance is influenced by the chosen time step

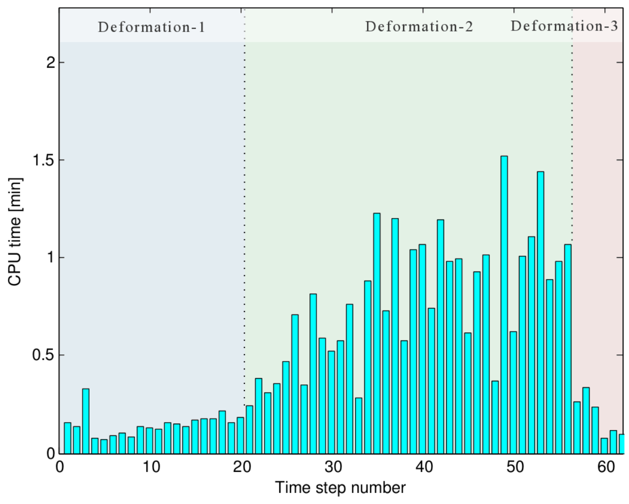


Figure 35: Hypothetical case highlighting DK Phases for strategy 1 in figure 33a.

and that the time steps are chosen based on the current state of the sheet of metal which is being simulated. The automatic analysis already discusses about *convergence phases*. It is indeed, a straightforward question which comes into mind: *In which phase of the simulation is the current iteration taking place?*

Domain Knowledge brings this one detail into the picture with only minimal effort. Using the DK Phase concept and the corresponding LaDoK annotation, one could mark in the source code the starting point of a new phase of the simulation. In case of INDEED, this should be found in the logic of the adaptation strategy itself. There the current status is checked and a decision on the width of the next time step is taken. The LaDoK annotation is very simple, like this:

```
\phase{Deformation-N}
...
```

but the benefits would make a difference.

On the one hand, the histograms for the manual analysis could be enriched with the visualisation of the phase as well. When the complexity of the simulated data increases and the number of steps gets larger, such high-level information could prove very handy for easier analysis. Figure 35 presents a hypothetical case.

Of course, one could get further and add some DK Operation definitions together with some well chosen attributes, like the `param/checkop` combination we described in section 9.3.3. We limit ourselves to the DK Phases now, in order to stress their importance as well.

Some tools, like the Score-P backend measurement system also offer the possibility to track user-defined markers within the execution and Vampir offers a nice support for handling and visualising such markers. This is a nice feature when it comes to identifying the place where some particular event occurred. The DK Phases are similar in this sense with the markers, but they also go beyond simply marking some events. For example, a DK Phase sets the forthcoming execution of the application under this particular *modus* until another phase is met.

On the other hand, the automatic dynamics analysis could also benefit from such simple DK enhancement. One could use, for example, targeted performance measurements. Especially when it comes to long running simulations, the amount of gathered data increases a lot. Restricting measurements collection only for relevant application phases would reduce the overall amount of data. Besides the amount of collected data, also the quality of the automatic analysis might improve. Given that the target is set on the dynamics of the performance, narrower measurement intervals would result in more precise results.

10.3 PARTICLE SIMULATOR

Let us consider now the hypothetical particles simulation code in listing 5, which we presented in detail when discussing about the SEAA approach in section 3.6.1. The challenge formulated by the author with respect to this code snippet was to provide performance metrics per faces of the cube, instead of single particles.

This is a typical case of DK: the computation taking part in the `ProcessParticle()` function is connected semantically with the `face` variable from the `GenerateParticles()` function. In the same time, there is no syntactical expression of this dependency at computing time, no means to correctly assign the measured performance metrics by common profiling techniques.

The relevant Domain Knowledge in this case is the very fact that the particles are grouped by the faces of the cube. The necessary LaDoK annotations are given in listing 22:

1. annotation of the array of particles as a dynamic entityset;
2. annotation of the particles generation action with an operation of initialization type and with a link to the entityset above, as well as with a type variable;
3. annotation of the actual processing action with a computation operation.

As one can see, LaDoK already provides the appropriate syntax to support all needed annotations. From the semantic point of view, the entity and entityset implementations should be extended to support the dynamic initialization of their attributes. This can be easily achieved by adding in the instrumentation logic the corresponding processing of operations of type `initialization`.

An important detail is given in this context by the `typevar` attribute. We already discussed about its potentials for performance profiling in section 10. The current study case is very similar to the one presented there. The *time per **entity type*** is replaced here with *time per **face***. In essence though, it is exactly the same situation, as in our annotations we are defining `face` as the variable giving the *type* of the entities.

While the first two annotations in the list above are mainly about the LaDoK framework capabilities, the last annotation is the responsibility of the DK-enhanced profiling framework. This is also a straightforward extension for the current implementation and instrumentation logic. Since the `ProcessFace` is an operation of type `computation`, the execution time should be registered as computation time for all involved entities. Please note, that the dynamic information approach implemented through the `ldk_flags` in the LaDoK framework, already provides the necessary information at runtime for correctly identifying the corresponding DK Entity for the local argument `p`.

```
\begin{entityset}{Particles}[type=dynamic]
   Particle* P[MAX]; /* Array of particles */
\end{entityset}

int GenerateParticles() {
   /* distribute particles over all surfaces of the cube */
   for (int face=0, last=0; face < 6; face++){
      int particles_on_this_face = ... face ; /* particles on this face */
      for (int i=last; i < particles_on_this_face; i++) {
       \begin{operation}{GenerateParticles}[type=initialization,
           linkedentityset=Particles, idx=i, typevar=face]
         P[i] = ... f(face); /* properties of each particle are some function f
             of face */
       \end{operation}
      }
      last+= particles_on_this_face; /* increment the position of the last
          particle */
   }
}

\begin{operation}{ProcessFace}[type=computation]
int ProcessParticle(Particle *p){
    /* perform some computation on p */
}
\end{operation}

int main() {
   GenerateParticles(); /* create a list of particles */
   for (int i = 0; i < N; i++)
      ProcessParticle(P[i]); /* iterates over the list */
}
```

Listing 22: A hypothetical particles simulation [73] with corresponding LaDoK annotations.

Finally, the visualisation and analysis of the measurements depends on the Vampir capabilities, or any other tool used in this step. One example would be to use a histogram to display the total execution time per face. This is similar to the total execution time per entity type, as we described already in section 9.5.2.

As compared to the SEAA aproach, DK has first of all the advantage of providing LaDoK as a standard, yet flexible means to annotate the code. Furthermore, due to splitting the language from the actual performance framework, the measurement and analysis potential is increased considerably. One is not limited to time measurements anymore, but can also run other performance metrics measurements if needed.

11 SUMMARY AND OUTLOOK

Most computational sciences share the same historical root. Computational scientists from today are the savants and philosophers from centuries ago, trying to discover and understand the rules of nature. Computing centres, and, particularly, HPC centres are today's experimental labs. Natural and computational scientists need an adequate set of tools and methods in order to be able to perform their experiments in this new laboratory.

In this thesis we investigated the formulation and applicability of Domain Knowledge (DK), a new approach which targets the narrowing of the gap between the natural scientists and the computer scientists. Especially in the HPC environment, such gaps build easily due to the complexity of both implemented applications, as well as underlying hardware and systems. The main focus of this work was set particularly on the gap between the scientific applications on the one side and the performance and measurements tools on the other side.

The formulation of the DK approach is based on the fact that scientific applications are computer-based implementations of mathematical formulations of the natural laws.

There are three main steps within DK: the identification of the relevant knowledge, the explicit formulations of knowledge and, finally, their application and usage. Each of these steps deals with the three DK concepts: DK Entities, DK Operations and DK Phases.

The identification step is a rather subjective process, depending not only on the actual application or scientific domain, but more on the scientist itself. A suited training of the scientists should enable them to perform this first step. Throughout this thesis we provide many examples of identifications of relevant domain knowledge for different use cases.

For the explicit formulations of the knowledge, we propose the Language for Domain Knowledge (LaDoK). LaDoK is an annotation language meant for the enhancement of source codes with explicit DK elements. LaDoK has a familiar syntax similar to $\LaTeX$ and is flexible enough to support any particular scientific domain. Moreover, the concise syntax reduces the learning slope and stimulates personalized usage. In performance analysis terms, LaDoK annotations resemble the manual instrumentations necessary for later performance measurements.

The semantics of LaDoK is strongly connected to the three DK concepts above. It supports the declaration of such concepts directly in the source code, including a series of attributes which can be specified for each of them. An important feature is the ability to set links between DK Objects. In particular, DK Operations can be linked to corresponding DK Entities.

An outcome of this work is also a LaDoK framework, able to process Fortran source codes with LaDoK annotations and build a corresponding database of DK Objects. The latter can be used afterwards in any DK-enhanced advanced programming tool. Of special importance is the implemented support for dynamic runtime information, which can be combined with the static information provided by users through annotations. By means of this feature, DK Entites can be identified anywhere in the execution of an application.

For the third step in the approach, namely the application and usage of the formulated knowledge, we focused on the particular context of performance measurements and analysis in HPC. DK emerges in this area by means of the DK metrics and counters. In this thesis we defined Memory per Entity (MPE), Computed Entities Count (CEC) and Iterometers.

In order to prove the usability of DK in this context, we developed a DK-enhanced performance analysis framework, based on technologies and tools like Periscope, OTF2 and Vampir. The framework supports the entire performance analysis cycle, from manual and automatic instrumentation, to measurements, visualisation and analysis.

In the last part of this thesis we looked at several different use cases, showing how the DK approach can be deployed and which results can be achieved with the two frameworks which we developed.

The main usability studies are based on a quantum chemistry application, ParaGauss, which is used to simulate and analyse properties of different electronic systems. There are two main results with respect to the deployment of DK in analysing the performance of this code. First of all, the MPE metric is successfully traced by our DK-enhanced framework and it offers an enriched visualisation and perception of the memory usage within code runs. The only needed user-input is the declaration of the DK Entities within the source code.

Secondly, the load balancing strategy used in ParaGauss can be better traced with the help of the DK techniques. In particular, the dynamic information tracing feature of LaDoK and the linked DK Operations and DK Entities supply great performance analysis potential.

Additional features of the DK deployment in performance analysis cycle were analysed for other application classes as well. For example, one could enhance the analysis of the performance dynamics, or extend the support for the multigrid methods analysis with runtime high-level information.

Overall, the seamless integration of DK with other technologies promises key enhancements which improve the end results.

11.1 FUTURE WORK

If the interaction with the common gadgets evolved from signal buttons to natural swipe gestures, why not letting the supercomputers follow the same path and make it possible for the natural scientists to interact with them using their everyday domain specific terms?

As any other research, this thesis opens doors for a series of further possible developments and investigation tracks.

First of all, there are several future work directions related to the performance measurements and analysis field. One important aspect is the integration of the data-centric profiling with the DK approach. For example, variable accesses could be traced for DK Operations which have linked DK Entities. This would bring a plus of insight regarding the behaviour of those DK Objects and possible optimizations could be unveiled.

Another direction to follow is the extension and adjustment of the visualisation possibilities for DK information. We already presented some suggestions with respect to profiling diagrams within Vampir. Other ideas are directed towards handling DK Objects through summary views and activating direct access from these elements to corresponding hot-spots in the performance measurements.

Besides graphical visualisation, command-line tools, like those already provided for OTF2 formatted files are also of interest. Computational scientists are usually accustomed with analysing text-based results, usually tables and lists of statistics. Some DK Metrics like MPE or Iterometers qualify for this type of visualisation.

Besides the performance analysis field, there are also other areas for which further developments can be foreseen. First of all, in the closely related performance tuning field, LaDoK annotations of the code could be used as hints by the autotuning tools. One could start, for example, with filtering tuning regions by the declared DK Phases.

Another practical future work would be in the Software Engineering (SE) area. Especially the development of the scientific implementations could benefit from a DK-enhanced SE cycle. One big challenges for these codes is the very long lifetime of their development process. More precisely, implementations are continuously being extended with new scientific methods and results and the source codes tend to have unclear SE designs or structures. Using LaDoK annotations and developing a corresponding tool for generating DK-views of the code structure could ease the overall SE experience.

BIBLIOGRAPHY

[1] Krste Asanovic et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec. 2006. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[2] Peter W. Atkins and Julio de Paula. *Kurzlehrbuch Physikalische Chemie*. 4., vollständig überarbeitete Auflage. John Wiley & Sons Ltd., 2008.

[3] Er A Auer et al. "Automatic code generation for many-body electronic structure methods: The Tensor Contraction Engine." In: *Molecular Physics* (2006).

[4] R J Bartlett et al. *SIAL Programmer Guide*. Tech. rep. 2011, pp. 1–103. URL: http://www.qtp.ufl.edu/ACES/downloads/SIALProgrammerGuide.pdf(accessedon8Dec.2013).

[5] Gerald Baumgartner et al. "A high-level approach to synthesis of high-performance codes for quantum chemistry." In: *Supercomputing Conference*. 2002, pp. 1–10. DOI: 10.1145/762761.762818.

[6] Th. Belling et al. "ParaGauss: A Density Functional Approach to Quantum Chemistry on Parallel Computers." English. In: *Scientific Computing in Chemical Engineering II*. Ed. by Frerich Keil et al. Springer Berlin Heidelberg, 1999, pp. 66–73. ISBN: 978-3-642-64295-1. DOI: 10.1007/978-3-642-60185-9_6. URL: http://dx.doi.org/10.1007/978-3-642-60185-9_6.

[7] Anca Berariu. "Fostering Domain Knowledge for Improved Performance Analysis." In: *Concurrency and Computation: Practice and Experience* (Submitted).

[8] H Beyer and H B K Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann, 1998. ISBN: 9781558604117. URL: http://books.google.de/books?id=sVKuMvaFzjQC.

[9] Doug Brown, John Levine, and Tony Mason. *lex & yacc*. 2nd Edition. O'Reilly Media, 1992. ISBN: 978-1-56592-000-2.

[10] Marc Casas et al. "Automatic phase detection of MPI applications." In: *Proceedings of the 14th Conference on Parallel Computing (ParCo 2007)*. Aachen and Juelich, 2007.

[11] George Crabtree et al. *Computational Materials Science and Chemistry: Accelerating Discovery and Innovation through Simulation-Based Engineering and Science*. Technical Report. Accessed on 03.08.2012. U.S. DOE, Office of Science, July 2010. URL: http://science.energy.gov/$\sim$/media/bes/pdf/reports/files/cmsc_rpt.pdf.

[12] Christopher J. Cramer. *Essentials of computational chemistry: theories and models*. 2nd edition. John Wiley & Sons Ltd., 2004.

[13] Juan Carlos Cuevas. *Introduction to Density Functional Theory, Institut für Theoretische Festkorperphysik, Universität Karlsruhe*. URL: www-tfp.physik.uni-karlsruhe.de/cuevas.

[14] Erik Deumens et al. "Software design of ACES III with the super instruction architecture." In: *Wiley Interdisciplinary Reviews: Computational Molecular Science* 1.6 (2011), pp. 895–901. ISSN: 1759-0884. DOI: 10.1002/wcms.77. URL: http://dx.doi.org/10.1002/wcms.77.

[15] Arie van Deursen, Paul Klint, and Joost Visser. "Domain-specific languages: an annotated bibliography." In: *SIGPLAN Not.* 35.6 (June 2000), pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035. URL: http://doi.acm.org/10.1145/352029.352035.

[16] *Dyninst API*. Accessed on 20.09.2015. URL: http://www.dyninst.org/dyninst.

[17] Bernd Mohr Felix Wolf. *Specifying Performance Properties Using Compound Runtime Events*. Technical Report. Forschungszentrum Jülich GmbH, 2000.

[18] Michael Gerndt, Karl Fürlinger, and Edmond Kereku. "Periscope: Advanced techniques for performance analysis." In: *PARCO* (2005), pp. 15–26.

[19] M. Gerndt and M. Ott. "Automatic Performance Analysis with Periscope." In: *Concurr. Comput. : Pract. Exper.* 22.6 (Apr. 2010), pp. 736–748. ISSN: 1532-0626. DOI: 10.1002/cpe.v22:6. URL: http://dx.doi.org/10.1002/cpe.v22:6.

[20] *GNU Emacs*. Accessed on 14.08.2015. URL: https://www.gnu.org/software/emacs/.

[21] Samuel Zev Guyer. "Incorporating Domain-Specific Information into the Compilation Process." PhD thesis. 2003.

[22] S Z Guyer and C Lin. "Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries." In: *Proceedings of the IEEE* 93.2 (2005), pp. 342–357. ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.840489.

[23] Richard Hamming. *Numerical Methods for Scientists and Engineers*. Second edition. McGraw-Hill, Inc., New York, 1973.

[24] M T Heath and J A Etheridge. "Visualizing the performance of parallel programs." In: *Software, IEEE* 8.5 (), pp. 29–39. ISSN: 0740-7459.

[25] Dimitri van Heesch. *Doxygen Homepage*. Date accessed: 11.12.2012. URL: http://www.stack.nl/~dimitri/doxygen/.

[26] T. Helgaker, P. JÃžrgensen, and J. Olsen. *Molecular Electronic-Structure Theory*. John Wiley & Sons Ltd., 2000.

[27] Michael A. Heroux, Padma Raghavan, and Horst D. Simon. *Parallel Processing for Scientific Computing*. Society for Industrial and Applied Mathematics, 2007.

[28] *Highly accurate finite element simulation for sheet metal forming*. Accessed on 23.08.2015. URL: http://gns-mbh.com/indeed.html.

[29] A Hinchliffe. *Chemical Modelling: Applications and Theory*. Specialist Periodical Reports v. 4. Royal Society of Chemistry, 2006. ISBN: 9780854042432. URL: http://books.google.de/books?id=Hhxm8nSWI98C.

[30] P. Hohenberg and W. Kohn. "Inhomogeneous Electron Gas." In: *Physical Review* 136 (Nov. 1964), pp. 864–871. DOI: 10.1103/PhysRev.136.B864.

[31] *HPCToolkit*. Accessed on 20.09.2015. URL: http://hpctoolkit.org/.

[32] *Intel Trace Analyzer and Collector*. Accessed on 20.09.2015. URL: https://software.intel.com/en-us/intel-trace-analyzer.

[33] R . Bruce Irvin. "Performance Measurement Tools for High-Level Parallel Programming Languages." PhD thesis. 1995.

[34] R Bruce Irvin and Barton P Miller. "Mapping Performance Data for High-Level and Data Views of Parallel Program Performance." In: *International Conference on Supercomputing*. 1996.

[35] Wibe A. de Jong et al. "Utilizing high performance computing for chemistry: parallel computational chemistry." In: *Phys. Chem. Chem. Phys.* 12 (26 2010), pp. 6896–6920. DOI: 10.1039/C002859B. URL: http://dx.doi.org/10.1039/C002859B.

[36] Ken Kennedy et al. "Telescoping Languages : A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries." In: *Journal of Parallel and Distributed Computing* 61 (2001), pp. 1803–1826.

[37] Donald E Knuth. "Literate Programming." In: *The Computer Journal* 27.2 (1984). Ed. by A Ralston, E D Reilly, and DEditors Hemmendinger, pp. 97–111. URL: http://comjnl.oupjournals.org/cgi/doi/10.1093/comjnl/27.2.97.

[38] W. Kohn and L. J. Sham. "Self-Consistent Equations Including Exchange and Correlation Effects." In: *Physical Review* 140 (Nov. 1965), pp. 1133–1138. DOI: 10.1103/PhysRev.140.A1133.

[39] Collaborators Karol Kowalski et al. *A Quantum Chemistry Domain-Specific Language for Heterogeneous Clusters*. Tech. rep. 2012, pp. 1–22.

[40] Rubin Landau, Manuel José Paéz, and Cristian Bordeianu. *A Survey of Computational Physics. Introductory Computational Science.* Princeton University Press, 2012.

[41] V Lotrich et al. "Parallel implementation of electronic structure energy, gradient, and Hessian calculations." In: *The Journal of chemical physics* 128.19 (May 2008), p. 194104. ISSN: 0021-9606. DOI: 10.1063/1.2920482. URL: http://www.ncbi.nlm.nih.gov/pubmed/18500853.

[42] A D Malony, S S Shende, and A Morris. "Phase-based parallel performance profiling." In: *Proceedings of the PARCO 2005 conference*. 2005.

[43] J. H. Meinke, S. Mohanty, and O. Zimmermann. "Protein Folding and Structure Prediction at the Simulation Laboratory Biology." In: *NIC Symposium 2010*. IAS Series. Forschungszentrums Jülich, 2010, pp. 87–94.

[44] Marjan Mernik, Jan Heering, and Anthony M Sloane. "When and how to develop domain-specific languages." In: *ACM Computing Surveys* 37.4 (2005), pp. 316–344. URL: http://portal.acm.org/citation.cfm?doid=1118890.1118892.

[45] Anna Morajko et al. "MATE: Toward Scalable Automated and Dynamic Performance Tuning Environment." English. In: *Applied Parallel and Scientific Computing*. Vol. 7134. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 430–440. ISBN: 978-3-642-28144-0. DOI: 10.1007/978-3-642-28145-7_42. URL: http://dx.doi.org/10.1007/978-3-642-28145-7_42.

[46] Technische Universität München. *ParaGauss - a program package for high-performance computations of molecular systems*. URL: `http://www.theochem.tu-muenchen.de/welcome/index.php?option=com_content&task=view&id=61` (visited on 07/18/2015).

[47] Peter Murray-Rust and Henry S Rzepa. "Chemical Markup, XML, and the Worldwide Web. 1. Basic Principles." In: *Journal of Chemical Information and Computer Sciences* 39.6 (1999), pp. 928–942. DOI: `10.1021/ci990052b`. URL: `http://pubs.acs.org/doi/abs/10.1021/ci990052b`.

[48] Jarek Nieplocha et al. "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit." In: *International Journal of High Performance Computing Applications* 20.2 (), pp. 203–231.

[49] Astrid Nikodem et al. "Load balancing by work-stealing in quantum chemistry calculations: Application to hybrid density functional methods." In: *International Journal of Quantum Chemistry* 114.12 (2014), pp. 813–822. ISSN: 1097-461X. DOI: `10.1002/qua.24677`. URL: `http://dx.doi.org/10.1002/qua.24677`.

[50] Yury Oleynik. "Automatic Characterization of Performance Dynamics with Periscope." PhD thesis. 2014.

[51] Nuno Oliveira et al. "Domain Specific Languages: A Theoretical Survey." In: *INForum'09 — Simpósio de Informática*. Lisboa, Portugal: Faculdade de Ciências da Universidade de Lisboa, Sept. 2009, pp. 35–46.

[52] *Open Trace Format 2*. Accessed on 20.09.2015. URL: `https://silc.zih.tu-dresden.de/otf2-current/html/`.

[53] Oracle. *Javadoc Tool Homepage*. URL: `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`.

[54] James Dean Palmer and Eddie Hillenbrand. "Reimagining Literate Programming." In: *Work* (2009), pp. 1007–1014. DOI: `10.1145/1639950.1640072`. URL: `http://portal.acm.org/citation.cfm?doid=1639950.1640072`.

[55] Cherri M. Pancake. "Applying Human Factors to the Design of Performance Tools." In: *Euro-Par 99 Parallel Processing*. Vol. 1685. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 44–60. ISBN: 978-3-540-66443-7.

[56] *Paradyn Tools Project*. Accessed on 14.08.2015. URL: `http://www.paradyn.org/`.

[57] David Peleg. *Distributed computing: a locality-sensitive approach*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2000. ISBN: 0-89871-464-8.

[58] Weerapong Phadungsukanan et al. "The semantics of Chemical Markup Language (CML) for computational chemistry : CompChem." In: *Journal of cheminformatics* 4.1 (Jan. 2012), p. 15. ISSN: 1758-2946. DOI: `10.1186/1758-2946-4-15`. URL: `http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=3434037\&tool=pmcentrez\&rendertype=abstract`.

[59] Vreda Pieterse, Derrick G Kourie, and Andrew Boake. "A case for contemporary literate programming." In: *Computer* (2004), pp. 2–9. URL: `http://portal.acm.org/citation.cfm?id=1035054`.

[60] T Plewa, T Linde, and V G Weirs. *Adaptive Mesh Refinement - Theory and Applications*. Lecture Notes in Computational Science and Engineering. Springer, 2003. ISBN: 9783540211471. URL: `http://books.google.de/books?id=R4YFoTP3l6cC`.

[61] H. QUINEY and S. WILSON. *THEORY AND COMPUTATION IN THE STUDY OF MOLECULAR STRUCTURE*. Progress in Theoretical Chemistry and Physics Series. Springer Netherlands, 2006. Chap. I, pp. 3–12. ISBN: 978-1-4020-4527-1. DOI: `10.1007/1-4020-4528-X\_1`. URL: `http://books.google.de/books?id=MxZhcgIg9x0C`.

[62] D. C. Rapaport. *The Art of Molecular Dynamics Simulation*. 2nd edition. Cambridge University Press., 2004.

[63] Eric S. Raymond. *The Art of Unix Programming*. Addison-Wesley Professional, 2003.

[64] Martin Roderus et al. "Scheduling Parallel Eigenvalue Computations in a Quantum Chemistry Code." In: *Euro-Par 2010 - Parallel Processing*. Vol. 6272. Lecture Notes in Computer Science. 2010, pp. 113–124.

[65] Jennifer Rowley. "The wisdom hierarchy: representations of the DIKW hierarchy." In: *Journal of Information Science* 33.2 (2007), pp. 163–180. DOI: `10.1177/0165551506070706`. eprint: `http://jis.sagepub.com/content/33/2/163.full.pdf+html`. URL: `http://jis.sagepub.com/content/33/2/163.abstract`.

[66] N. Rutar and J.K. Hollingsworth. "Data Centric Techniques for Mapping Performance Measurements." In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. May 2011, pp. 1274 –1281. DOI: `10.1109/IPDPS.2011.275`.

[67] Jean E Sammet. *Programming Languages: History and Fundamentals*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1969. ISBN: 0137299885.

[68] Beverly A Sanders et al. "A Block-Oriented Language and Runtime System for Tensor Algebra with Very Large Arrays." In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. ISBN: 978-1-4244-7559-9. DOI: `10.1109/SC.2010.3`. URL: `http://dx.doi.org/10.1109/SC.2010.3`.

[69] *Scalsca*. Accessed on 14.08.2015. URL: `http://www.scalasca.org/`.

[70] Eric Schulte et al. "A Multi-Language Computing Environment for Literate Programming and Reproducible Research." In: *Journal of Statistical Software* 46.3 (2012), pp. 1–24. ISSN: 1548-7660. URL: `http://www.jstatsoft.org/v46/i03`.

[71] *Score-P Scalable Performance Measurement Infrastructure for Parallel Codes*. Accessed on 14.08.2015. URL: `http://www.vi-hps.org/projects/score-p/`.

[72] Sameer S. Shende and Allen D. Malony. "The TAU Parallel Performance System." In: *The International Journal of High Performance Computing Applications* 20 (2006), pp. 287–311. DOI: `DOI:10.1177/1094342006064482`.

[73] Sameer Suresh Shende. "The role of instrumentation and mapping in performance measurement." PhD thesis. 2001.

[74] S Shende et al. "Performance evaluation of adaptive scientific applications using TAU." In: *International Conference on Parallel Computational Fluid Dynamics*. 2005.

[75] Robert Sinkovits. *Introduction to Data and Memory Intensive Computing*. Presentation at Gordon Summer Institute & Cyberinfrastructure Summer Institute for Geoscientists. Accessed on 20.08.2012. Aug. 2011. URL: `http://education.sdsc.edu/gordon2011/gsi/IntroToDataIntensive\_GSI11\_Sinkovits.pdf`.

[76] Thomas Martin Soini. "Self-Interaction, Delocalization, and Static Correlation Artifacts in Density Functional Theory: Studies with the Program ParaGauss." PhD thesis. 2015.

[77] Giriprasad Sridhara et al. "Towards automatically generating summary comments for Java methods." In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ASE '10. New York, NY, USA: ACM, 2010, pp. 43–52. ISBN: 978-1-4503-0116-9. DOI: `10.1145/1858996.1859006`. URL: `http://doi.acm.org/10.1145/1858996.1859006`.

[78] The Connection Machine System. *CM Fortran Programming Guide*. 1991, pp. 1–156.

[79] Z. Szebenyi, F. Wolf, and B. Wylie. "Performance Analysis of Long-Running Applications." In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum* (May 2011), pp. 2105–2108.

[80] *TAU Performance System*. Accessed on 22.09.2015. URL: `http://www.cs.uoregon.edu/research/tau/home.php`.

[81] Thomas Trappenberg. *Fundamentals of Computational Neuroscience*. 2nd edition. Oxford University Press, 2010.

[82] M Valiev et al. "NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations." In: *Computer Physics Communications* 181.9 (2010), pp. 1477–1489. ISSN: 0010-4655. DOI: `10.1016/j.cpc.2010.04.018`. URL: `http://www.sciencedirect.com/science/article/pii/S0010465510001438`.

[83] *Vampir - Performance Optimization*. Accessed on 22.09.2015. URL: `https://www.vampir.eu/`.

[84] *VampirTrace*. Accessed on 20.09.2015. URL: `http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/projekte/vampirtrace`.

[85] Dagmar Waltemath et al. "Reproducible computational biology experiments with SED-ML - The Simulation Experiment Description Markup Language." In: *BMC Systems Biology* 5.1 (2011), p. 198. ISSN: 1752-0509. DOI: `10.1186/1752-0509-5-198`. URL: `http://www.biomedcentral.com/1752-0509/5/198`.

[86] P. Wesseling. *Introduction to multigrid methods*. Technical Report. NASA ICASE Report No. 95-11, 1995.

[87] P H Worley. "Phase modeling of a parallel scientific code." In: *Scalable High Performance Computing Conference, SHPCC-92, Proceedings*. 1992, pp. 322–327.

[88] Oleg Zikanov. *Essential Computational Fluid Dynamics*. Wiley, 2010.