PROGRAMMING ABSTRACTIONS FOR DATA LOCALITY

2014 Workshop on
Programming Abstractions
for Data Locality

Lugano, Switzerland
April 28–29, 2014

Programming Abstractions for Data Locality

April 28 – 29, 2014, Swiss National Supercomputing Center (CSCS), Lugano, Switzerland

Co-Chairs

Didem Unat (Koç University)   Torsten Hoefler (ETH Zürich)   John Shalf (LBNL)
Thomas Schulthess (CSCS)   Anshu Dubey (LBNL)

Workshop Participants/Co-Authors

Adrian Tate (Cray)

Amir Kamil (Lawrence Berkeley National Laboratory)

Anshu Dubey (Lawrence Berkeley National Laboratory)

Armin Größlinger (University of Passau)

Brad Chamberlain (Cray)

Brice Goglin (INRIA)

Carter Edwards (Sandia National Laboratories[1])

Chris J. Newburn (Intel)

David Padua (UIUC)

Didem Unat (Koç University)

Emmanuel Jeannot (INRIA)

Frank Hannig (University of Erlangen-Nuremberg)

Gysi Tobias (ETH Zürich)

Hatem Ltaief (KAUST)

James Sexton (IBM)

Jesus Labarta (Barcelona Supercomputing Center)

John Shalf (Lawrence Berkeley National Laboratory)

Karl Fuerlinger (Ludwig-Maximilians-University Munich)

Kathryn O'Brien (IBM)

Leonidas Linardakis (Max Planck Inst. for Meteorology)

Maciej Besta (ETH Zürich)

Marie-Christine Sawley (Intel, Europe)

Mark Abraham (KTH)

Mauro Bianco (CSCS)

Miquel Pericàs (Chalmers University of Technology)

Naoya Maruyama (RIKEN)

Paul Kelly (Imperial College)

Peter Messmer (Nvidia)

Robert B. Ross (Argone National Laboratory)

Romain Cledat (Intel)

Satoshi Matsuoka (Tokyo Institute of Technology)

Thomas Schulthess (CSCS)

Torsten Hoefler (ETH Zürich)

Vitus Leung (Sandia National Laboratories)

Executive Summary

The goal of the workshop and this report is to identify common themes and standardize concepts for locality-preserving abstractions for exascale programming models. Current software tools are built on the premise that computing is the most expensive component, we are rapidly moving to an era that computing is cheap and massively parallel while data movement dominates energy and performance costs. In order to respond to exascale systems (the next generation of high performance computing systems), the scientific computing community needs to refactor their applications to align with the emerging data-centric paradigm. Our applications must be evolved to express information about data locality. Unfortunately current programming environments offer few ways to do so. They ignore the incurred cost of communication and simply rely on the hardware cache coherency to virtualize data movement. With the increasing importance of task-level parallelism on future systems, task models have to support constructs that express data locality and affinity. At the system level, communication libraries implicitly assume all the processing elements are equidistant to each other. In order to take advantage of emerging technologies, application developers need a set of programming abstractions to describe data locality for the new computing ecosystem. The new programming paradigm should be more data centric and allow to describe how to decompose and how to layout data in the memory.

Fortunately, there are many emerging concepts such as constructs for *tiling*, *data layout*, *array views*, *task and thread affinity*, and *topology aware communication libraries* for managing data locality. There is an opportunity to identify commonalities in strategy to enable us to combine the best of these concepts to develop a comprehensive approach to expressing and managing data locality on exascale programming systems. These programming model abstractions can expose crucial information about data locality to the compiler and runtime system to enable performance-portable code. The research question is to identify the right level of abstraction, which includes techniques that range from template libraries all the way to completely new languages to achieve this goal.

---

[1]Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Contents

Chapter 1

Introduction

With the end of classical technology scaling, the clock rates of high-end server chips are no longer increasing, and all future gains in performance must be derived from explicit parallelism [58, 84]. Industry has adapted by moving from exponentially increasing clock rates to exponentially increasing parallelism in an effort to continue improving computational capability at historical rates [10, 11, 1]. By the end of this decade, the number of cores on a leading-edge HPC chip is expected to be on the order of thousands, suggesting that programs have to introduce 100x more parallelism on a chip than today. Furthermore, the energy cost of data movement is rapidly becoming a dominant factor because the energy cost for computation is improving at a faster rate than the energy cost of moving data on-chip [59]. By 2018, further improvements to compute efficiency will be hidden by the energy required to move data to the computational cores on a chip. Whereas current programming environments are built on the premise that computing is the most expensive component, HPC is rapidly moving to an era where computing is cheap and ubiquitous and data movement dominates energy costs. These developments overturn basic assumptions about programming and portend a move from a computation-centric paradigm for programming computer systems to a more data-centric paradigm. Current compute-centric programming models fundamentally assume an abstract machine model where processing elements within a node are equidistant. Data-centric models, on the other hand, provide programming abstractions that describe how the data is laid out on the system and apply the computation to the data where it resides (in-situ). Therefore, programming abstractions for massive concurrency and data locality are required to make future systems more usable.

In order to align with emerging exascale hardware constraints, the scientific computing community will need to refactor their applications to adopt this emerging *data-centric* paradigm, but modern programming environments offer few abstractions for managing data locality. Absent these facilities, the application programmers and algorithm developers must manually manage data locality using ad-hoc techniques such as loop-blocking. It is untenable for applications programmers to continue along our current path because of the labor intensive nature and lack of automation for these transformations offered by existing compiler and runtime systems. There is a critical need for abstractions for expressing data locality so that the programming environment can automatically adapt to optimize data movement for the underlying physical layout of the machine.

1.1 Workshop

Fortunately, there are a number of emerging concepts for managing data locality that address this critical need. In order to organize this research community create an identity as an emerging research field, a two day workshop was held at CSCS on April 28-29 to bring researchers from around the world to discuss their technologies and research directions. The purpose of the *Workshop on Programming Abstractions for Data Locality (PADAL)* was to identify common themes and standardize concepts for locality-preserving abstractions for exascale programming models (*http://www.padalworkshop.org*). This report is a compilation of the workshop findings organized so that they can be shared with the rest of the HPC community to define the scope of this field of research, identify emerging opportunities, and promote a roadmap for future research investments in emerging data-centric programming environments.

1.2 Organization of this Report

Chapter 3 discusses the requirements of applications and their wish list from the programming environments. There are numerous abstractions for multidimensional arrays through *tiling, array views, layouts* and *iterators* to managing data locality, which are described in Chapter 4 and their language and compiler approaches are discussed in Chapter 5 of this document. These abstractions also extend to task-oriented asynchronous programming paradigms (Chapter 6) and to system-level optimization issues (Chapter 7), such as optimizing the mapping of the application communication topology to the underlying network topology. These programming model abstractions can expose crucial information about data locality to the compiler and runtime system to enable performance-portable code. The research question is to identify the right level of abstraction, which includes techniques that range from template libraries all the way to new languages to achieve this goal. By bringing together pioneers in the field of developing new data-centric programming models, there is an opportunity to identify commonalities in strategy to enable us to combine the best of these concepts to develop a comprehensive approach to expressing and managing data locality on exascale programming systems.

1.3 Summary of Findings and Recommendations

In this section, we provide a succinct list of findings and recommendations that are derived from the conclusions of five panels at the workshop

1. Motivating applications and their requirements,

2. Data structure and layout abstractions,

3. Language and compiler support for data locality,

4. Data locality in runtimes for task models, and

5. State-scale data locality management.

The details of findings and recommendations are discussed in their respective chapters.

1.3.1 Motivating Applications and Their Requirements

Findings: Applications must be refactored to work within the constraints of anticipated exascale machine architectures. Being able to express the latent data locality is critical, however, there are no real options for doing so in the current programming environments. The available options usually force a choice between performance and portability, and often act as inhibitors instead of aiding compilers in extracting the locality information. Because the need is urgent, and application developers are chary of dependencies they cannot rely on, if there are no formal agreements on mechanisms for expressing locality, there is a real possibility that every application will develop its own ad-hoc solution. This is clearly suboptimal because of the massive duplication of efforts in development and long-term cost incurred by continued maintenance of numerous ad-hoc implementations. It will be **much** harder to get applications to change **after** they have integrated a coping mechanism that works.

Recommendations: Application developers must be part of the conversation regarding the development of new programming environment support. Finding commonalities across the many constituent applications can provide guidance to developers of data-centric programming models and tools. Getting functioning abstractions for data locality into the hands of application and algorithm developers **early** in the process of code refactoring is critical. The window of opportunity for this interaction will be open for a few generations of systems on the path towards exascale at the end of the decade.

1.3.2 Data Structures and Layout Abstractions

Findings: There are a number of existing library, annotation, and compiler-based techniques that are built around the abstraction of data structures and layout that rely upon very similar underlying semantics. These semantics include the association of additional metadata with array types that describe data decomposition (e.g., tiling) and data layouts and the use of lambda functions (anonymous functions that are now part of the C++11 standard) to abstract loop ordering and to promote array dimensionality a first-class citizen. There are a broad array of examples of variations on these same themes of data layout abstractions that have demonstrated valuable performance portability properties, and are even being pressed into service for production codes in library form (Dash, TiDA, Gridtools, and Kokkos).

Recommendations: In the area of Data Structures and Layout Abstractions, the various formulations that use common underlying semantics (lambdas, dimensionality, and metadata describing tiling and layout) are well positioned for some form of standardization of practices through meetings and agreements between the many parties who have created implementations of these formulations. The value of a standardization effort would be to enable a common runtime system to serve the needs of multiple implementations of these constructs, and also to pave the way for compiler support. A prerequisite for extensions to any language standard is such standardization of the underlying semantics.

1.3.3 Language and Compiler Support for Data Locality

Findings: While locality-aware programming can be expressed using library-based notations (such as embedded DSLs implemented via template metaprogramming), we identified several benefits of raising such abstractions to the language level, including: support for clearer syntax, which can clarify a program's locality decisions for a human reader while also making the author's intention manifest to the compiler; support for stronger semantic checks and type-system-based guarantees within the compilation environment; and optimization opportunities in which the compiler can improve the code's execution in ways that a strictly library-based solution could not. A common theme across most of the programming languages surveyed in the workshop (not to mention the library-based approaches) was the specification of locality concerns as part of a program's declarations rather than its computation sections—for example, by describing how an array is distributed to a set of memories as part of its declaration rather than annotating each individual loop nest or array operation with such information. This approach permits a programmer to switch between distinct locality options in a data-centric way, allowing the compiler and/or runtime to propagate those choices down to the computations, thereby minimizing the amount of code that must be modified as architectures and program requirements change.

Recommendations: Our recommendation is to continue pursuing library-based programming notations as a means of prototyping new abstractions, but to simultaneously invest in language- and compiler-based solutions in order to maximize the return on the user's investment via the strengths afforded by a language-based approach. We recommend pursuing languages that support a *multiresolution philosophy* in which programmers can seamlessly move from higher-level declarative abstractions to lower-level imperative control as needed by their algorithm or its requirements; in such an approach, programmers should also be able to author and deploy their own higher-level abstractions (e.g., distributed data structures) using the lower-level concepts. To maximize adoptability and avoid throwing highly-tuned code away, such languages must remain interoperable with conventional ones. There was also a strong consensus among the group to avoid compiler-based techniques that prematurely lower high-level abstractions to an equivalent but simplified internal representation, thereby discarding crucial locality-oriented semantic information. A key example is the lowering of multidimensional arrays to 1D arrays with per-dimension offsets and multipliers.

1.3.4 Data Locality in Runtimes for Task Models

Findings: Task models have historically focused on balancing computational work among parallel processing elements using simple mechanisms such as work-stealing and self-scheduling. However, as we move forward a locality agnostic balancing mechanism will optimize computational load-imbalances in a manner that runs at cross purposes to the need to minimize data movement. Models that make use of functional semantics

to express tasks to facilitate more flexible task migration are also well structured to facilitate locality-aware or data-centric models, but work remains on effective locality-aware heuristics for the runtime/scheduling-algorithm that properly trade-off load-balancing efficiency with the cost of data movement.

Recommendations: Information on data locality and lightweight cost models for the cost of migrating data will play a central role in creating locality-aware runtime systems that can schedule tasks in a manner that minimizes data-movement. Such locality-aware advanced runtime systems are still an active area of research. The semantics for expressing inter-task locality are emerging in runtime systems such as Open Community Runtime, CHARM++, Swarm, and HPX, but the optimal set of heuristics or mechanisms to effectively exploit that information requires further research.

1.3.5 System-Scale Data Locality Management

Findings: The cost of allocating, moving or accessing data is increasing compared to that of processing it. With the deepening of the memory hierarchy and greater complexity of interconnection networks, the coherence management, the traffic and the contention in the interconnection network will have a huge impact on the application's runtime and energy consumption. Moreover, in a large-scale system multiple applications run simultaneously, and therefore compete for resources. The locality management must therefore take into account local constraints (the way the application behaves) and system-scale constraints (the way it accesses the resources). A global integration of these two types of constraints is the key for enabling scalability of applications execution in future.

Recommendations: Our recommendation is to address several complementary directions: models, abstraction and algorithms for managing data locality at system scale. New models are required to describe the topology on which the application is running both at the node level and at the network level. New abstractions will provide means of expressing the way to access system level services such as storage or the batch scheduler by the applications. These abstractions must expose the topology in a generic manner without deeply impacting the programming model while also providing scalable mapping algorithms that account for the deep hierarchy and complex topology. It is critical that this research be done co-operatively with other aspects of data managment in order to avoid optimization conflicts and also to offer a unified view of the system and its locality management.

Chapter 2

Background

The cost of data movement has become the dominant factor of a high performance computing system both in terms of energy consumption and performance. To minimize data movement, applications have to be optimized both for vertical data movement in the memory hierarchy and horizontal data movement between processing units. These hardware challenges have been modest enough that the community has largely relied upon compiler technology and software engineering practices to mitigate the coarse-grained effects such as manual loop blocking or 2-level MPI+X parallelism. The effects were modest enough that these manual techniques were sufficient to enable codes to perform on different architectures. However, with the exponential rise in explicit parallelism and increasing energy cost of data movement relative to computation, application developers need a set of programming abstractions to describe data locality on the new computing ecosystems.

2.1 Hardware Trends

This section will briefly cover the primary hardware architecture trends that have motivated the move from a compute-centric programming model towards a data-centric model.

2.1.1 The End of Classical Performance Scaling

The year 2004 marked the approximate end of Dennard Scaling because chip manufacturers could no longer reduce voltages at the historical rates. Other gains in energy efficiency were still possible; for example, smaller transistors with lower capacitance consume less energy, but those gains would be dwarfed by leakage currents. The inability to reduce the voltages further did mean, however, that clock rates could no longer be increased within the same power budget. With the end of voltage scaling, single processing core performance no longer improved with each generation, but performance could be improved, theoretically, by packing more cores into each processor. This multicore approach continues to drive up the theoretical peak performance of the processing chips, and we are on track to have chips with thousands of cores by 2020. This increase in parallelism via raw core count is clearly visible in the black trend line in Peter Kogge's classic diagram (Figure 2.1) from the 2008 DARPA report [58]. This is an important development in that programmers outside the small cadre of those with experience in parallel computing must now contend with the challenge of making their codes run effectively in parallel. Parallelism has become everyone's problem and this will require deep rethinking of the commercial software and algorithm infrastructure.

2.1.2 Data Movement Dominates Costs

Since the loss of Dennard Scaling, a new technology scaling regime has emerged. Due to the laws of electrical resistance and capacitance, a wire's intrinsic energy efficiency for a fixed-length wire does not improve appreciably as it shrinks down with Moore's law improvements in lithography as shown in Figure 2.2. In contrast, the power consumption of transistors continues to decrease as their gate size (and hence capacitance)

How much parallelism must be handled by the program?
From Peter Kogge (on behalf of Exascale Working Group), "Architectural *Challenges* at the Exascale Frontier", June 20, 2008
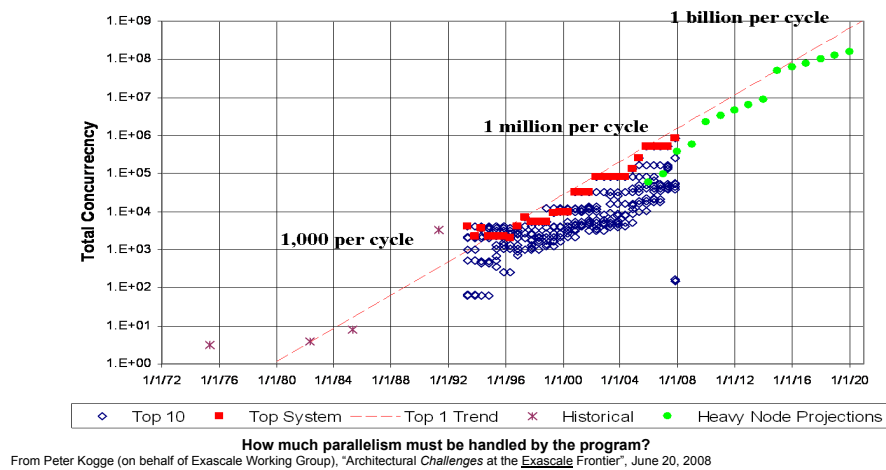
Figure 2.1: Explicit parallelism of HPC systems will be increasing at an exponential rate for the foreseeable future.
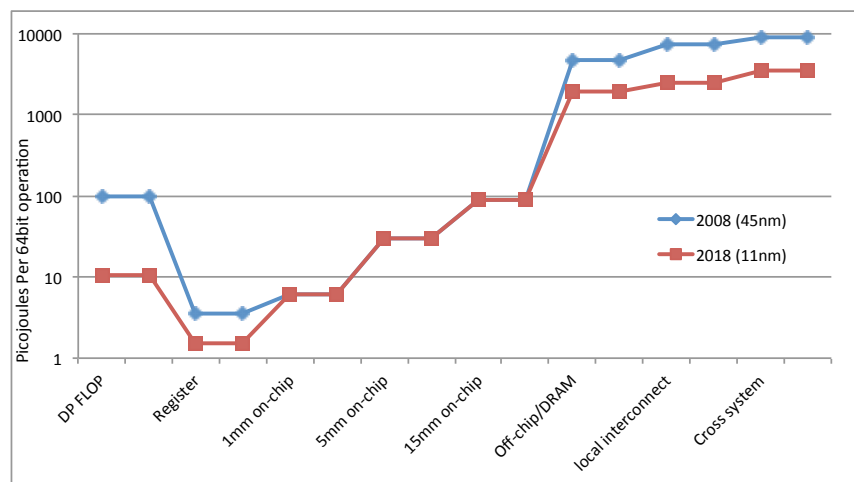

Figure 2.2: The cost of moving the data operands exceeds the cost of performing an operation upon them.

decreases. Since the energy efficiency of transistors is improving as their sizes shrink, and the energy efficiency of wires is not improving, the point is rapidly approaching where the energy needed to move the data exceeds the energy used in performing the operation on those data.

Data locality has long been a concern for application development on supercomputers. Since the advent of caches, vertical data locality has been extraordinarily important for performance, but recent architecture trends have exacerbated these challenges to the point that they can no longer be accommodated using existing methods such as loop blocking or compiler techniques. This report will identify numerous opportunities to greatly improve automation in these areas.

2.1.3   Increasingly Hierarchical Machine Model

Future performance and energy efficiency improvements will require fundamental changes to hardware architectures. The most significant consequence of this assertion is the impact on the scientific applications that run on current high performance computing (HPC) systems, many of which codify years of scientific domain knowledge and refinements for contemporary computer systems. In order to adapt to exascale architectures, developers must be able to reason about new hardware and determine what programming models
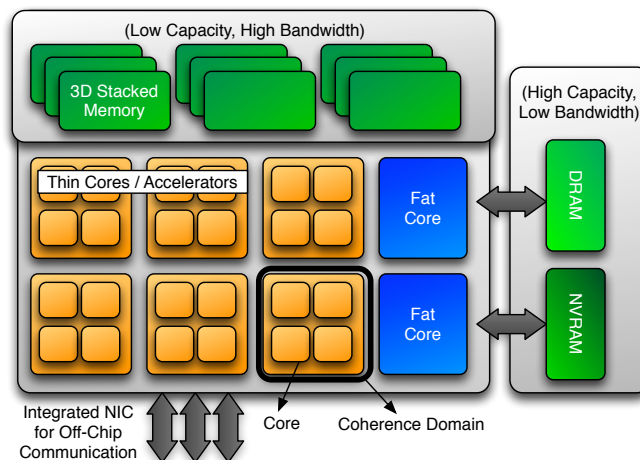
Figure 2.3: The increasingly hierarchical nature of emerging exascale machines will make conventional manual methods for expressing hierarchy and locality very challenging to carry forward into the next decade.

and algorithms will provide the best blend of performance and energy efficiency into the future. While many details of the exascale architectures are undefined, an abstract machine model (AMM) enables application developers to focus on the aspects of the machine that are important or relevant to performance and code structure. These models are intended as communication aids between application developers and hardware architects during the co-design process. Figure 2.3 depicts an AMM that captures the anticipated evolution of existing node architectures based on current industry roadmaps [3].

As represented in the figure, future systems will express more levels of hierarchy than we are normally accustomed to in our existing 2-level MPI+X programming models. Not only are there more levels of hierarchy, it is likely that the topology of communication will become important to optimize for. Programmers are already facing NUMA (non-uniform-memory access) performance challenges within the node, but future systems will see increasing NUMA effects between cores within an individual chip die in the future. Overall, our current programming models and methodologies are ill-equipped to accommodate the changes to the underlying abstract machine model, and this breaks our current programming systems.

2.2   Limitations of Current Approaches

Architectural trends break our existing programming paradigm because the current software tools are focused on equally partitioning computational work. In doing so, they implicitly assume all the processing elements are equidistant to each other and equidistant to their local memories within a node. For example, commonly used modern threading models allow a programmer to describe how to parallelize loop iterations by dividing the iteration space (the computation) evenly among processors and allowing the memory hierarchy and cache coherence to move data to the location of the compute. Such a compute-centric approach no longer reflects the realities of the underlying machine architecture where data locality and the underlying topology of the data-path between computing elements are crucial for performance and energy efficiency. There is a critical need for a new *data-centric* programming paradigm that takes data layout and topology as a primary criteria for optimization and migrates compute appropriately to operate on data where it is located.

The applications community will need to refactor their applications to align with this emerging *data-centric* paradigm, but the abstractions currently offered by modern programming environments offer few abstractions for managing data locality. Absent these facilities, the application programmers and algorithm developers must manually manage data locality using manual techniques such as strip-mining and loop-blocking. To optimize data movement, applications must be optimized both for vertical data movement in the memory hierarchy and horizontal data movement between processing units as shown in figure 2.4. Such transformations are labor-intensive and easily break with minor evolutions of successive generations of compute platforms. Hardware features such as cache-coherence further inhibit our ability to manage data

Vertical Locality Management
(spatio-temporal optimization)

Horizontal Locality Management
(topology optimization)
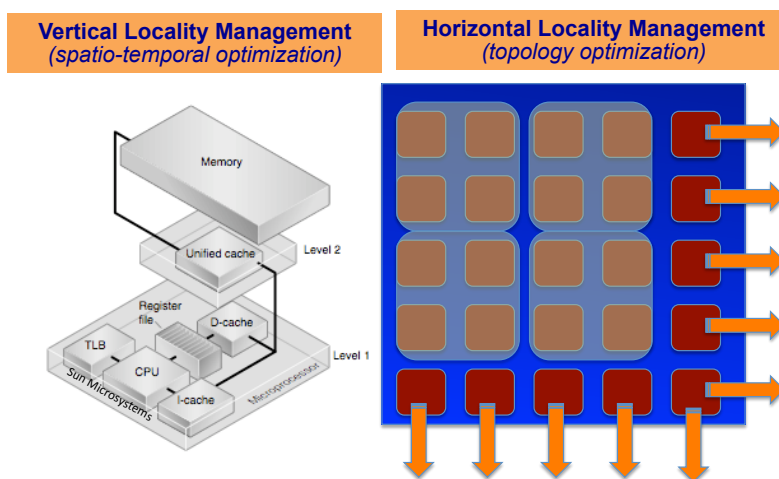
Figure 2.4: Vertical data locality concerns the management of data motion up and down the memory hierarchy whereas horizontal locality concerns communication between peer processing elements.

locality because of their tendency to virtualize data movement. In the future, software-managed memory and incoherent caches or scratchpad memory may be more prevalent, but we have yet to offer powerful abstractions for making such software managed memories productive for the typical application programmer. Thus, application developers need a set of programming abstractions to describe data locality on the new computing ecosystems.

Chapter 3

Motivating Applications and Their Requirements

Discussion of data locality from the perspective of applications requires consideration of the range of modeling methods used for exploring scientific phenomena. Even if we restrict ourselves to only a small group of scientific applications, there is still a big spectrum to be considered. We can loosely map the applications along two dimensions: spatial connectivity, and componentization as shown in Figure 3.1. Spatial connectivity has direct implications on locality: the bottom end of this axis represents zero-connectivity applications which are embarrasingly parallel and at the top end are the ones with dynamic connectivity such as adaptive meshing, with static meshes falling somewhere in-between. The componentization axis is concerned with software engineering where the bottom end represents small static codes, while at the top end are the large-multicomponent codes where the components are swapped in and out of active state, and there is constant development and debugging. The HPC applications space mostly occupies the top right quadrant, and was the primary concern in this workshop.

While the application space itself is very large, the participating applications and experts provided a good representation of the numerical algorithms and techniques used in the majority of state-of-the-art application codes (i.e. COSMO[6, 61], GROMACS[42, 77, 74], Hydra & OP2/PyOP2[15, 79], Chombo[25]). Additionally, because several of them model multi-physics phenomena with several different numerical and algorithmic technologies, they highlight the challenges of characterizing the behavior of individual solvers when embedded in a code base with heterogeneous solvers. These applications also demonstrate the importance of interoperability among solvers and libraries. The science domains which rely on multiphysics modeling include many physical, biological and chemical systems, e.g. climate modeling, combustion, star formation, cosmology, blood flow, protein folding to name only a few. The numerical algorithms and solver technologies on which these very diverse fields rely include structured and unstructured mesh based methods, particle methods, combined particle and mesh methods, molecular dynamics, and many specialized pointwise (or 0-dimensional) solvers specific to the domain.

Algorithms for scientific computing vary in their degree of arithmetic intensity and inherent potential for exploiting data locality.

- For example, GROMACS short-ranged non-bonded kernels treat all pairwise interactions within a group of particles, performing large quantities of floating-point operations on small amounts of heavily-reused data, normally remaining within lowest-level cache. Exploiting data locality is almost free here, yet the higher-level operation of constructing and distributing the particle groups to execution sites, and the lower-level operation of scheduling the re-used data into registers, require tremendous care and quantities of code in order to benefit from data locality at those levels. The long-ranged global component can work at a low resolution, such that a judicious decomposition and mapping confines a fairly small amount of computation to a small subset of processes. This retains reasonable data locality, which greatly reduces communication cost.
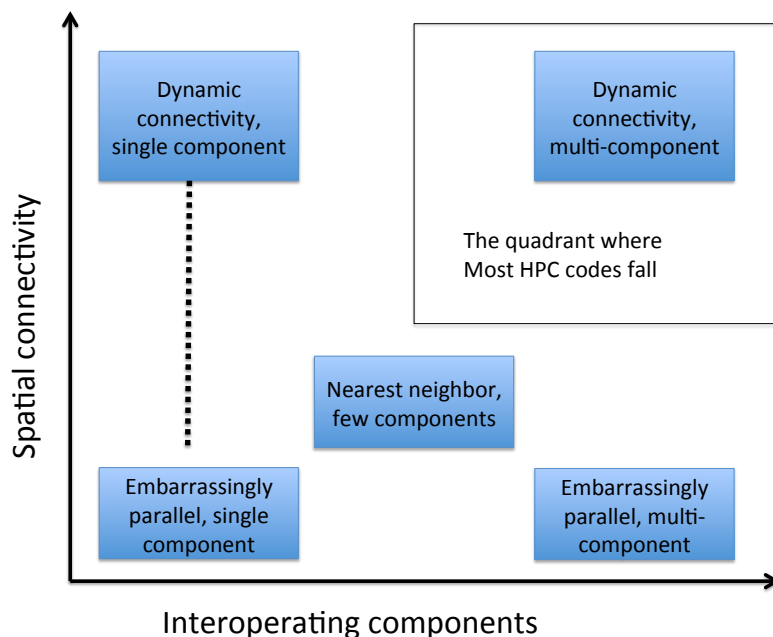
Figure 3.1: The distribution of applications with respect to data locality challenges

- By contrast, lower-order solvers for partial differential equation have few operations per data item even with static meshing and therefore struggle with achieving a high degree of data reuse. Adaptivity in structured AMR (i.e. Chombo) further reduces the ratio of arithmetic operations to data movement by moving the application right along the connectivity axis. Unstructured meshes have an additional layer of indirection that exacerbates this problem.

- Multiphysics applications further add a dimension in the data locality challenge; that of the different data access patterns in different solvers. For example in applications where particles and mesh methods co-exist, the distribution of particles relative to the mesh needs to balance spatial proximity with load balance, especially if the particles tend to cluster in some regions.

There is a genuine concern in the applications communities about protecting the investment already made in the mature production codes of today, and wise utilization of the scarcest of the resources - the developers' time. Therefore, the time-scale of change in paradigms in the platform architecture, that might require major rewrites of codes, is perhaps the most important consideration for the applications community. A stable programming paradigm with a life-cycle that is several times the development cycle of the code must emerge for sustainable science. The programming paradigm itself can take any of the forms under consideration, such as domain-specific languages, abstraction libraries or full languages, or some combination of these. The critical aspects are the longevity, robustness and the reliability of tools available to make the transition.

3.1 State of the Art

Among the domain science communities relying on modeling and simulation to obtain results, there is huge variation in awareness and preparedness for the ongoing hardware platform architecture revolution. The applications component of the workshop was focused on computation-based science and engineering research efforts that rely upon multi-component codes with many moving parts that require HPC resources to compute. For such applications, efficient, sustainable and portable scientific software is an absolute necessity, though not all practitioners in these communities are cognizant of either the extent or the urgency of the need for rethinking their approach to software. Even those that are fully aware of the challenges facing them have been hampered in their efforts to find solutions because of a lack of a stable paradigm

or reliable solutions that they can adopt. For example, viewed on a multi-year time-scale, GROMACS has re-implemented all of its high-performance code several times, always to make better use of data locality [42, 77, 74] and almost never able to make any use of existing portable high-performance external libraries or DSLs. Many of those internal efforts have since been duplicated by third parties with general-purpose, re-usable code, that GROMACS could have used if they been available at the time.

The research communities that have been engaged in the discussions about peta- and exa-scale computing are well informed about the challenges they face. Many have started to experiment with approaches recommended by the researchers from the compilers, programming abstractions and runtime systems communities in order to be better prepared for the platform heterogeneity. At the workshop, examples of many such efforts were presented. These efforts can be mainly classified into: Approaches based on Domain-Specific programming Languages (DSL), library-based methods, or combinations of the two. For example, OP2/PyOP2 [79], STELLA (STEncil Loop LAnguage) [72] and HIPA^{cc} (Heterogeneous Image Processing Acceleration) [64] are *embedded domain-specific languages* (see Section 5) that are tailored for a certain field of application and abstract from details of a parallel implementation on a certain target architecture. PyOP2 uses Python as the host language, while OP2 and the latter approaches use C++. OP2 and PyOP2 target mesh-based simulation codes over unstructured meshes, generating code for MPI clusters, multicore CPUs and GPUs. STELLA considers stencil codes on structured grids. An OpenMP and a CUDA back end are currently under development. HIPA^{cc} targets the domain of geometric multigrid applications on two-dimensional structured grids [65] and provides code generation for accelerators, such as, GPUs (CUDA, OpenCL, RenderScript) and FPGAs (C code that is suited for high-level synthesis). The latest efforts in GROMACS also have moved the code from hand-tuned, inflexible assembly CPU kernels to a form implemented in a compiler- and hardware-portable SIMD intrinsics layer developed internally, for which the code is generated automatically for a wide range of model physics and hardware, including accelerators. In effect, this is an embedded DSL for high-performance short-ranged particle-particle interactions. All the aforementioned approaches can increase *productivity*—e.g., reducing the lines of application code and debugging time—in their target science domain as well as *performance portability* across different compute architectures.

Other efforts, such as the use of tiling within patches in AMR for exposing greater parallelism rely on a library-based approach. Many of these efforts have met with some degree of success. Some are in effect a usable component of an overall solution to be found in future, while others are experiments that are far more informative about how to rethink the data and algorithmic layout of the core solvers. Though all these efforts are helpful in part, and hold promise for the future, none of these approaches currently provide a complete stable solution that applications can use for their transition to long term viability.

3.2 Discussion

There are many factors that affect the decision by the developers of a large scientific library or an application code base to use an available programming paradigm, but the biggest non-negotiable requirement is the stability of the paradigm. The fear of adding a non-robustly-supported critical dependency prevents code developers who use high-end platforms from adopting technologies that can otherwise benefit them. This fear may be due to the lack of guarantee about continued future support or the experimental nature of the technology that might compromise the portability in current and/or future platforms. Often the developers opt for a suboptimal or custom-built solution that does not get in the way of being able to run their simulations. For example, even today research groups exist that use their own I/O solutions with all the attendant performance and maintenance overhead because they were built before the parallel I/O libraries became mature, robust and well supported. A group that has adopted a custom built solution that suits their purpose is much more difficult to persuade to use external solutions even if those solutions are better. It is in general hard to promote adoption of higher-level abstractions unless there is a bound on the dependency through the use of a library that was implemented in a highly portable way, and easy to install and link. For this reason embedded DSLs, code transformation technologies, or any other approaches that provide an incremental path of transition are more attractive to the applications. Any solution that demands a complete rewrite of the code from scratch presents two formidable challenges to the users and developers of the application codes: (1) verifying correctness of algorithms and implementations, since direct comparison with subsections of existing code may not always be possible, and (2) a long gap when production must

continue with existing, non-evolving (and therefore sometimes outdated) code until the new version of the code comes online. The trade-off is the possibility of missing out on some compiler-level optimizations.

There are other less considered but possibly equally critical concerns that have to do with expressibility. The application developers can have a very clear notion of their data model without finding ways of expressing the models effectively in the available data structures and language constructs. The situation is even worse for expressing the latent locality in their data model to the compilers or other translational tools. None of the prevalent mature high-level languages being used in scientific computing have constructs to provide means of expressing data locality. There is no theoretical basis for the analysis of data movement within the local memory or remote memory. Because there is no formalism to inform the application developers about the implications of their choices, the data structures get locked into the implementation before the algorithm design is fully fleshed out. The typical development cycle of a numerical algorithm is to focus on correctness and stability first, and then performance. By the time performance analysis tools are applied, it is usually too late for anything but incremental corrective measures, which usually reduce the readability and maintainability of the code. A better approach would be to model the expected performance of a given data model before completing the implementation, and let the design be informed by the expected performance model throughout the process. Such a modeling tool would need to be highly configurable, so that its conclusions might be portable across a range of compilers and hardware, and valid into the future, in much the same way that numerical simulations often use ensembles of input-parameter space in order to obtain conclusions with reduced bias.

3.3 Application requirements

Most languages provide standard containers and data structures that are easy to use in high-level code, yet very few languages or libraries provide interfaces for the application developer to inform the compiler about expectations of data locality, data layout, or memory alignment. For example, a common concern for the PDE solvers is the data structure containing multiple field components that have identical spatial layout. Should it be an array with an added dimension for the field components or a structure; and within the array or structure, what should be the order for storing in memory for performance. The application's solution is agnostic to the layout, but choice of data structure bakes into the platform on which the code will have a better performance. There are many similar situations that force application implementations to become more platform specific than they need to be. Furthermore, the lack of expressibility can also present false dependencies to the compiler and prevent optimization.

3.4 The Wish List

In response to the application requirements identified above, we constructed a *Wish List* for programming environment features that will efficiently address application needs. The list outlines some abstractions and/or language constructs that would allow the applications to avoid false constraints and be more expressive to the software stack for optimization possibilities.

- Ability to write dimension independent code easily.

- Ability to express functionally equivalent data structures, where one of them can be arbitrarily picked for implementing the algorithm with the understanding that the compiler can do the transformation to one of the equivalent ones suited for the target architecture.

- The ability to map abstract processes to given architectures, and coupled to this, the ability to express these mappings in either memory-unaware or at least flexible formulations.

- More, and more complex, information has been demanded from modelers that is not relevant to the model itself, but rather to the machines. Requiring less information, and allowing formulations close to the models in natural language, is a critical point for applications. This does not restrict the possible approaches, quite the opposite: for example, well engineered libraries that provide the memory and operator mechanics can be very successful, if they provide intelligent interfaces.

- Source-to-source transformations for mapping from high level language to low level language. With appropriately designed transformation tools that allow for language extensions, it might even become possible for the science communities to develop their own domain specific language without having to build the underlying infrastructure.

3.5 Research Areas

Several research areas emerged from the discussions during the workshop that can directly benefit the applications communities. Some of them involve research to be undertaken by the application developers, while others are more applicable to the compilers and tools communities. From the applications perspective, it is important to understand the impact of different locality models. For instance, with the increase in the heterogeneity of available memory options it is worthwhile evaluating whether scratch-pads are more useful to the applications, or should the additional technology just be used to deepen the cache hierarchy. Similarly, within the caching model it is important to know whether adding horizontal caching is a valuable proposition for the applications. These concerns tie into what could be the most urgent area of research for the applications community; what should a high level multi-component framework look like in order to maximize data locality in the presence of diverse and conflicting demands of data movement. The questions to be addressed include:

- what methodology should be used to determine what constitutes a component,

- what degree of locality awareness is appropriate in a component,

- what is the optimum level of abstraction in the component-based design, i.e. who is aware of spatial decomposition, and who is aware of functional decomposition, if it exists,

- how to architect various layers in the framework of the code so that numerical complexity does not interleave with the complexity arising out of locality management, and

- how to account for concerns other than data locality such as runtime management within the framework so that they do not collide with one another.

Tools for aiding the development of scientific software often either hide all complexity of interaction with the system from the application, or they leave it entirely to the application. The former try to cover too many corner cases and become unwieldly, often providing the wrong capability, while the latter all but eliminate portability. True success in achieving scientific and performance goals of the applications is more likely to be achieved by co-operation between the application and the programming models/tools. In an ideal world applications should be able to express locality guidelines best suited to them and the code translators/compilers/runtimes should be able to translate them into performant executables without facing too many optimization blockers.

Chapter 4

Data Structures and Layout Abstractions

In this chapter, we discuss the key considerations when designing data structure and layout abstractions, emerging approaches with (im)mature solutions, and potential research areas. We focus on locality management on data-parallel algorithms and leave the discussion on task-oriented abstractions to Chapter 6. Our goals are to enable a range of abstractions, to converge on ways to specify these abstractions and the mapping between them, and to enable these abstractions to be freely and effectively layered without undue restrictions. The abstractions must be flexible to accommodate the diversity of ways in which the data structures can be organized and represented for several reasons. First, users are diverse in what data representations they find to be convenient, intuitive and expressive. Second, there are differences between what is natural to users and what leads to efficient performance on target architectures. And third, efficient mapping of data structure organization and representations to a diverse collection of target architectures must be accommodated in a general, portable framework. A single abstraction is unlikely to span these three kinds of diversities effectively.

Recently, a number of programming interfaces such as Kokkos [31], TiDA [87], proposed OpenMP4 extensions [26], GridTools [34], hStreams [1], DASH [35], and Array Extensions have arisen to give developers more control over data layout and to abstract the data layout itself from the application. One goal of this workshop was to normalize the terminology used to describe the implementation of each of these libraries, and to identify areas of commonality and differentiation between the different approaches. Commonalities in the underlying implementation may pave the way to standardization of an underlying software infrastructure that could support these numerous emerging programming interfaces.

4.1   Terminology

Before going into further discussions, we define relevant terminology.

Memory Space is an address range of memory with unique memory access characteristics. Examples include different explicitly addressable levels of the memory hierarchy (scratchpads), NUMA nodes of different latencies, different coherence domains associated with subsets of CPUs, GPUs or co-processors, or different types of memories (e.g. cached, software-managed or persistent).

Iteration Space defines the space of indices generated by a loop nest (the space scoped out by the iteration variables of the loops) irrespective of traversal order. The dimensionality of the iteration space is typically defined in terms of the number of loop nests (e.g. a N-nested loop defines a N-Dimensional iteration space). Traversal order indicates the order in which the loop nest visits these indices.

Tiling The purpose of the tiling is to shorten the distance between successive references to the same memory location, so that it is more probable that the memory word resides in the memory levels near to the

---

[1]https://software.intel.com/en-us/articles/prominent-features-of-the-intel-manycore-platform-software-stack-intel-mpss-version-34

processor when the data reuse occurs. This can be done by tiling of the iteration space, or by data space tiling, and *domain decomposition*).

Iteration Space Tiling is a code restructuring technique used to reduce working set size of a program by dividing the iteration space defined by the loop structures into tiles.

Data Space Tiling involves subdividing a rectilinear array into a regular array of rectilinear chunks within the **same** *memory space* in order to maximize the consecutive memory locations that are visited during traversal.

Data Decomposition is the partitioning of a *data structure's* arrays into smaller chunks with the intent of assigning each chunk to a **different** *memory space* for introducing parallelism across chunks or improving data locality.

Data Placement/Pinning is the mapping of the *tiles* or *chunks* of data from a *domain decomposed data structure* to *memory spaces*.

Task Placement/Pinning This is the assignment of threads of execution to a particular physical processor resource and its related hierarchy of *memory spaces*. Many contemporary programming environments do not offer an automatic method to directly relate the *task placement* to the *data placement,* aside from very loose policies such as "first touch memory affinity."

Data Layout is the mapping of a data structure's chunk to the physical addresses corresponding to the addressing scheme used by a given abstraction layer, which in turn provides affine mapping from the logical address space seen by the programmer to the physical layout of the data arrays in a memory.

There may be multiple *abstraction layers*, which together serve a diversity of objectives, as outlined above. For example, users may prefer to refer to struct members of an array (AoS), whereas an array of structures of arrays (AoSoA) layout may be more profitable at the physical storage level. Physical layout may differ between a software-managed memory that, for example, uses Hilbert curves, and a hardware-managed cache. There must be a mapping between the *data layouts* in the various abstraction layers, and potentially in each of the memory spaces.

A computation's memory access is dictated by its *data access pattern* that is a composition of

1. how the computation's parallel tasks traverse the data structure *iteration space* and *traversal order*,

2. how the computation's tasks are scheduled (i.e., the *task placement*),

3. data layout,

4. data placement, and

5. data decomposition.

Performance is constrained by the cost (time and energy) of moving data in service of the computation, which is directly impacted by the computation's data access pattern as represented in Figure 2.2 in the Background chapter of this document. *Data locality* is a proxy measure for the volume and distance of data movement; smaller "distance" and lower volumes imply lower cost. The idea is to shorten the distance between successive references to the same memory location, so that it is more probable that the memory word resides in the memory levels near to the processor when the data reuse occurs. Data locality has both spatial and temporal dimensions as depicted in Figure 2.4 in the Background chapter of this document.

4.2   Key Points

The research community has been developing data structure abstractions for several years to improve the expressiveness of parallel programming environments in order to improve application performance. Recently, there has been a shift towards a more data-centric programming to provide locality abstractions for data

structures because locality has become one of the most important design objectives. During the PADAL workshop, we identified the following design principles as important and desired by the application programmers.

- We seek to improve performance by controlling the separate components (traversal, execution policy, data layout, data placement, and data decomposition) whose composition determines a computation's data access pattern.

- Separation of concerns is important to maximize expressivity and optimize performance. That principle implies a clear distinction between a logical, semantic specification, and lower-level controls over the physical implementation. At the semantic level, sequential work is mapped onto parallel data collections, using logical abstractions of data layout which best promote productivity. Domain experts can also expose semantic characteristics like reference patterns that can inform trade-offs made by tools.

- A computation's algorithm is expressed in terms of operations on certain abstract data types, such as matrices, trees, etc. The code usually expresses data traversal patterns at high level, such as accessing the parent node in a tree, or the elements in a row of a matrix. This expresses the *algorithmic locality* of the data, which may be not related to the *actual* locality of the data access. The actual locality is obtained when the algorithm's traversals, tasks, and data structures are mapped to the physical architecture.

- The efficiency of a computation's on-node data access pattern may depend upon the architecture of the computer on which it executes. Performance may be improved by choosing a data layout appropriate for the architecture. This choice can be made without modifying the computation's source code if the data structure has a *polymorphic data layout*.

- Performance tuners, who may be distinct from the domain experts, may exercise control over performance with a set of interfaces that provide more explicit control over the data movement, and can be distinct from those that specify semantics. Through these interfaces, they may decompose, distribute and map parallel data collections onto efficient physical layouts with specialized characteristics. This may be done in a modular fashion that leaves the semantic expression of the code intuitive, readable and maintainable. This approach presents an explicit "separation of concerns" by offering different interfaces to different programmers (the domain scientists vs. the computer science and hardware experts.

- A key goal of *separation of concerns* is to separate the interface of the architecture-agnostic *abstract machine* (AM) from that of the *physical machine* (PM) using hierarchical abstractions. An AM is necessary to express an algorithm, whether in a functional language, as a PGAS abstraction, or something else. An algorithm written for an AM must be translated into program for a *physical machine* (PM). The PM may or may not be closely related to the AM. The translation effort depends on the "distance" between the AM and the PM. Since there are many different PMs, and the lifetime of applications should be much longer that the lifetime of an architecture, the usual requirement is that an algorithm for an AM should be executed efficiently in the widest possible class of foreseeable PMs. This is the well-known problem of *performance portability*. From this perspective, an AM should be distinct from PMs for two reasons: first it gives the user a uniform view of the programming space, often at a much higher level of abstraction than any specific programming model for a PM; second, the hope is that by being equidistant from any PM, the translation can lead to equally efficient implementations with similar effort.

4.3 State of the Art

There is a concern that data structure and layout features that are needed to "get us to exascale" are lacking in existing, standard languages. There are few options for mitigating this situation.

- Employ standard language features, e.g. in library-based solutions. Languages vary in their support for library-based data layout abstractions.

    - C++ seems to provide an opportunity for enabling data layout abstractions due to its ability to extend the base language syntax using template metaprogramming. C++ meta-programming can cover many desired capabilities, e.g. polymorphic data layout and execution policy, where specialization can be hidden in template implementation and controlled by template parameters. Although C++ template metaprogramming offers more syntactic elegance for expressing solutions, the solution is ultimately a library based approach because the code generated by the template metaprogram is not understood by the baseline compiler and therefore the compiler cannot provide optimizations that take advantage of the higher-level abstractions implemented by the templates. The primary opportunity in the C++ approach is the hope that the C++ standards committee would adopt it as part of the standard. The standards committee is aggressive at adoption, and it already supports advanced features like lambdas, and the C++ Standard Library. However if these syntactic extensions are adopted, compiler writers would still need to explicitly target those templates and make use of the higher-level semantics they represent. At present it is not clear how well this strategy will work out.

    - In contrast to C++, Fortran is relatively limited and inflexible in terms of its ability to extend the syntax, but having multidimensional arrays as first class objects gives it an advatage in expressing data locality. A huge number of applications are implemented in Fortran, and computations with regular data addressing are common. Library-based approaches to extending locality-aware constructs into Fortran are able to exploit the explicit support for multidimensional arrays in the base language. However, these library-based approaches may seem less elegant in Fortran because of the inability to perform syntactic extensions in the base language

    - Dynamically-typed scripting languages like Python, Perl, and Matlab provide lots of flexibility to users, and enable some forms of metaprogramming, but some of that flexibility can make optimization difficult. Approaches to overcome the performance challenges of scripting languages involve using the integrated language introspection capabilities of these languages (particularly Python) that enables the scripting system to intercept known motifs in the code and use Just in Time (JIT) rewriting or specialization. Examples include Copperhead [19] and PyCuda [57], which recognize data-parallel constructs and rewrites and recompiles them as CUDA code for GPUs. SEJITS and the ASP frameworks [18] are other examples that use *specializers* to recognize particular algorithmic motifs and invoke specialized code rewriting rules to optimize those constructs. This same machinery can be used to recognize and rewrite code that uses metaprogramming constructs that express data locality information.

- Augment base languages with directives or embedded domain-specific languages (DSLs). Examples include OpenMP, OpenACC, Threading Building Blocks[2] and Thrust[3].

Most contributors to this report worked within the confines of existing language standards, thereby maximizing the impact and leveraging market breadth of the supporting tool chain (e.g., compilers, debuggers, profilers). Wherever profitable, the research plan is to "redeem" existing languages by amending or extending them, e.g. via changes to the specifications or by introducing new ABIs.

The interfaces the authors of this report are developing are Kokkos [31], TiDA [87], C++ type support, OpenMP extensions to support SIMD, GridTools [34], hStreams [4]), and DASH [35]. The **Kokkos** library supports expressing multidimensional arrays in C++, in which the polymorphic layout can be decided at compile time. An algorithm written with Kokkos uses the AM of C++ with the data specification and access provided by the interface of Kokkos arrays. Locality is managed explicitly by matching the data layout with the algorithm logical locality.

TiDA allows the programmer to express data locality and layout at the array construction. Under TiDA, each array is extended with metadata that describes its layout and tiling policy and topological affinity for

---

[2]https://www.threadingbuildingblocks.org
[3]http://docs.nvidia.com/cuda/thrust
[4]https://software.intel.com/en-us/articles/prominent-features-of-the-intel-manycore-platform-software-stack-intel-mpss-version-34

an intelligent mapping on cores. This metadata follows the array through the program so that a different configuration in layout or tiling strategy do not require any of the loop nests to be modified. Various layout scenarios are supported to enable multidimensional decomposition of data on NUMA and cache coherence domains. Like Kokkos, the metadata describing the layout of each array is carried throughout the program and into libraries, thereby offering a pathway towards better library composability. TiDA is currently packaged as a Fortran library and is minimally invasive to Fortran codes. It provides a tiling traversal interface, which can hide complicated loop traversals, parallelization or execution strategies. Extensions are being considered for the **C++ type system** to express semantics related to the consistency (varying or uniform) of values in SIMD lanes. This is potentially complementary to ongoing investigations in how to introduce new **OpenMP-compatible ABIs** that define a scope within which more relaxed language rules may allow greater layout optimization, e.g. for physical storage layout that's more amenable to SIMD.

GridTools provides a set of libraries for expressing distributed memory implementations of regular grid applications, like stencils. It is not meant to be universal, in the sense that non-regular grid applications should not be expressed using Gridtools libraries, even though possible in principle, for performance reasons. Since the constructs provided by GridTools are high level and semi-functional, locality issues are taken into account at the level of performance tuners and not by application programmers. At the semantic level the locality is taken into consideration only implicitly. The **hStreams** library provides mechanisms for expressing and implementing data decomposition, distribution, data binding, data layout, data reference characteristics, and execution policy on heterogeneous platforms. **DASH** is built on a one-sided communication substrate and provides a PGAS abstraction in C++ using operator overloading. The DASH AM is basically a distributed parallel machine with the concept of hierarchical locality.

As can be seen, there is no single way of treating locality concerns, and there is no consensus on which one is the best. Each of these approaches is appealing in different scenarios that depend on the scope of the particular application domain. There is the opportunity of naturally building higher level interfaces using lower level ones. For instance, TiDA or DASH multidimensional arrays could be implemented using Kokkos arrays, and GridTools parallel algorithms could use the DASH library, and Kokkos arrays for storage, etc. This is a potential benefit from interoperability that arises from using a common language provided with generic programming capabilities.

Ultimately, the use of lambdas to abstract the iteration space and metadata to carry information about the abstracted data layouts are common themes across all of these implementations. This points to a potential for a lower-level standardization of data structures and APIs that can be used under-the-covers by all of these APIs (a common abstraction layer that could be used by each library solution). One outcome of the workshop is to initiate efforts to explicitly define the requirements for a common runtime infrastructure that could be used interoperable across these library solutions.

4.4 Discussion

This chapter presents and begins to resolve several key challenges:

- Defining the abstraction layers. The logical layer is where the domain expert provides a semantic specification and offers hints about the program's execution patterns. The physical layer is where the performance tuner specifies data and execution policy controls that are designed to provide best performance on target machines. These layers are illustrated in Figure 4.1.

- Enumerating the data and execution policy controls of interest. These are listed below in this section and are highlighted in Figure 4.1.

- Suggest some mechanisms which enable a flexible and effective mapping from the logical down to the physical layer, while maintaining a clean separation of controls and without limiting the freedom of expression and efficiency at each layer. One class of mechanisms is oriented around individual data objects, e.g. with data types, and another is oriented around control structures, e.g. with ABIs that enable a relaxation of language rules. The choice between these two orientations is illustrated in Figure 3.1.
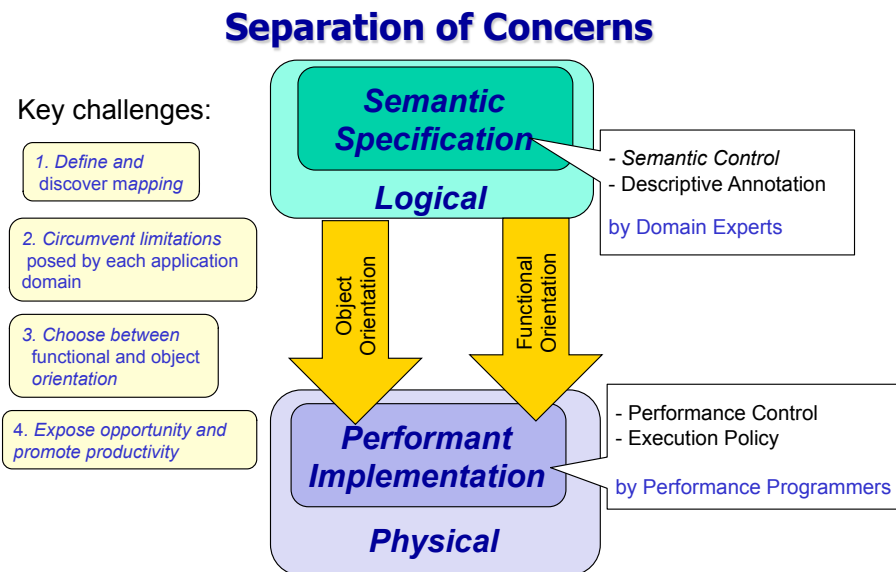
Separation of Concerns

Key challenges:

1. Define and discover mapping

2. Circumvent limitations posed by each application domain

3. Choose between functional and object orientation

4. Expose opportunity and promote productivity

Semantic Specification

Logical

- *Semantic Control*
- Descriptive Annotation

by Domain Experts

Object Orientation

Functional Orientation

Performant Implementation

Physical

- Performance Control
- Execution Policy

by Performance Programmers

Figure 4.1:   Separation of concerns in data structure abstractions

Separation of concerns.   High performance computing presents scientists and performance tuners with two key challenges: exposing parallelism and effectively harvesting that parallelism. A natural separation of concerns arises from those two efforts, whereby the scope of effort for each of domain experts and performance tuning experts can be limited, and the two may be decoupled without overly restricting each other. Charting a solid path forward toward a clean separation of concerns, defining appropriate levels of abstraction, and highlighting properties of language interfaces that will be effective at managing data abstractions are the subject of this effort.

Domain experts specify the work to be accomplished. They want the freedom to use a representation of data that is natural to them. Most of them prefer not to be forced to specify performance-related details. Performance tuning experts want full control over performance, without having to become domain experts. So they want domain experts to fully expose opportunities for parallelism, without over-specifying how that parallelism is to be harvested. This leads to a natural separation of concerns between a logical abstraction layer, at which semantics are specified - the upper box in Figure 3.1, and a lower, physical abstraction layer, at which performance is tuned and the harvesting of parallelism on a particular target machine is controlled - the lower box in the figure. This separation of concerns allows code modifications to be localized, at each of the semantic and performance control layers. The use of abstraction allows high-level expressions to be polymorphic across a variety of low-level trade-offs at the physical implementation layer. Several interfaces are now emerging that maintain the discipline of this separation of concerns, and that offer alternative ways of mapping between the logical and physical abstraction layers.

Performance-related controls pertain to 1) data and to 2) execution policy.

1. Data controls may be used to manage:

   - *Decomposition*, which tends to be either trivial (parameterizable and automatic; perhaps along multiple dimensions) or not (explicit, and perhaps hierarchical)

   - Mechanisms for, and timing of, *distribution* to data space/locality bindings

   - Data *layout*, the arrangement of data according to the addressing scheme, mapping logical abstractions to arrangement in physical storage

   - *Binding*, storage to memories that support a particular access pattern (e.g. read only, streaming), to a phase-dependent depth in the memory hierarchy (prefetching, marking non-temporal), to memory structures which support different kinds of sharing (software-managed or hardware-managed cache), or to certain kinds of near storage (e.g. scalar or SIMD registers)

2. Execution policy controls may be used to manage

   - *Decomposition* of work, e.g. iterations, nested iterations, hierarchical tasks
   - *Ordering* of work, e.g. recursive subdivision, work stealing
   - *Association* of work with data, e.g. moving work to data, binding to hierarchical domains like a node, an OpenMP place, a thread or a SIMD lane

Mechanisms. These controls may be applied at different scopes and granularities through a variety of mechanisms:

- Data types - these are fine-grained, applying to one variable or parameter at a time; they apply across the lexical scope of the variable

- Function or construct modifiers - instead of applying to individual variables, these apply a policy to everything in a function or control construct

- Environmental variable controls - these global policies apply across a whole program

Note that through modularity, the scope to which data types and function or construct modifiers may be refined. For example, with many functions, a given variable's data type may vary by call site.

These controls may be applied at different scopes and granularities through a variety of mechanisms. The interactions and interdependencies of data controls, execution controls, and the management of granularity or scope can get quite complex. We use SIMD as a motivating example to illustrate some of the issues.

- The *order among dimensions* in a multi-dimensional array obviously impacts which dimension's elements are contiguous. This, in turn, can affect which dimension is best to vectorize with a unit stride, to maximize memory access efficiency. Relevant compiler transformations may include loop interchange, data re-layout, and the selection of which loop nesting level to vectorize. The dimension order for an array can be specified with a data type, or an new ABI at a function boundary can be used to create a smaller scope within which indicators of the physcial layout cannot escape, such that the compiler can be left free to re-layout data within that scope.

- A complicating factor for this data layout is that the best layout may vary within a scope. For example, one loop nest may perform better with one dimensional ordering, or one AoSoA (array of structures of arrays) arrangement, while the best performance for the next loop nest may favor a different layout. This can lead to a complex set of trade-offs that are not optimally solved with greedy schemes. It is possible to isolate the different nests in different functions, and to use an ABI to hide where data relayout occurs.

- The number of elements that can be packed into a fixed-width SIMD register may depend on the precision of each element, for some target architectures. Consider a loop iteration that contains a mix of double- and single-precision operands, where vlen single-precision operations may be packed into a single SIMD instruction, whereas since only vlen/2 double-precision operations may be packed into a single instruction, two instructions are required. Matching the number of available operations may require *unrolling the loop* by an additional factor of 2. For situations such as this, a hard encoding of the elements in each SIMD register using types may inhibit the compiler's freedom to make tuning trade-offs.

We have many areas of agreement. Leaving the programmer free from being forced to specify performance controls makes them more *productive*, and leaving the tuner free to add controls without introducing bugs through inadvertent changes to semantics makes them more productive. If the tuner is more productive, and if the tuner has a clear set of priorities of which controls to apply and the ability to apply them effectively, they can achieve better *performance* more quickly. Finally, isolating the performance tuning controls and presenting them in a fashion which allows target-specific implementations to be applied easily make performance more *performance portable*.

4.5   Research Plan

The notion of a separation of concerns around expressing semantics and exerting control, and the taxonomy of data and execution policy controls offer a clearer framework in which to fit current and ongoing research. Some relevant promising research areas are then

- Identify where and how the type system should be augmented in standard languages like C++ to better convey semantics, for example for multidimensional arrays

- Extend the reflection capabilities of standard languages, to ease programmer effort and to remove barriers to freedom of abstraction. See below for more detail.

- Experiment with and promote the use of libraries that enable the separation of semantic expression from target-specific implementations of data and execution policy controls such as iterators.

- Identify and close gaps in the completeness of data and execution policy controls in existing embedded DSLs.

- Identifying other language extensions needed to facilitate data layout abstractions such as tiling.

As we proceed along the course of the research plan suggested above, the following questions need to be evaluated:

- Are some frameworks for providing data and execution policy controls limited in their portability across architectures? Where performance portability and generality is sacrificed, is that specialization worthwhile? Do those specializations tend to be domain specific?

- Is there a minimal set of data or execution policy controls that are found to be of highest impact, that we can recommend to be covered by most all data layout optimization frameworks?

- Can we generalize our formulations of costs for various optimization alternatives into a more general data model that we can use to reason about locality?

- As systems mature, is there a natural progression from total control by a user, to select control only when heuristics aren't good enough, to default heuristics that are good enough for most users? Do our frameworks provide an opportunity to offer users the full range of ease of use through automation to total control and visibility?

- Which data and execution policy controls are best implemented by each of the user, a static compiler, or a dynamic runtime?

- Identifying other language extensions needed to facilitate data layout abstractions

Augmenting C++ reflection capabilities is a key enabler for some forms of data layout abstraction. Since the C++ language committe is showing interest in this, there is an opportunity to drive proposals for solutions in this area. Better *reflection* support can make template-based techniques more tractable, and can enable embedded DSLs.

- Template-based techniques can be used to introduce an abstraction that maps a programmer-focused logical layout to a performant physical storage layout. But since a complete mapping is bidirectional, the designer of the template-based solution has to create a dual mapping, which is laborious and error prone. This is relevant to Arrow Street and SIMD Building Blocks, mentioned above. The *introspection* aspect of reflection allows walking struct or class members and references to infer the reverse mapping, so that this can be performed by an automated system instead of manually. This could significantly enable ease of use.

- Language rules can inhibit the freedom of expression at multiple abstraction layers. For example, suppose an embedded DSL is used to express a data layout abstraction. One might think that a source to source translator could be used to map from the DSL to C/C++ code, and the compiler could remain

free to create an alternate physical storage layout that's more performant. Unfortunately, C and C++ expose physical data layout, establishing a contract with the programmer that limits a compiler's ability to modify the data layout without changing the semantics. The source to source translator can't "reach around" the C/C++ front end in order to directly modify the compiler's internal abstract syntax tree (AST). But C++ extensions to provide the *intercession* aspect of reflection could enable direct modification of the AST.

Chapter 5

Language and Compiler Support for Data Locality

The concept of locality has not yet become a first-class citizen in general-purpose programming languages. As a result, although application programmers are often confronted with the necessity of restructuring program codes to better exploit locality inherent in their algorithms, even a simple simulation code can become highly non-intuitive, difficult to maintain, and non-portable across diverse architectures.

The overarching vision of this workshop is to solve these issues by presenting application programmers with proper abstractions for expressing locality. In this chapter, we explore data locality management in the context of parallel programming *languages* and *compiler* technologies More specifically, we discuss language concepts for data locality, their delivery in general-purpose languages, domain-specific languages and active libraries, and the compiler technology to support them.

Language-based solutions may come in the form of new, general-purpose parallel programming languages. In addition to supporting intuitive syntax, language-based approaches enable ambitious compilation techniques, sophisticated use of type systems, and compile-time checking of important program properties. Language concepts for locality can also be delivered within existing languages, supported in libraries, through metaprogramming in C++, and in "active libraries" with library-specific analysis and optimizations.

Locality is about data, and locality abstractions often refer to abstract data types. Multidimensional arrays are a crucial starting point. Many more complex domains can be characterized by the abstract, distributed data structures on which parallel computation occurs. Domain-specific tools are often focused on particular data structures—graphs, meshes, octrees, structured adaptive-mesh multigrids, etc. While Chapter 4 tackles particular data structures, in this chapter we look at how to build tools to support programmers who build such abstractions.

Locality is also about affinity—the placement of computations (e.g., tasks) relative to the data they access. Dominant HPC programming models have typically been based on long-running tasks executing in fixed locations fetching remote data. Emerging architectures may also compel us to pursue languages in which computations are moved to the data they wish to access. To implement such models, languages will need to also support the creation of remote tasks or activities.

The following terms are used in this chapter:

- *Active library:* a library which comes with a mechanism for delivering library-specific optimizations [89]. Active library technologies differ in how this is achieved - examples include template metaprogramming in C++, Lightweight Modular Staging in Scala [81], source-to-source transformation (for example using tools like ROSE [94]), and run-time code generation, driven either by delayed evaluation of library calls [82], or explicit run-time construction of problem representations or data flow graphs.

- *Embedded domain-specific language:* a technique for delivering a language-based solution within a host, general-purpose language [46]. Active libraries often achieve this to some extent, commonly by overloading to capture expression structure. Truly syntactic embedding is also possible with more ambitious tool support [32].

- *Directive-based language extensions:* tools like OpenMP[1], OpenACC[2], OpenStream[3] decorate a host language, like C++ or Fortran, with annotations. In OpenMP and its derivatives, the objective is that these "pragmas" can be ignored to yield purely sequential code with the same semantics. The directive language is separate from the host language. Directives are similar to annotations in Java and C#, which are user-extensible and often used to drive aspect-oriented transformation tools. Both directives and annotations share problems with integration with the host language. Directive-based tools like OpenMP suffer from compositionality issues, for example in calling a parallel function from within a parallel loop.

- *Global-view vs. Local-view Languages:* Global-view languages are those in which data structures, such as multidimensional arrays, are declared and accessed in terms of their global problem size and indices, as in shared-memory programming. In contrast, local-view languages are those in which such data structures are accessed in terms of local indices and node IDs.

- *Multiresolution Language Philosophy:* This is a concept in which programmers can move from language features that are more declarative, abstract, and higher-level to those that are more imperative, control-oriented, and low-level, as required by their algorithm or performance goals. The goal of this approach is to support higher-level abstractions for convenience and productivity without removing the fine-grained control that HPC programmers often require in practice. Ideally, the high-level features are implemented in terms of the lower-level ones in a way that permits programmers to supply their own implementations. Such an approach supports a separation of roles in which computational scientists can write algorithms at high levels while parallel computing experts can tune the mappings of those algorithms to the hardware platform(s) in distinct portions of the program text.

5.1   Key Points

During the PADAL workshop, we identified the following key points to be considered when designing languages for data locality issues.

- Communication and locality should be clearly evident in the source code, so that programmers have a clear model of data movement and its associated costs. At the same time, the programming language should make it easy to port flat-memory code to locality-aware code, or to write code that can execute efficiently on both local and remote data. One mechanism to accomplish this is to encode locality in the type system, so that modifying the locality characteristics of a piece of code only requires changing type declarations. In languages that support generic programming, this also enables a programmer to write the same code for both local and remote data, with the compiler producing efficient translations for both cases.

- In addition to providing primitives for moving data to where computation is located, a programming language should also enable a user to move computation to where data are located. This is particularly important for irregular applications in which the distribution of data is not known until runtime or changes over the course of the computation. For large data sets, code movement is likely to be significantly cheaper than moving data.

- A program should not require rewriting when moving to a different machine architecture. Instead, the language should provide a machine model that does not have to be hard-coded into an application. In particular, the machine model should be represented separately from user code, using a runtime data structure. The language should either automatically map user code to the machine structure at compile or launch time or provide the user with mechanisms for adapting to the machine structure during execution.

- A unified machine model should be provided that encompasses all elements of a parallel program, including placement of execution, load balancing, data distribution, and resilience.

---

[1]http://openmp.org/
[2]http://www.openacc-standard.org/
[3]http://openstream.info/

- Seamless composition of algorithms and libraries should be supported by the language; composition should not require a code rewrite. The machine model can facilitate composition by allowing a subset of the machine structure to be provided to an algorithm or library.

- The language should provide features at multiple levels of abstraction, following the multiresolution design philosophy. For example, it may provide data-parallel operations over distributed data structures, with the compiler and runtime responsible for scheduling and balancing the computation. At the same time, the language might also allow explicit operations over the local portions of the data structure. Such a language would be a combination of global and local view, providing default global-view declarations and operations while also allowing the user to build and access data structures in a local-view manner.

- Higher-level features in the language and runtime should be built on top of the same lower-level features that the user has access to. This enables a user to replace the built-in, default operations with customized mechanisms that are more suitable to the user's application. The compiler and runtime should perform optimizations at multiple levels of abstraction, enabling such custom implementations to reap the advantages of lower-level optimizations.

5.2 State of the Art

HPF and ZPL are two languages from the 1990s that support high-level locality specifications through the distribution of multidimensional arrays and index sets to rectilinear views of the target processors. Both can be considered *global view* languages, and as a result all communication was managed by the compiler and runtime. A key distinction between the languages was that all communication in ZPL was syntactically evident, while in HPF it was invisible. While ZPL's approach made locality simpler for a programmer to reason about, it also required code to be rewritten whenever a local/non-distributed data structure or algorithm was converted to a distributed one. HPF's lack of syntactic communication cues saved it from this problem, but it fell afoul of others in that it did not provide a clear semantic model for how locality would be implemented for a given program, requiring programmers to wrestle with a compiler to optimize for locality, and to then to rewrite their code when moving to a second compiler that took a different approach.

As we consider current and next-generation architectures, we can expect the locality model for a compute node to differ from one vendor or machine generation to the next. For this reason, the ZPL and HPF approaches are non-viable. To this end, we advocate pursuing languages that make communication syntactically invisible (to avoid ZPL's pitfalls) while supporting a strong semantic model as a contract between the compiler and programmer (to avoid HPF's). Ideally, this model would be reinforced by execution-time queries to support introspection about the placement of data and tasks on the target architecture.

Chapel is an emerging language that takes this prescribed approach, using a first-class language-level feature, the *locale* to represent regions of locality in the target architecture. Programmers can reason about the placement of data and tasks on the target architecture using Chapel's semantic model, or via runtime queries. Chapel follows the Partitioned Global Address Space (PGAS) philosophy, supporting direct access to variables stored on remote locales based on traditional lexical scoping rules. Chapel also follows the multiresolution philosophy by supporting low-level mechanisms for placing data or tasks on specific locales, as well as high-level mechanisms for mapping global-view data structures or parallel loops to the locales. Advanced users may implement these data distributions and loop decompositions within Chapel itself, and can even define the model used to describe a machine's architecture in terms of locales.

X10 [22] is another PGAS language that uses *places* as analogues to Chapel's locales. In X10, execution must be colocated with data. Operating on remote data requires spawning a task at the place that owns the data. The user can specify that the new task run asynchronously, in which case it can be explicitly synchronized later and any return value accessed through a future. Thus, X10 makes communication explicit in the form of remote tasks. Hierarchical Place Trees [92] extend X10's model of places to arbitrary hierarchies, allowing places to describe every location in a hierarchical machine.

Unified Parallel C (UPC), Co-Array Fortran (CAF), and Titanium [93] are three of the founding PGAS languages. UPC supports global-view data structures and syntactically-invisible communication while CAF has local-view data structures and syntactically-evident communication. Titanium has a local-view data

model built around ZPL-style multidimensional arrays. Its type system distinguishes between data guaranteed to be local and data that may be remote using annotations on variable declarations. On the other hand, access to local and remote data is provided by the same syntax. Thus, Titanium strikes a balance between the HPF and ZPL approaches, making communication explicit in declarations but allowing the same code fragments to operate on local and remote data.

UPC, CAF, and Titanium differ from HPF, ZPL, Chapel, and X10 in that programs are written using an explicit bulk-synchronous Single-Program, Multiple Data (SPMD) execution model. These copies of the executing binary form the units of locality within these languages, and remote variable instances are referenced based on the symmetric namespaces inherent in the SPMD model. While the flat SPMD models used in these languages (and commonly in MPI) do allow programmers to colocate data and execution, they will likely be insufficient for leveraging the hierarchical architectural locality present in emerging architectures. Currently such models are often forced to be part of hybrid programming models in which distinct locality abstractions are used to express finer-grained locality concerns.

Recent work in Titanium has replaced the flat SPMD model with the more hierarchical Recursive Single-Program, Multiple-Data (RSPMD) model [53]. This model groups together data and execution contexts into teams that are arranged in hierarchical structures, which match the structure of recursive and compositional algorithms and emerging hierarchical architectures. While the total set of threads is fixed at startup as in SPMD, hierarchical teams can be created dynamically, and threads can enter and exit teams as necessary. Titanium provides a mechanism for querying the machine structure at runtime, allowing the same program to target different platforms by building the appropriate team structure during execution.

Other work has been done to address the limitations of the flat SPMD model in the context of Phalanx [36] and UPC++ [96], both active libraries for C++. The Phalanx library uses the Hierarchical Single-Program, Multiple-Data (HSPMD) model, which is a hybrid of SPMD and dynamic tasking. The HSPMD model retains the cooperative nature of SPMD by allowing thread teams, as in RSPMD, but it allows new teams of threads to be spawned dynamically. Unlike SPMD and RSPMD, the total set of executing threads is not fixed at startup. Both RSPMD and HSPMD allow expression of locality and concurrency at multiple levels, though through slightly different mechanisms, allowing the user to take advantage of hierarchical architectures. The UPC++ library uses RSPMD as its basic execution model but additionally allows X10-style asynchronous tasks to be spawned at remote locations. This allows execution to be moved dynamically to where data are located and adds a further degree of adaptibility to the basic bulk-synchronous SPMD model.

5.3   Discussions

AGREEMENTS

- PRINCIPLES

  - Avoid losing information through premature "lowering" of the program representation. In particular, many locality-oriented analyses and optimizations are most naturally effective when applied to multidimensional index spaces and data structures. To that end, languages lacking multidimensional data structures, or compilers that aggressively normalize to 1D representations, undermine such optimization strategies. Expression of computations in their natural dimensionality and maintenance of that dimensionality during compilation are key.

  - A common theme in many promising language- and library-oriented approaches to locality is to express distribution- and/or locality-oriented specifications in a program's variable and type declarations rather than scattering it throughout the computation. Since locality is a cross-cutting concern, this minimizes the amount of code that needs to change when the mapping of the program's constructs to the hardware must. The compiler and runtime can then leverage the locality properties exposed in these declarations to customize and optimize code based on that information.

  - Isolate cross-cutting locality concerns. Locality — data layout and distribution — is fundamentally more difficult than parallelization because it affects all the code that touches the data.

- SOLUTIONS

  - There is a lot of consensus around multidimensional data/partitioning/slicing, and how to iterate over them (generators, iterators) - parameterisation of layouts.

  - There is potential to expose polyhedral concepts to the programmer/IR, and to pass them down to the backend (eg Chapels domains, mappings)

  - The back-end model for the target for code generation and implementation of models/compilers is missing - for portable implementation, for interoperability

  - While we can do a lot with libraries and template metaprogramming, there is a compelling case for a language-based approach

DISAGREEMENTS

- No consensus on the requirements on intermediate representations and runtime systems

- There was disagreement within the workshop attendees about the extent to which a language's locality-specification mechanisms should be explicit ("allocate/run this here") vs. suggestive ("allocate/run this somewhere near-ish to here, please") vs. locality-oblivious or automatic ("I don't want to worry about this, someone else [the compiler / the parallel expert] should figure it out for me."). This disagreement is arguably an indication that pursuing multiresolution features would be attractive. In such a model, a programmer could be more or less explicit as the situation warrants; and/or distinct concerns (algorithm vs. mapping) could be expressed in different parts of the program by programmers with differing levels of expertise.

- Language-based abstractions are central to tackling locality management, but two fundamentally different design philosophies have emerged that lead to very different implementation strategies – An explicit *"do what I say"* approach that is covered in subsection 5.3.2 and an implicit *"do what I mean"* approach that is described in subsection 5.3.3. More research is required to tease out the benefits and challenges of implementing each of these approaches in practice because the strategies differ on a very fundamental level.

- Explicit programmer control over locality is essential, but the expression of locality needs to be portable across machines. The naive interpretation is that "explicit control" is synonymous with explicitly mapping a particular task to specific core. However, in such a case, machine portability would be compromised. The loose interpretation of "explicit control" led to misunderstandings and requires more specific definition. For the purpose of these discussions, "explicit control" refers primarily to the interface provided to the performance layer (the "do what I say" layer for expert programmers), and is not intended to be exposed to the "do what I mean" layer for non-expert programmers.

5.3.1 Multiresolution Tools

The power of a language-based attack on locality is that we can support abstractions that can really help.

A key distinction to be understood and confronted is the tension between tools that control how code is executed (perhaps with the help of powerful abstractions), and tools that automatically make good choices, based on information provided by the programmer:

$$\underbrace{\text{"do what I say"}}_{\text{"but tell me only once"}} \quad vs \quad \underbrace{\text{"do what I mean"}}_{\text{"and let me take it from here"}}$$

Vigorous debate on this question led to the conclusion that there is a clear need for both - in fact, a need for a spectrum of levels of control. A key feature of any automated solution is that it can inter-operate with explicit control.

5.3.2   Partition Data *vs* Partition Computation

One philosophy in building abstractions for locality is to support abstractions for **explicit partitioning of the data**. The idea is that this is done in one place, localizing the programmer's specification of partitioning policy to one place for each data structure. The partitioning and distribution of computation is derived, by the compiler, from the data partitioning. By making the partitioning of data explicit the computation can operate on a local view of each data partition. It has the advantage of reducing the amount of complex reasoning that must be performed by the runtime to correctly associate the computation with the relevant data, and also makes communication cost more visible to the programmer.

However, on the negative side, explicit control could lead to composition challenges if the data layout choices are inconsistent. Data partitioning and distribution choices may turn out to be inconsistent when computations combine data from different sources, or when operations are composed. Thus, this approach naturally leads to programmers being required to make communication explicit.

Data partitioning fully constrains partitioning of computation if the implementation is confined to a uniform placement rule, like "owner computes" - the idea is that the execution of each assignment's RHS is determined by the placement of the location it is being assigned to.

In contrast, **explicit partitioning and distribution of the computation** is attractive as it is a local decision for example, for each parallel loop separately. The distribution, and movement, of data is then automatically derived by the compiler. Because the computational code is independent of partitioning it necessarily has a global view of the data.

Explicit partitioning of computation naturally leads to data movement being hidden - the communication cost of producing data with one distribution, while using in with another, is an implicit consequence of where the computation has been placed. There are also problems with other ways of combining software components - for example, with nested parallel loops and loops that call parallel functions.

5.3.3   Compositional Metadata

Communication and locality are a consequence of the dependences and reuses that arise when software components are composed. Thus, communication and locality are fundamentally non-compositional - they belong to the composition, not to the component. To make the idea of software components work - a key foundation for abstraction - we need to characterize each component in a way that allows the dependences to be derived when the component is combined with others.

This idea leads to a "do what I mean" model, where computational partitioning, data distribution and data movement can be derived automatically. The programmer does not, then, have explicit control of what happens where, or where communication will occur. However, programmers can reason about the communication cost of data accesses — and profiling tools can apportion communication costs to data accesses. This approach is advantageous because it offers more flexibility to the runtime in making choices about how to map data to the underlying hardware, but this also requires a lot of intelligence on the part of the runtime system to make cogent decisions.

5.4   Research Plan

The consensus from the meeting was that the major research challenges begin with:

- **Getting the abstraction right:** We need to design abstractions and representations that (1) achieve a separation of concerns, so that experts at algorithmic levels can develop sophisticated higher-level methods independently from experts at lower-level performance-focused levels developing sophisticated implementation techniques. We need to design abstractions that enable programmers to reason at an appropriate level for their task, and their expertise.

- **Multiresolution:** We need to support automatic implementation of high-level, abstract models. We also need tool and language support for explicit management of performance issues, such as partitioning and movement of data. We need explicit mechanisms for programmers to work at multiple such levels in a fully supported way.

- **Generality beyond multidimensional arrays:** Multidimensional arrays, structured meshes and regular data structures are important and there is huge scope for more sophisticated tools to support them - but richer data structures offer enormous scope, particularly for more ambitious methods, and more complex applications. We need to design tools, languages and abstractions that support hierarchical, hybrid and unstructured meshes, adaptivity in various forms, sparse representations and graph-based data.

The meeting brought together researchers pursuing a variety of directions; we highlight the following as perhaps the most promising:

- **Language design and implementation:** There is a clearly-demonstrated opportunity for languages in which locality is a first-class concept. Doing so opens up opportunities for expressivity, performance and enhanced correctness.

- **Compilation, code synthesis and multistage programming:** Tools that generate code have proven enormously powerful - but are often built in ad-hoc ways. There is potential to support code synthesis, runtimecode generation, and multistage programming as a first-class language feature. Code generation allows static scheduling to be combined flexibly with dynamic scheduling. Code generation allows high-level abstractions to be supported in the *generator*, thus avoiding runtime overheads.

- **Programming languages theory, in particular type systems:** How can we have polymorphic collection types without overcommitting to storage layout? Can we use types to track sharing, transfer of ownership, uniqueness, protocols? There is promising work in all of these areas.

- **Close engagement with leading science code communities and their users:** There is a long tradition of tools being developed in isolation from serious use-cases, or deeply embedded within just one, narrow application context. The most promising strategy must be to develop tools in a context that allows a broad class of application engagements beyond toy, or benchmark, problems.

- **Polyhedral methods:** The polyhedral model provides a uniquely powerful model for describing scheduling and data layout. There are huge opportunities to bring this power to bear in more general and more complex contexts.

Chapter 6

Data Locality in Runtimes for Task Models

Runtimes for task models enable a problem-centric description of an application's parallelism while hiding the details of task scheduling to complex architectures from the programmer. This separation of concerns is probably the most important reason for the success of using runtime environment systems for task models. It enables developers to "taskify" their applications while focusing on the scientific algorithms they are most familiar with. Programmers delegate all responsibilities related to efficient execution to the task scheduling runtime thereby achieving higher productivity and portability across architectures. The initial niche of these task model oriented runtimes was the dense linear algebra community [12, 88] upon which many scientific fields depend on for high performance computing. More recently, it has been extended to other scientific domains such as computational astronomy [21], fluid-structure interactions [66] and N-body problems [63, 2].

Runtimes for task models can be roughly divided into two groups:

Task-parallel runtime In a task-parallel runtime (e.g., Cilk [8], Intel TBB task_groups [49], OpenMP 3.0 tasks [71]), created tasks are ready to be executed in parallel as soon as they are created. The data dependencies in these runtimes are often expressed through parent-child states (for instance, due to the fork-join paradigm). Child states will inherit resolved dependencies from parent state(s) and, therefore, no dependence discovery is necessary for these runtimes.

Task-dataflow runtime In a task-dataflow runtime (e.g., OmpSs [7], StarPU [5], QUARK [55], Super-Matrix [20], OpenMP 4.0 tasks [71], Intel TBB graph parallelism [49], PaRSEC [13], or OCR [69]), instantiated tasks may not be immediately ready to execute and may depend on data generated from previous tasks. In these runtimes, data dependencies must be resolved prior to executing tasks.

When it comes to parallel efficiency, runtimes for task models usually provide decent performance. However, due to their dynamic or non deterministic behaviors, wrong scheduling decisions at runtime may considerably increase time to solution. For instance, many task-based runtimes make use of distributed queues for performance purposes, which enable local task scheduling but may come at the price of introducing load imbalance. In such runtimes, **work stealing** is the mechanism by which idle cores/threads/workers obtain tasks to execute and, by the same token, improves load balancing. When tasks are "stolen", the working set of those tasks is also frequently stolen, i.e., a work stealing operation usually results in data migration.

In fact, an oblivious work stealing strategy can further exacerbate the overhead of data motion, which may hide the existing benefit of data locality at the first place. To deal with such issues, it is necessary to make the runtime aware of high level data access pattern information so that it can perform locality-aware scheduling decisions. In this chapter, we explore the proposition that task-based programming models and the runtime system should offer a systematic way to express locality information, while still maintaining high productivity for end users.

6.1 Key Points

We now briefly discuss the key issues of data locality in task-based systems.

6.1.1 Concerns for Task-Based Programming Model

For both task-parallel and task-dataflow runtimes, the programmer and/or the runtime must consider two issues: **a)** the runtime's scheduling mode, and **b)** finding the appropriate task granularity.

a) Runtime Scheduling Mode

Previous efforts to optimize and improve the performance of a given application have always involved tuning an application to a particular machine, for example by optimizing for cache size, memory hierarchy, or the number of cores. Moving forward, this approach is untenable as machine variability will only increase therefore making a static decision impractical. At the same time, static or deterministic scheduling enables offline data locality optimizations but cannot deal with runtime hazards such as load imbalance issues. Two levels of scheduling (static and dynamic) have shown promising results by proposing an interesting trade off between concurrency and data locality [29].

    Finding the assignment of a set of fixed-time tasks to a set of processors that minimizes the makespan is a hard computational problem for which no fast optimal algorithm exists for more than two processors. In a shared memory multiprocessor, the assignment itself perturbs the execution time for each task due to resource sharing. This makes this already complex problem even harder. In practice, runtimes rely on simple heuristics to generate decent scheduling policies. Depending on the tasking semantics this involves different amounts of work. In task-parallel runtimes, scheduling involves fetching a task from the ready queue(s) or running the work-stealing loop if there are no ready tasks in the queues. As long as work stealing is minimized, this scheme features low overhead and generates locality efficient execution for codes optimized for locality on the sequential path. In task-dataflow runtimes, there are two scheduling levels **a)** resolving dependencies to find ready tasks and **b)** scheduling ready tasks to workers. The latter is in general performed as in task-parallel schemes. Task-dataflow schemes can improve performance by weakening synchronization points. However, since tasks may be inserted in the task queues out of order, the inter-task locality is often not exploited as efficiently [75].

    Extending these runtimes to perform locality-aware schedules in the general case is a complex task. First of all, how to provide locality information to the runtime is still an open problem. In the optimal case, the runtime should have a global notion of the application's parallel structure and data access pattern. However, such information can not generally be extracted automatically, and it is also a difficult task for the programmer. Explicitly stating information on task inputs and outputs allows the runtime to analyze the data reuse of a window of tasks. On the other hand, such optimizations often trade off concurrency for locality. Spending too much time in the runtime analyzing tasks instead of scheduling them can result in performance loss. Any information that a runtime takes into account for locality should still result in a quick scheduling decision.

Impact of work stealing In task-based systems, work stealing is a common scheduling technique used primarily for load-balancing. Work stealing can be optimized to exploit locality across tasks. If sets of tasks can constructively share the cache then limiting the work stealing to the tasks within the set will limit the working set migration to the private caches (shared caches will not be affected). Parallel-depth-first schedulers attempt to constructively share the cache by scheduling the oldest task in sequential order in the set of cores sharing the cache and only resorting to global work stealing when the task queues become empty [70].

    In general we want to minimize the number of steals. This works well if the application can quickly be partitioned into almost equivalent sets of work that can then proceed independently. This is generally the case for divide-and-conquer parallelism, but for more general approaches such as loop-style parallelism (i.e., tasks generated inside a *for* loop) using work stealing to keep all cores busy is usually not efficient. This happens because distributing N tasks generated by a N-iteration loop will require exactly N work steals. If N is larger than the number of processors, then a work-sharing partitioning of the loop can be more efficient.

A big challenge is how to communicate the hierarchical data properties of an application to the runtime so that they can be exploited to generate efficient schedules. Classical random work stealers (e.g. Cilk-like [8]) do not exploit this. Socket-aware policies exist (e.g. Qthread [70]) that perform hierarchical work stealing: **a)** first among cores in a socket and **b)** then among sockets. Some programming models expose an API that allows programmers to specify on which NUMA node/socket a collection of tasks should be executed (e.g. OmpSs [7]). Configurable work stealers which can be customized with *scheduling hints* have also been developed [91]. Finally, a more extreme option is to allow the application programmer to attach a custom work stealing function to the application [67]. How to effectively specify this information in a programmer-friendly way is an open topic of research.

b) Task Granularity

Large tasks (i.e., coarse-grained tasks) can result in load imbalance at synchronization points. Reducing task sizes is a common method to improve load balance. On the other hand, the granularity of tasks directly influences scheduling overheads. Too fine-grained tasks increase the amount of time spent inside the runtime performing jobs such as scheduling and work stealing. Task granularity impacts locality since larger tasks also have bigger memory footprints. Task sizes are not only a trade-off between parallelism and runtime overheads, but should also be set in order to efficiently exploit the memory hierarchy. Last level shared caches provide larger storage that can be exploited particularly well when groups of tasks share some of their inputs. This is called constructive cache sharing [24].

Finding the best granularity is a complex problem since all three metrics to be optimized (overheads, load balance, data locality) are connected. Since the optimum configuration may depend on the input, auto-tuning techniques are a promising approach. Enabling auto-tuning involves programmer effort to find ways to partition data sets in a parametric way, allowing the runtime to tune the task size.

6.1.2 Expressing Data Locality with Task-Based systems

Nested parallelism: Nested or recursive algorithmic formulation is a well-known technique to increase data reuse at the high levels of the memory hierarchy and therefore, to reduce memory latency overheads. This often requires slight changes in the original algorithm. At the same time, to witness the performance benefits of such formulation, it is critical to make sure the resulting nested algorithm is correctly mapped into the underlying architecture. Task-parallel runtimes should exploit recursive formulations of an algorithm by scheduling nested parallel tasks to processing units, which share some level of caches. This should not prevent task-parallel runtimes to still perform work stealing in case load imbalance is detected as long as it does not hinder the overall performance.

Distance-aware scheduling: Reducing data movement needs to happen at all levels of the system architecture to be effective: from the single CPU socket within a multi-socket shared-memory node up to multiple distributed-memory nodes linked through high performance network interconnect. This bottom-up approach highlights the importance of improving data locality within the socket itself and foremost. Therefore, task-parallel runtimes need to provide an abstraction to algorithm developers to facilitate the scheduling of several tasks, which own common data dependencies, on the same socket. Similarly to the nested parallelism technique, in case of load balancing issues, work should be first stolen from the closest CPU sockets. This is paramount especially for NUMA architectures.

Pipelining across Multiple Computational Stages: Many numerical algorithms are often built on top of optimized basic blocks. For instance, dense Eigensolvers require three computational stages: matrix reduction to condensed form, iterative solver to extract the eigenvalues and back transformation to get the associated Eigenvectors. Each stage corresponds to an aggregation of several computational kernels, which may already be optimized independently for data locality. However, pipelining across multiple subsequent basic blocks (e.g., thanks to look ahead optimizations) may mitigate the benefits of data locality since task-parallel runtimes may not have the global directed acyclic graph before execution. There is, therefore, an urgent need to express and maintain data locality of the overall directed acyclic graph, without delaying the execution of tasks belonging to the critical path. While this is still an open research problem in the

context of distributed-memory systems, this issue is handled on shared-memory machines through specific data locality flags, provided by the user.

6.2 State of the art

There are many software projects on task-parallel runtimes. We list a number of them here but it is by no means a complete list.

Nanos++ The Nanos++ dynamic runtime system interfaces the task-parallel application with the underlying hardware architecture. A core component of the runtime is in charge of handling data movement and ensuring coherency, so that the programmer does not need to deal with memory allocation and movements. Another core component are the scheduling policies, which contain a newly introduced policy for mitigating the work stealing overhead, called the distance-aware work stealing policy. The OmpSs programming model, a high-level task-based parallel programming model, is powered by the Nanos++ runtime. It relies on compiler technology and supports task parallelism using synchronization based on data-dependencies. Data parallelism is also supported by means of *services* mapped on top of its task support. OmpSs also supports heterogeneous programming, as reflected in the modular support for different architectures in Nanos++ (SMP, CUDA, OpenCL, simulators such as TaskSim, etc.). Nanos++ also provides instrumentation tools to help identifying performance issues.

Quark QUARK (QUeuing And Runtime for Kernels) provides a library that enables the dynamic execution of tasks with data dependencies in a multi-core, multi-socket, shared-memory environment. QUARK infers data dependencies and precedence constraints between tasks from the way that the data is used, and then executes the tasks in an asynchronous and dynamic fashion in order to achieve a high utilization of the available resources. The dependencies between the tasks form an implicit DAG, however this DAG is never explicitly realized in the scheduler. The structure is maintained in the way that tasks are queued on data items, waiting for the appropriate access to the data. The tasks are inserted into the scheduler, which stores them and executes them when all the dependencies are satisfied. In other words, a task is ready to be executed when all parent tasks have completed. The execution of ready tasks is handled by worker threads that simply wait for tasks to become ready and execute them using a combination of default task assignments and work stealing.

SuperMatrix Similarly to QUARK, SuperMatrix is a runtime system that mainly parallelizes matrix operations for SMP and/or multi-core architectures. This runtime system demonstrates how code described at a high level of abstraction can achieve high performance on such architectures, while completely hiding the parallelism from the library programmer. The key insight entails viewing matrices hierarchically, consisting of blocks that serve as units of data where operations over those blocks are treated as units of computation. The implementation transparently enqueues the required operations, internally tracking dependencies, and then executes the operations utilizing out-of-order execution techniques inspired by superscalar microarchitectures. This separation of concerns allows library developers to implement algorithms without concerning themselves with the parallelization aspect of the problem. Different heuristics for scheduling operations can be implemented in the runtime system independently of the code that enqueues the operations.

PaRSEC PaRSEC is a generic framework for architecture aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. Applications can be expressed as a Direct Acyclic Graph of tasks with labeled edges designating data dependencies. DAGs are represented in a compact problem-size independent format that can be queried on-demand to discover data dependencies in a totally distributed fashion. PaRSEC assigns computation threads to the cores, overlaps communications and computations and uses a dynamic, fully-distributed scheduler based on architectural features such as NUMA nodes and algorithmic features such as data reuse. The framework includes libraries, a runtime system, and development tools to help application developers tackle the difficult task of porting their applications to distributed memory, highly heterogeneous and diverse environments.

StarPU StarPU is a runtime system that schedules tasks onto accelerator-based platforms. It is meant to be used as a back-end for parallel language compilation environments and high-performance libraries. The two basic principles of StarPU is firstly that tasks can have several implementations, for some or each of the various heterogeneous processing units available in the machine, and secondly that transfers of data pieces to these processing units are handled transparently by StarPU. Thanks to auto-tuning facilities, StarPU transparently predicts execution time and data transfer overhead. This permits StarPU's dynamic scheduler to avoid load imbalance while enforcing data locality to reduce the pressure on the memory subsystem, which is a critical resource for accelerator-based platforms.

The Open Community Runtime (OCR) OCR is a task-dataflow programming model and research runtime aimed at determining what works and what does not for large scale task-dataflow programming models and runtimes. It provides a simple low-level API with which programmers or higher-level programming languages or abstractions can inform a runtime system of tasks, data used by the tasks (called data-blocks) as well as dependencies between them.

From a data locality perspective, all aforementioned dynamic runtime systems provide mechanisms to limit data motion. This is achieved in one of two ways. One approach is to do it internally through scheduling policies, in a transparent-to-the-user fashion. This method is typical for runtimes using source-to-source compiler technology since dependencies are discovered ahead of the actual execution and therefore decent scheduling decisions can be made to preserve data locality. An alternative approach is via scheduling optimization flags, a method which may require user intervention to prevent moving excessively data between worker threads. This method is representative of dynamic scheduling libraries, which discover dependencies only at runtime.

6.3   Discussion

Areas of Agreement

- **Static partitioning will become increasingly hard:** The performance and energy characteristics of today's hardware are difficult to predict; the unpredictability of a hit or miss in a cache, the variable latencies introduced by branch mis-predictions, etc. are only some of the factors that contribute to a very dynamic hardware behavior. This trend will only accelerate with future hardware as near threshold voltage (NTV) and aggressive energy management increase the level of performance variability. Architectures, unfortunately, do not provide any guarantees on execution time or energy consumption, impeding the task of time and energy estimation.

  In light of this, toolchains will become increasingly unable to statically partition and schedule code to efficiently utilize future parallel resources. Hardware characteristics will vary dynamically and we therefore cannot do without a dose of dynamism in the runtime: dynamic task based programming models need to be a part of the solution.

- **Optimal Granularity:** Task-based programming models rely on the computation being split into chunks called "tasks". The partitioning of code into tasks implicitly partitions data into the corresponding per-task working sets. The optimal size of tasks (the granularity) is difficult to determine as it needs to balance the overheads of the task-based runtime system with the need to expose sufficient parallelism, while making efficient use of processor caches, which depends on the size and access pattern of the working set. A static granularity will be sub-optimal for future machines.

- **Separation of Concerns vs Co-design:** One of the main reasons behind the success of these task-based dynamic runtime systems is their capacity to abstract the underlying hardware complexity. This separation of concerns between scheduling and algorithmic development allows end users to focus primarily on designing their numerical algorithms sequentially while adding parallelism features at a later stage. This separation of functionality and performance enhances productivity. However, moving forward with exascale systems, this free lunch may come to an end if one wants to exploit systems with billion of cores efficiently. Future programming environments and runtimes will need to target stronger

software-hardware co-design. Algorithms based on recursive formulations are a good example of such co-design, as divide-and-conquer approaches can often express parallelism while implicitly preserving data locality. The memory hierarchy at all levels of the system can then be exploited and further performance improvement can be expected in case of data reuse.

Areas of Disagreement

- **Level of Standardization:** There is an agreement on the fact that a standardization needs to happen for task-based programming model but there is a disagreement as to the level at which this standardization should happen. One option is to standardize the APIs at the runtime level in a way similar to the Open Community Runtime (OCR). Another option is to standardize the interface of the programming model, like OpenMP or OmpSs do. Currently there is no clear reason to decide for a particular scheme, so both approaches are being actively researched.

- **Expression of Granularity:** Another area of disagreement is how to deal best with the expression of granularity; specifically, is it better for the programmer to break-down tasks and have a runtime system re-assemble them if needed or is it preferable to have the programmer express coarse grained tasks and data and allow the runtime system to break them down further. The latter approach has been used successfully in runtimes targeted to problems that are recursively divisible. The former approach would require some sort of "recipe" for the runtime to be able to stitch smaller tasks or chunks of data into larger ones. There is a debate as to which approach is simpler and more likely to yield positive results.

6.4   Research Plan

There are several areas that would benefit from further research in task-based models and runtime systems: **a)** a better understanding of the broader performance implications of task-based systems, **b)** better understanding of tools for debugging task-based models, and **c)** abstractions for the programmer to be able to encapsulate and express *hints* regarding task granularity, data locality and other information not already captured by the expression of dependencies.

Apart from these three main areas of research, detailed below, the community would also greatly benefit from experiments showing the benefits and downsides of task-based programming models and runtimes.

6.4.1   Performance of task-based runtimes

Wall-clock execution time has long been the golden standard of performance metrics. This metric, however, is no longer sufficient in understanding the performance of task-based runtime systems especially in how they relate to data-locality.

Task-based runtimes require that the programmer relinquish control of aspects that he has traditionally controlled: scheduling, data placement, etc. For these runtimes to be accepted, programmers need to be convinced that the runtime systems are not making "stupid" decisions; in other words, that by delegating some of the traditional aspects of parallel programming, the programmer is not sacrificing inordinate amounts of performance. To that end, metrics that capture changes to data locality, load balancing, etc. need to be developed. Separately, metrics showing the specific benefits of task based systems also need to be developed: programmer productivity, better resiliency, etc.

6.4.2   Debugging tools

Task-based programming models are notoriously difficult to reason about and debug given that the parameters specified by the programmer to constrain execution (dependencies) purposefully allow for a wide range of execution options. Certain task-based runtime systems, which allow the dynamic construction of the task-graph (such as OCR), only exacerbate these problems. Tools allowing the programmer to understand the execution of a task-based program need to be developed.

These tools will need to cover two broad areas:

- Information on the execution flow of the application in terms of the tasks and data-elements defined by the user;

- Information about the mapping of those tasks and data-elements to the computing and memory resources.

The former area will help the user in determining whether the application is executing correctly while the latter will help in identifying performance issues due to improper choices in the runtime (bad scheduling, bad locality, etc.). Both areas will be important for different audiences: the former will be more important for application developers while the latter will prove crucial for runtime system developers.

6.4.3 Hint framework

Task-based programming models do a good job requiring very little information from the programmer: what are the tasks, the data items and the dependencies between them. While this is sufficient to produce a correct execution of the program, it may not be sufficient to produce an efficient one. As previously mentioned, task-based runtimes that are oblivious to locality constraints will perform badly.

It is therefore important to understand the type of information that a programmer can easily supply and how best to communicate it through the programming model to the runtime system. Task-based runtimes already know the dependence structure between tasks; the research question is what other information they could make use of that the programmer can easily provide to improve their scheduling and placement heuristics.

Chapter 7

System-Scale Data Locality Management

As the complexity of parallel computers rises, applications find it increasingly more difficult to access data efficiently. Today's machines feature hundreds of thousands of cores, a deep memory hierarchy with several cache layers, non-uniform memory access with several levels of memory (e.g., flash, non-volatile, RAM), elaborate topologies (both at the shared-memory level and at the distributed-memory level using state-of-the-art interconnection networks), advanced parallel storage systems, and resource management tools. With so many degrees of freedom arranging bytes becomes significantly more complex than computing flops, and the expectation of decrease in memory per core in the coming years will only worsen the situation

To address the data locality issue system-wide, one possible approach is to examine and refactor the application ecosystem, i.e., the execution characteristics, the way in which the resources are used, the application's relation with the topology, or its interaction with other executing applications. This approach implies a strong need for new models and algorithms, or at least significantly refactored ones. It also points to the need for new mechanisms and tools for improving (1) topology-aware data accesses, (2) data movements across the various software layers, and (3) data locality and transfers for applications.

7.1   Key points

There is rich literature with evidence of tremendous efforts expended on optimizing various applications statically. The approaches mentioned in the literature include data layout optimizations, compilation optimization, and parallelism structuring. These approaches in general have been extremely effective however, there are several additional factors that cannot be optimized for prior to execution, but, can have dramatic impact on the application performance beyond the static optimizations. Among these factors are:

- the configuration of allocated resources for the specific run,

- the network traffic induced, or any other interference caused by other running applications,

- the topology of the target machine,

- the relative location of the data accessed by the application on the storage system,

- dependencies of the input on the execution,

and many more.

Often these runtime factors are orthogonal to the static optimizations that can be performed. For instance, recent results [27, 60] show that a non-contiguous allocation can reduce the performance by more than 30%. However, a batch scheduler cannot always provide a contiguous allocation and even if it could, the way processes are mapped to the allocated resources still has a big impact on the performance [23, 28, 45, 50]. The reason is often the complex network and memory topology of modern HPC systems and that some pairs of processes exchange more data than some other pairs.

Furthermore, energy constraints imposed by exascale goals are altering the balance of interconnect capabilities, reducing the bandwidth to compute ratio while increasing injection rates. This shift is causing

fundamental reconsideration of the BSP programming model and interconnect design. One of the leading contenders for a new interconnect is a multi-level direct network such as Dragonfly [4, 56]. Such networks are formed from highly-connected parts, placing every node within a few hops of all other nodes in the system. This may benefit unstructured communications that often occur in graph algorithms, but limited connections between parts can be bottlenecks for structured communication patterns [76]. At the node level, a promising approach for fully utilizing higher core counts on next-generation architectures is over-decomposed task parallelism [52], which will stress the interconnect in ways different from the traditional BSP model.

In order to optimize system-scale application execution we need models of the machine at different scales. We also need models of the application and its algorithms, and tools to optimize the execution within the whole ecosystem. Literature provides many models and abstractions for writing parallel codes that have been successful in the past. However, these models alone may not be sufficient for scaling in future applications due to the data traffic and coherence management constraints [78]. Current models and abstraction are more concerned with computations than with the cost incurred by data movement, topology and synchronization. It is important to provide new hardware models to account for these phenomena as well as abstractions to enable the design of efficient topology-aware algorithms and tools.

A hardware model is needed to understand how to control locality. Modeling the future large-scale parallel machines will require work in the following directions: (1) ability to better describe the memory hierarchy, (2) a way to provide an integrated view of nodes and the network, (3) inclusion of qualitative knowledge such as latencies, bandwidths, or buffering strategies, and (4) providing ways to express the multi-scale properties of the machine.

Applications need abstractions allowing them to express their behavior and requirement in terms of data access, locality and communication. For this, we need to define metrics to capture the notions of data access, affinity, and network traffic. The MPI standard offers the process topology interface that allows an application to specify the dataflow between processes [43]. While this interface is a viable first step, it is limited to BSP-style MPI applications. More general solutions are needed for wider coverage. To optimize execution at system scale, we need to extract the application requirements from the application models and abstractions and apply them to the mechanisms, tools and algorithms provided by the network model. With enough such information available it becomes feasible to perform several optimizations such as: improving storage access, mapping processes onto resources based on their affinity [28, 44, 45, 80], selecting resources according to the application communication pattern and the pattern of the currently running applications. It is also possible to couple allocation and mapping. Appropriate abstraction In the context of storage bring in the possibility of exploiting an often overlooked aspect of locality. Not only is it possible for the applications to request that the data and execution are local to each other, it is also possible to spread out data and corresponding execution to take advantage of parallelism inherent in the system.

7.2 State of the Art

Various approaches for topology mapping have been developed: TreeMatch [51], provides mapping of processes onto computing resources in the case of a tree topology (such as current NUMA nodes and fat tree topologies). LibTopoMap [45] addresses the same problem as TreeMatch but for arbitrary topology such as torus, grid, or completely unstructured networks. The general problem is NP-hard and no good approximation schemes are known, thus forcing developers to rely on various heuristics. However, several specialized versions of the problem can be solved near-optimal in polynomial time, for example, mapping Cartesian topologies to Dragonfly networks [76].

Topology mapping can also be seen as a graph embedding problem where an application graph is embedded into a machine graph. Therefore, graph partitioners such as Scotch [33] or ParMetis [54] could provide a solution, though they may require more precise information than more specialized tools and the solutions are not always good [51]. Zoltan2 [9, 28] is a toolkit that, after processes are allocated to an application, can map these processes to resources based on geometric partitioning where processes and computing units are identified by coordinates in a multidimensional space

Hardware Locality (hwloc) [38, 47] is a library and a set of tools aimed at discovering and exposing the hardware topology of machines, including processors, cores, threads, shared caches, NUMA memory nodes and I/O devices. Netloc [39, 68] is a network model extension of hwloc to account for locality requirements

of the network, including the fabric topology. For instance, the network bandwidth and the way contention is managed may change the way the distance within the network is expressed or measured.

Modeling the data-movement requirements of an application in terms of network traffic and I/O can be supported through performance-analysis tools such as Scalasca [37]. It can also be done by tracing data exchange at the runtime level with a system like OVIS [73, 85], by monitoring the messages transferred between MPI processes for instance. Moreover, compilers, by analyzing the code and the way the array are accessed can, in some cases, determine the behavior of the application regarding this aspect.

Resource managers or job scheduler, such as SLURM [95], OAR [16], LSF [97] or PBS [41] have the role to allocate resources for executing the application. They feature technical differences but basically they offer the same set of services: reserving nodes, confining application, executing application in batch mode, etc. However, none of them is able to match the application requirements in terms of communication with the topology of the machine and the constraints incurred by already mapped applications.

Parallel file systems such as Lustre [14], GPFS [83], PVFS [17], and PanFS [90] and I/O libraries such as ROMIO [86], HDF5 [40], and Parallel netCDF [62] are responsible for organizing data on external storage (e.g., disks) and moving data between application memory and external storage over system network(s). The "language" used for communicating between applications and libraries and underlying parallel file systems is the POSIX file interface [48]. This interface was designed to hide underlying topological information, and so parallel file system developers have provided proprietary enhancements in some cases to expose information such as the server holding particular data objects, or the method of distribution of data across storage devices. Additionally, work above the parallel file system is beginning to uncover methods of orchestrating I/O *between* applications [30]. This type of high-level coordination can assist in managing shared resources such as network links and I/O gateways, and is complementary to an understanding of the storage data layout itself.

7.3 Discussion

To address the locality problem at system scale, several challenges must be solved. First, scalability is a very important cross-cutting issue since the targets are very large-scale, high-performance computers. On one hand, applications scalability will mostly depends on the way data is accessed and locality is managed and, on the other hand, the proposed solutions and mechanisms have to run at the same scale of the application and their inner decision time must therefore be very short.

Second, it is important to tackle the problem for the whole system: taking into account the whole ecosystem of the application (e.g., storage, resource manager) and the whole architecture (i.e., from cores to network). It is important to investigate novel approaches to control data locality system-wide, by integrating cross-layer I/O stack mechanisms with cross-node topology-aware mechanisms.

Third, most of the time, each layer of the software stack is optimized independently to address the locality problem with the result that sometimes there are conflicting outcomes. It is, therefore, important to observe the interaction of different approaches with each-other and propose integrated solutions that provide a global optimizations across different layers. An example of such an approach is mapping independent application data accesses to a set of storage resources in a balanced manner, which requires an ability to interrogate the system regarding what resources are available, some "distance" metric in terms of application processes, and coordination across those processes (perhaps supported by a system service) to perform an appropriate mapping. Ultimately, the validation of the models and solutions to the concerns and challenges above will be a key challenge.

7.4 Research Plan

To solve the problems described above, we propose co-design between application communication models, specific network structures, and algorithms for allocation and task mapping.

For instance, we currently lack tools and APIs to connect the application with the mapping of its components. While Process topologies in MPI offers a facility to communicate the application's communications pattern to the MPI library, there is no such option for the reverse direction. Therefore, the applications have no way of customizing themselves to the underlying system architecture even when such potential exists.

Another issue is that the process topology can only be exploited at the MPI level. Lower levels; with task mapping for threaded/hybrid programming mechanisms for querying the topology are available (e.g., hwloc) but there are no automatic frameworks supporting data mapping yet.

Moreover, process and data mapping is based on the notion of *affinity*. How to measure this affinity is still an open question. Different metrics (i.e., size of the data, number of messages, etc.) exist but they do not account for other factors that may have a role to play depending on the structure of the exchange (i.e., dilatation, congestion, number of hops). We currently lack the ability to confidently express what is required to be optimized within the algorithmic process. The effect of over-decomposition on application communication and mapping is also an open question.

We also need to develop hierarchical algorithms that will deal with different levels of models and abstraction in order to tackle the problem system at full scale in parallel. A global view of the system will also be needed to account for the whole environments (storage, shared resources, other running application, network usage, etc.)

Storage services must be adapted to expose a locality model as well. One approach would be to follow models being used to express topology in networks, and similarly techniques for balancing network traffic across network links might be adapted to assist in mapping data accesses to storage resources. That said, because data is persistent and may be used in the far future in ways the storage service cannot predict, the need for applications to explicitly control data layout on storage resources, based on knowledge of future requirements, is also apparent. These issues motivate an investigation of alternative storage service models, beyond current parallel file system models.

Chapter 8

Conclusion

The Programming Abstractions for Data Locality (PADAL) Workshop is intent on establishing the identity for a community of researchers who are bound together by the notion that data locality comes first as the primary organizing principle for computation. This paradigm shift from compute-centric towards data-centric specification of algorithms has upended assumptions that underpin our current programming environments. Parallelism is inextricably linked to data locality, and current programming abstractions are centered on abstractions for compute (threads, processes, parallel do-loops). The PADAL workshop gives voice to the groundswell of activity to explore abstractions for expressing parallelism in terms of data locality. The time has arrived to embrace data locality as being the anchor for computation. PADAL has identified a community that is actively exploring a wide-open field of new approaches to describing computation and parallelism in a way that conserves data movement. A number of these projects have produced working technologies that are rapidly approaching maturity During this early phase of development, it is crucial to establish research collaborations that leverage for commonalities and opportunities for inter-operation between these emerging technologies.

8.1 Priorities

The highest priorities moving forward are to;

- Define data-centric approaches to computation as a community.

- Foster development of an identity through continued workshops and interactions.

- Define a common research agenda that cuts across the different research areas in libraries, compilers/languages, runtime/task models, and system-scale implementations of data-locality centric programming systems.

Much research in this area (as with all emerging fields of research) has focused on rapidly producing implementations to demonstrate the value of data-centric programming paradigms. In order to get to the next level of impact, there is a benefit to formalizing the abstractions for representing data layout patterns and the mapping of computation to the data where it resides. It is our desire to create standards that promote interoperability between related programming systems and cooperation to ensure all technology implementations offer the most complete set of features possible for a fully functional programming environment. The only way to achieve these goals is for this community to organize, consider our impact on the design of the software stack at all levels, and work together towards the goal of creating interoperable solutions that contribute to a comprehensive environment.

8.2 Research Areas

The following key research areas are outlined in this report.

Data structures and layout abstractions to express and manage data layout : These are the overarching data layout abstractions that provide a compact description of data layout patterns. These patterns can be embedded in libraries and data structures for library-based building blocks for applications, and also form a baseline for higher-level language and compiler support.

Language and compiler support for data locality: Compiler and language support for those abstractions take best practices from library approaches and elevate them into language constructs benefit from broad community adoption of the overarching abstractions

Support for data locality in runtimes and tasks models: Many data layouts can be implemented statically, but the runtime system must be part of the solution for automating those choices for data sizes and layouts that are only known at runtime. For task models some decisions that balance load-balance against data-locality can be can be static, semi-static, or fully dynamic depending on context.

Addressing system-scale data locality management issues: The reigning abstraction for system-level resource management is that all resources at system level are equidistant and equal-cost to access, which ignores optimizations for affinity or locality. System-wide systems considerations present an area of research that speaks to the need to develop a new class of locality aware resource managers, locality aware communication libraries, or adding hooks to enable introspection and control of data locality.

8.3   Next Steps

Follow-on Workshops: In order to maintain our momentum and continue to organize this research community will require active participation in workshops and perhaps continued dialogue in mini-symposia and other research gatherings. The Lugano workshop is hopefully the first of a series. The organizers intend to schedule the next workshop in Berkeley, California for the Spring or Summer of 2015.

Define Research Agenda: As enumerated in section 8.2 four primary research areas have been identified and the introduction to this report has provided findings and recommendations for the next steps to carry out a research agenda in each area. As the next step, the community should identify leaders in each of these respective areas to continue community dialogue on advancing the recommendations and engaging in their funding agencies to describe the value proposition for investing in this research and to identify funding opportunities in related areas.

Interoperability: The underlying abstractions and data structures that were outlined by independent researchers showed much commonality in approach and abstractions despite differences in implementations. The commonality of approach points to an opportunity to create a common lower-level implementation for these myriad research products. Mature technologies should consider standards to support compatibility and interoperability across implementations and work towards a common underlying system software that can enable multiple user-facing implementations. Such design for interoperability and (where possible) standardization of lower level interfaces enables a diversity of the higher-level user-facing APIs.

It is crucial that these systems interoperate with runtimes systems to perform the mapping from data structures that describe data layout to optimal mapping of the application onto the underlying system fabric. To accomplish this goal, libraries, languages, and runtimes should offer common interfaces to the libraries and languages can introspect the system and interact with the system services that control data placement. This need for interoperation of system services and program description underpins the desire to get all four of these thrust areas to work together towards common abstractions or standards where possible. It also ensures a practical amount of effort on the part of HPC system manufacturers by tying to a common underlying system software infrastructure.

The PADAL workshop has identified several broad research areas that have common cause and related technologies and solutions. We have held the first of what we hope to be many workshops and meetings to further establish this research community and to establish points of research collaboration and sharing of ideas. We look forward to future collaborations and a continued progress in this emerging area of parallel programming environments research.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S.W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 248–259, June 2000.

[2] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-Based FMM for Multicore Architectures. *SIAM Journal on Scientific Computing*, 36(1):66–93, 2014.

[3] J.A. Ang, , R.F. Barrett, R.E. Benner, D. Burke, C. Chan, D. Donofrio, S.D. Hammond, K.S. Hemmer, S.M. Kelly, H. Le, V.J. Leung, D.R. Resnick, A.F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N.J. Wright. Abstract machine models and proxy architectures for exascale computing. Technical report, DOE Technical Report (joint report of Sandia Laboratories and Berkeley Laboratory), May 2014.

[4] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony. The PERCS High-Performance Interconnect. In *Proceedings of 18th Symposium on High-Performance Interconnects (Hot Interconnects 2010)*. IEEE, Aug. 2010.

[5] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency Computat. Pract. Exper.*, 23:187–198, 2011. (to appear).

[6] M. Baldauf, O. Fuhrer, M. J. Kurowski, G. de Morsier, M. Muellner, Z. P. Piotrowski, B. Rosa, P. L. Vitagliano, and M. Ziemianski D. Wojcik. The cosmo priority project 'conservative dynamical core' final report. Technical report, MeteoSwiss, October 2013.

[7] Barcelona Supercomputing Center. The OmpSs Programming Model. `https://pm.bsc.es/ompss`.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of PPoPP '95*. ACM, July 1995.

[9] E. G. Boman, K. D. Devine, V. J. Leung, S. Rajamanickam, L. A. Riesen, M. Deveci, and U. Catalyurek. Zoltan2: Next generation combinatorial toolkit. Technical Report SAND2012-9373C, Sandia National Laboratories, 2012.

[10] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, 2007.

[11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communnications of the ACM*, 54(5):67–77, 2011.

[12] G. Bosilca, A Bouteiller, A Danalis, M. Faverge, A Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A YarKhan, and J. Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1432–1441, May 2011.

[13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science and Engineering*, 15(6):36–45, 2013.

[14] Peter J. Braam. The lustre storage architecture. Technical report, Cluster File Systems, Inc., 2003.

[15] M.S. Campobasso and M.B. Giles. Effects of flow instabilities on the linear analysis of turbomachinery aeroelasticity. *Journal of Propulsion and Power*, 19(2):250–259, March 2014.

[16] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, pages 776–783. IEEE, 2005.

[17] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[18] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanovi?, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and O Fox. Sejits: Getting productivity and performance with selective embedded jit specialization, 2009.

[19] Bryan C. Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 47–56, 2011.

[20] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Supermatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[21] A. Charara, H. Ltaief, D. Gratadour, D. Keyes, A. Sevin, A. Abdelfattah, E. Gendron, C. Morei, and F. Vidal. Pipelining computational stages of the tomographic reconstructor for multi-object adaptive optics on a multi-gpu system. In *Proceedings of the 2014 ACM/IEEE Conference on Supercomputing*, Supercomputing, 2014.

[22] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[23] Hu Chen, Wenguang Chen, Jian Huang, Bob Robert, and H. Kuhn. MPIPP: an Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In Gregory K. Egan and Yoichi Muraoka, editors, *Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006*, pages 353–360. ACM, 2006.

[24] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of SPAA*, 2007.

[25] P. Colella, D. T. Graves, D. Modiano, D. B. Serafini, and B. van Straalen. Chombo software package for AMR applications. Technical report, Lawrence Berkeley National Laboratory, 2000. `http://seesar.lbl.gov/anag/chombo/`.

[26] OpenMP Standards Committee. Openmp 4.0 application program interface. `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`, July 2013.

[27] Y Cui, E Poyraz, J Zhou, S Callaghan, P Maechling, TH Jordan, L Shih, and P Chen. Accelerating cybershake calculations on the xe6/xk7 platform of blue waters. In *Extreme Scaling Workshop (XSW), 2013*, pages 8–17. IEEE, 2013.

[28] Mehmet Deveci, Sivasankaran Rajamanickam, Vitus J Leung, Kevin Pedretti, Stephen L Olivier, David P Bunde, Umit V Catalyürek, and Karen Devine. Exploiting geometric partitioning in task mapping for parallel computers. In *IPDPS*, PHOENIX (Arizona) USA, May 2014.

[29] Simplice Donfack, Laura Grigori, William D. Gropp, and Vivek Kale. Hybrid Static/dynamic Scheduling for Already Optimized Dense Matrix Factorization. In *IPDPS*, pages 496–507. IEEE Computer Society, 2012.

[30] Matthieu Dorier, Gabriel Antoniu, Robert Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *Proceedings of the International Parallel and Distributed Processing Symposium*, May 2014.

[31] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 2014.

[32] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. *SIGPLAN Not.*, 46(10):391–406, October 2011.

[33] F. Pellegrini. SCOTCH and LIBSCOTCH 5.1 *User's Guide*. ScAlApplix project, INRIA Bordeaux – Sud-Ouest, ENSEIRB & LaBRI, UMR CNRS 5800, August 2008. `http://www.labri.fr/perso/pelegrin/scotch/`.

[34] Oliver Fuhrer, Mauro Bianco, Isabelle Bey, and Christoph Schr. Grid tools: Towards a library for hardware oblivious implementation of stencil based codes. `http://www.pasc-ch.org/projects/projects/grid-tools/`.

[35] Karl Fürlinger, Colin Glass, Jose Gracia, Andreas Knüpfer, Jie Tao, Denis Hünich, Kamran Idrees, Matthias Maiterth, Yousri Mhedheb, and Huan Zhou. DASH: Data structures and algorithms with support for hierarchical locality. In *Euro-Par Workshops*, 2014.

[36] Michael Garland, Manjunath Kudlur, and Yili Zheng. Designing a unified programming model for heterogeneous machines. In *Supercomputing 2012*, November 2012.

[37] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[38] Brice Goglin. Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc). In *Proceedings of 2014 International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, July 2014.

[39] Brice Goglin, Joshua Hursey, and Jeffrey M. Squyres. netloc: Towards a Comprehensive View of the HPC System Topology. In *Proceedings of the fifth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2014), held in conjunction with ICPP-2014*, Minneapolis, MN, September 2014.

[40] HDF5. `http://www.hdfgroup.org/HDF5/`.

[41] Robert L Henderson. Job scheduling under the portable batch system. In *Job scheduling strategies for parallel processing*, pages 279–294. Springer, 1995.

[42] Berk Hess, Carsten Kutzner, David van der Spoel, and Erik Lindahl. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. *J. Chem. Theory Comput.*, 4(3):435–447, 2008.

[43] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Traeff. The Scalable Process Topology Interface of MPI 2.2. *Concurrency and Computation: Practice and Experience*, 23(4):293–310, Aug. 2010.

[44] Torsten Hoefler, Emmanuel Jeannot, and Guillaume Mercier. Chapter 5: An overview of process mapping techniques and algorithms in high-performance computing. In Emmanuel Jeannot and Julius Žilinskas, editors, *High Performance Computing on Complex Environments*, pages 65–84. Wiley, 2014. To be published.

[45] Torsten Hoefler and Marc Snir. Generic Topology Mapping Strategies for Large-Scale Parallel Architectures. In David K. Lowenthal, Bronis R. de Supinski, and Sally A. McKee, editors, *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04, 2011*, pages 75–84. ACM, 2011.

[46] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[47] hwloc. Portable Hardware Locality. `http://www.open-mpi.org/projects/hwloc/`.

[48] IEEE. *2004 (ISO/IEC) [IEEE/ANSI Std 1003.1, 2004 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. IEEE, New York, NY USA, 2004.

[49] Intel Corporation. Threading Building Blocks. `https://www.threadingbuildingblocks.org/`.

[50] E. Jeannot and G. Mercier. Near-optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference*, volume 6272 of *Lecture Notes on Computer Science*, pages 199–210, Ischia, Italy, SEPT 2010. Springer.

[51] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002, April 2014.

[52] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[53] Amir Kamil and Katherine Yelick. Hierarchical computation in the SPMD programming model. In *The 26th International Workshop on Languages and Compilers for Parallel Computing*, September 2013.

[54] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis. *Parallel graph partitioning and sparse matrix ordering library. Version*, 2, 2003.

[55] A. Yar Khan, J. Kurzak, and J. Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. *University of Tennessee Innovative Computing Laboratory Technical Report, ICL-UT-11-02*, 2011.

[56] J. Kim, W.J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 77–88, June 2008.

[57] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan C. Catanzaro, Paul Ivanov, and Ahmed Fasih. Pycuda and pyopencl: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[58] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzen, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. ExaScale computing study: Technology challenges in achieving exascale systems. Technical report, DARPA, May 2008.

[59] Peter M. Kogge and John Shalf. Exascale computing trends: Adjusting to the "new normal"' for computer architecture. *Computing in Science and Engineering*, 15(6):16–26, 2013.

[60] William Kramer. Is petascale completely done? what should we do now? joint-lab on petsacale computing workshop`https://wiki.ncsa.illinois.edu/display/jointlab/Joint-lab+workshop+Nov.+25-27+2013`, November 2013.

[61] Xavier Lapillonne and Oliver Fuhrer. Using compiler directives to port large scientific applications to gpus: An example from atmospheric science. *Parallel Processing Letters*, 24(1), 2014.

[62] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, , and Michael Zingale. Parallel netCDF: A high-performance scientific I/O interface. In *Proceedings of SC2003*, November 2003.

[63] Hatem Ltaief and Rio Yokota. Data-Driven Execution of Fast Multipole Methods. *CoRR*, abs/1203.0889, 2012.

[64] Richard Membarth, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. Generating device-specific GPU code for local operators in medical imaging. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 569–581. IEEE, May 2012.

[65] Richard Membarth, Frank Hannig, Jürgen Teich, and Harald Köstler. Towards domain-specific computing for stencil codes in HPC. In *Proceedings of the 2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 1133–1138, November 2012.

[66] Q. Meng and M. Berzins. Scalable Large-Scale Fluid-Structure Interaction Solvers in the Uintah Framework via Hybrid Task-Based Parallelism Algorithms. *Concurrency and Computation: Practice and Experience*, 26(7):1388–1407, 2014.

[67] Jun Nakashima, Sho Nakatani, and Kenjiro Taura. Design and Implementation of a Customizable Work Stealing Scheduler. In *International Workshop on Runtime and Operating Systems for Supercomputers*, June 2013.

[68] netloc. Portable Network Locality. `http://www.open-mpi.org/projects/netloc/`.

[69] OCR Developers. Open Community Runtime. `https://01.org/open-community-runtime`.

[70] Stephen L. Olivier, Allan K. Porterfield, Kyle B. Wheeler, Michael Spiegel, and Jan F. Prins. OpenMP task scheduling strategies for multicore NUMA systems. *International Journal of High Performance Computing Applications*, 26(2):110–124, May 2012.

[71] OpenMP ARB. OpenMP Application Program Interface. `http://openmp.org/wp/openmp-specifications/`.

[72] Carlos Osuna, Oliver Fuhrer, Tobias Gysi, and Mauro Bianco. STELLA: A domain-specific language for stencil methods on structured grids. Poster Presentation at the Platform for Advanced Scientific Computing (PASC) Conference, Zurich, Switzerland.

[73] ovis. Main Page - OVISWiki. `https://ovis.ca.sandia.gov`.

[74] Szilárd Pall, Mark James Abraham, Carsten Kutzner, Berk Hess, and Erik Lindahl. Tackling exascale software challenges in molecular dynamics simulations with GROMACS. *Lecture Notes in Computer Science*, page in press, 2014.

[75] Miquel Pericàs, Kenjiro Taura, and Satoshi Matsuoka. Scalable Analysis of Multicore Data Reuse and Sharing. In *Proceedings of ICS'14*, June 2014.

[76] B. Prisacari, G. Rodriguez, P. Heidelberger, D. Chen, C. Minkenberg, and T. Hoefler. Efficient Task Placement and Routing in Dragonfly Networks . In *Proceedings of the 23rd ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'14)*. ACM, Jun. 2014.

[77] Sander Pronk, Szilárd Páll, Roland Schulz, Per Larsson, Pär Bjelkmar, Rossen Apostolov, Michael R. Shirts, Jeremy C. Smith, Peter M. Kasson, David van der Spoel, Berk Hess, and Erik Lindahl. GRO-MACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.

[78] Sabela Ramos and Torsten Hoefler. Modeling communication in cache-coherent smp systems: A case-study with xeon phi. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 97–108, New York, NY, USA, 2013. ACM.

[79] Florian Rathgeber, Graham R. Markall, Lawrence Mitchell, Nicolas Loriant, David A. Ham, Carlo Bertolli, and Paul H.J. Kelly. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *Proceedings of the 2nd International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 1116–1123, November 2012.

[80] E. Rodrigues, F. Madruga, P. Navaux, and J. Panetta. Multicore Aware Process Mapping and its Impact on Communication Overhead of Parallel Applications. In *Proceedings of the IEEE Symp. on Comp. and Comm.*, pages 811–817, July 2009.

[81] Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. *Commun. ACM*, 55(6):121–130, 2012.

[82] Francis P. Russell, Michael R. Mellor, Paul H. J. Kelly, and Olav Beckmann. Desola: An active linear algebra library using delayed evaluation and runtime code generation. *Sci. Comput. Program.*, 76(4):227–242, 2011.

[83] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *First USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, January 28-30 2002.

[84] John Shalf, Sudip S. Dosanjh, and John Morrison. Exascale computing technology challenges. In *International Meeting on High Performance Computing for Computational Science*, volume 6449 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2010.

[85] M. Showerman, J. Enos, J. Fullop, P. Cassella, N. Naksinehaboon, N. Taerat, T. Tucker, J. Brandt, A. Gentile, and B. Allan. Large scale system monitoring and analysis on blue waters using ovis. In *Proceedings of the 2014 Cray User's Group*, CUG 2014, May 2014.

[86] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[87] Didem Unat, Cy Chan, Weiqun Zhang, John Bell, and John Shalf. Tiling as a durable abstraction for parallelism and data locality. *Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, November 18, 2013.

[88] Field G. Van Zee, Ernie Chan, Robert A. van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. The libflame Library for Dense Matrix Computations. *IEEE Des. Test*, 11(6):56–63, November 2009.

[89] Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. *CoRR*, math.NA/9810022, 1998.

[90] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–33, 2008.

[91] Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. Work-stealing with configurable scheduling strategies. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 315–316, New York, NY, USA, 2013. ACM.

[92] Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, October 2009.

[93] Katherine Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance Java dialect. In *Workshop on Java for High-Performance Network Computing*, February 1998.

[94] Qing Yi and Daniel J. Quinlan. Applying loop optimizations to object-oriented abstractions through general classification of array semantics. In Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff, editors, *LCPC*, volume 3602 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2004.

[95] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[96] Yili Zheng, Amir Kamil, Michael Driscoll, Hongzhang Shan, and Katherine Yelick. UPC++: A PGAS extension for C++. In *The 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS14)*, May 2014.

[97] Songnian Zhou. Lsf: Load sharing in large heterogeneous distributed systems. In *I Workshop on Cluster Computing*, 1992.