# PROGRAMMING ABSTRACTIONS FOR DATA LOCALITY

**2014 Workshop** on
Programming Abstractions
for Data Locality

Lugano, Switzerland
April 28–29, 2014

CSO 14-CR 0542

# Programming Abstractions for Data Locality

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

**Co-Chairs**
Didem Unat (LBNL)
Torsten Hoefler (ETH Zürich)
John Shalf (LBNL)
Thomas Schulthess (CSCS)

### Workshop Participants/Co-Authors

Adrian Tate (Cray)
Amir Kamil (Lawrence Berkeley National Laboratory)
Anshu Dubey (Lawrence Berkeley National Laboratory)
Armin Grlinger (University of Passau)
Brad Chamberlain (Cray)
Carter Edwards (Sandia National Laboratories)
Chris J. Newburn (Intel)
David Padua (UIUC)
Didem Unat (Lawrence Berkeley National Laboratory)
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 Fuehrlinger (Ludwig-Maximilians-University)
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 Pericas (Chalmers University of Technology)
Naoya Maruyama (RIKEN)
Paul Kelly (Imperial College)
Peter Messmer (Nvidia)
Romain Cledat (Intel)
Satoshi Matsuoka (Tokyo Institute of Technology)
Thomas Schulthess (CSCS)
Torsten Hoefler (ETH Zürich)
Vitus Leung (Sandia National Laboratories)

### Executive Summary

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. While microarchitectural technology trends allow the scaling of the number of cores per chip, cache coherence will likely not scale to the large number of cores due to the traffic overhead of maintaining coherence. Architectural trends break our existing programming paradigm because the current software tools optimize for floating point operations not memory traffic. They ignore the incurred cost of communication and simply rely on the hardware cache coherency to virtualize data movement. For example, the current OpenMP-3 usage model describes how to parallelize loop iterations and divides the iteration space evenly among processors with limited flexibility for expressing data layout. 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 *tiling*, *array views* and *iterators* to managing data locality. There is an opportunity to identify commonalities in strategy to enable us to combine together 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 language constructs to achieve this goal.

***The goal of the workshop and this resultant report to identify common themes and standardize concepts for locality-preserving abstractions for exsacale programming models.***

# Acknowledgment

ack goes here

# Contents

# Chapter 1

# Introduction

- Define area and scope (Programming Abstractions for Data Locality: Moving towards Data-Centric Computing)
- Opportunities presented by research in data locality for programming systems
- Key overarching findings and recommendations

# 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. While microarchitectural technology trends allow the scaling of the number of cores per chip, cache coherence will likely not scale to the large number of cores due to the traffic overhead of maintaining coherence. In the future, software-managed memory and incoherent caches or scratchpad memory will be prevalent. Thus, application developers need a set of programming abstractions to describe data locality on the new computing ecosystems.

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. For example, we design applications for MPI to express the coarse-grained data locality where the programmer explicitly represents two levels of computation local (within the node) and message passing (between nodes). We also deal with cache locality by either relying on the compiler to perform loop blocking on our behalf or manually reorganize loops to explicitly block data for different levels of the cache hierarchy.

The effects were modest enough that these explicit techniques were sufficient to enable codes to perform on different architectures. However, with the exponential rise in explicit parallelism

## 2.1 Hardware Basis for Locality Optimizations

### 2.1.1 The End of Classical Performance Scaling

The year 2004 marked the approximate end of Dennard Scaling. Chip manufacturers could no longer reduce voltages at the historical rates because doing so would have caused the chip to become unreliable. Furthermore, as transistors became smaller, leakage current became a bigger issue. The power dissipated by this leakage current was formerly a small contributor to overall power consumption, but had since become a substantial concern. This meant there was no longer advantage in further reducing operating voltages. Other gains in energy efficiency were still possible; for example, smaller transistors with lower capacitance consume less energy. The inability to reduce the voltages further did mean, however, that clock rates could no longer be increased within the same power budget. Once the practical heat dissipation limit for consumer devices was reached (at approximately 100W of power consumption at the system level) further clock frequency scaling was abandoned.

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 core count is clearly visible in the black trend line in Figure***need figure***. 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.

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***need fig***. In contrast, the power consumption of transistors continues to decrease as their gate size (and hence capacitance) 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. See Figure***need data locality fig***

### 2.1.2 Data Locality

Many of the motivations for data locality optimization is rooted in recent hardware architectures trends.

First among the is the concern that the cost of data movement even within a chip is set to exceed the cost of

The trend was first observed in the 2008 exascale report, but this has

Whereas data locality has been a concern for

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. However, we have largely depended upon automatic management of hardware caches and compiler optimizers to make use of this features.

With the end of Dennard scaling, clock-rates are no longer increasing (figure **??**) and have been replaced with explicit parallelism. Figure **??** (b) shows a consistent path towards chips with hundreds or even thousands of cores per chip die. Fortunately, the primary growth in parallelism is on-chip rather than between chips. Consequently, the available bandwidth between these cores on chip is 10x higher than between nodes, and the latencies are a factor of 10x to 100x lower than between nodes in an HPC system. The much lower overheads within the chip (compared to off-chip overheads) offer better opportunities to derive performance through increased on-chip parallelism (provided we can express enough parallelism), but the challenge remains daunting.

Parallel computing owes its success to weak scaling between nodes, where the size of the problem solved grows proportionally with the increased parallelism. The exponential growth in explicit parallelism pushes us towards strong scaling where performance improvements are derived exclusively from applying more parallelism to the same-sized problem. Certainly, adding cores is fine for increasing compute capacity, but the move towards strong scaling causes communication overheads to become a huge concern. During the era of exponential scaling of the clock rates within the node took up a lot of the slack when it came to increasing execution rates proportional with the growth of the problem size that supported weak scaling. For example, in a climate model we could weak scale to increase the problem resolution by a factor of 2x, you must also reduce the time-step size by a factor of 2x to maintain numerical stability of the calculation. Fortunately, the processor clock rate typically increased by a factor of 2x in that same time interval under old scaling rules. With the new scaling rules, the 2x improvement in performance must be derived entirely from doubling the performance using increased parallelism. This is a far more daunting challenge as there are limits to how much parallelism one can express with conventional techniques such as domain decomposition before the communication overheads kill performance.

The research community is starting to look at more aggressive techniques that depart from our traditional bulk-synchronous or SPMD (Single Program Multiple Data) model for parallelism. One common approach involves "functional partitioning" where different solvers of the simulation are executed concurrently to overcome these granularity and processor speed limitations. For example a combustion code or climate simulation would concurrently schedule parts of the memory bandwidth intensive fluid dynamics compilation concurrently with the FLOP-intensive chemistry calculations – thereby increasing the number of utilized processors and hence overall throughput without relying on increased domain decomposition. In addition, you can use very low-overhead inter-core message queues to reduce communication overhead within a chip as we did for the Green Flash chip-multiprocessor design [**GreenFlash**]. However, conventional C and Fortran language semantics make it difficult to manage this kind of parallelism, so there is a new thrust to explore language semantics that support asymmetric and asynchronous approaches to achieving strong-

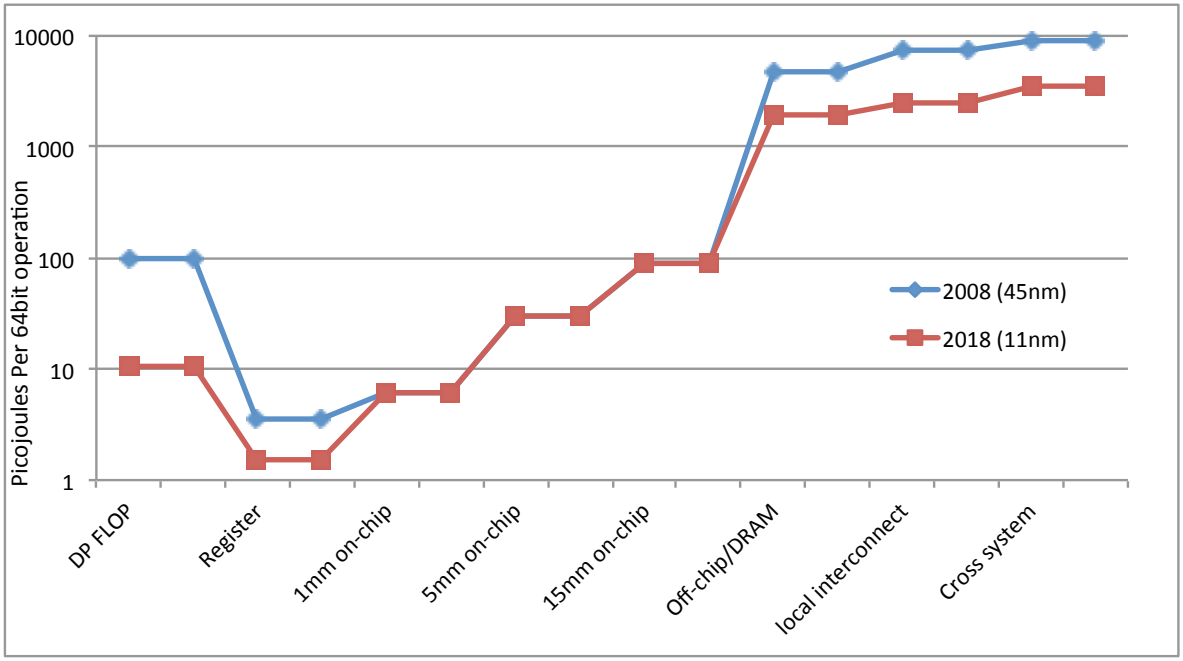


Figure 2.1: Data movement is overtaking computation as the most dominant cost of a system both in terms of dollars and in terms of energy consumption. Consequently, we should be more explicit about reasoning about data movement. This diagram shows the cost of operations or data movement for a 64-bit operand. By 2018, the cost of moving a 64-bit operand a mere 5mm across the chip will exceed that of a floating point operation that uses that operand. *The FPU and Register File energy model is computed from Tensilica's RTL design for an LX4 core with FPU. The data movement on-chip is based on a standard resistive/capacitive model with simple signal regeneration. The DRAM is based on Micron's projections for DDR parts. The interconnect model is based on extrapolating historical improvements in optical transceiver efficiency and data rates.*

scaling performance improvements from explicit parallelism. Successful demonstration of such parallelization procedures for a range of leading extreme-scale applications can then be utilized by other similar codes at smaller scales, which will accelerate development efforts for the entire field.

### 2.1.3 Performance Heterogeneity

We have evolved a parallel computing infrastructure that is optimized for bulk-synchronous execution models. It implicitly assumes that every processing element is identical and operates at the same performance. However, performance projections out to exascale in figure **??** show that the only technological path that has a hope of achieving exascale by 2020 involves a heterogeneous architecture consisting of both lightweight and heavyweight cores. Even if you are not enthusiastic about the complexity of programming heterogeneous compute engines (hybrid/accelerated computing), application developers will still need to confront heterogeneity even for homogenous processor technology. Emerging adaptive algorithms, near-threshold voltage operation, and clock-speed throttling challenge current assumptions of uniformity. Since the most energy-efficient FLOP is the one you do not perform, there is increased interest in using adaptive and irregular algorithms to apply computation only where it is required, and also to reduce memory requirements. Even for systems with homogeneous computation on homogeneous cores, new fine-grained power management makes homogeneous cores look heterogeneous. For example thermal throttling on Intel Sandybridge enables the core to opportunistically sprint to a higher clock frequency until it gets too hot, but the implementation cannot guarantee deterministic clock rate because chips heat up at different rates. In the future, nonuniformities in process technology and Near-Threshold-Voltage (NTV) for ultra-low-power logic will create non-uniform operating characteristics for cores on a chip multiprocessor. Fault resilience will also introduce

inhomogeneity in execution rates as even hardware error correction is not instantaneous, and software-based resilience will introduce even larger performance heterogeneity.

So even homogeneous hardware will look increasingly heterogeneous in future technology generations. Consequently, we can no longer depend on homogeneity, which presents an existential challenge to bulk-synchronous execution models. This has renewed interest in alternative execution models that are better able to accommodate performance non-uniformity. For example ETI Swarm, ParalleX, and Charm++ posit a completely asynchronous execution methodology that enables parallelism to be derived from concurrent scheduling of independent work rather than just by data decomposition. These techniques that borrow much from coarse-grained dataflow methods are garnering renewed interest because of their ability to flexibly schedule work and to accommodate state migration to correct load imbalances and failures for this kind of functional partitioning model.

Performance heterogeneity offers a number of challenges to conventional approaches to data management.

1) Breaking lexical order execution of programs 2) Balancing data locality with optimal scheduling 3) Isolation of side-effects 4) Abstractions for over-decomposition

# Chapter 3

# Motivating Applications and Their Requirements

Discussion of data locality from the perspective of applications implies 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. Spatial connectivity has direct implication on locality: the left end of this axis represents zero-connectivity applications which are embarrasingly parallel and at the right end are the ones with dynamic connectivity such as adaptive meshing with static meshes falling somewhere in-between. The componentization axis is more concerned with software engineering where the left end consists of the small static codes, while at the right end are the large-multicomponent codes with a continuous development-debugging process. The HPC applications space mostly occupies the top right quadrant, and was the primary concern in this workshop.

While the applications space itself is very large, the participating applications and experts provided a good representation of the numerical algorithms and techniques used in majority of state-of-the-art application codes (i.e. COSMO[**cosmo**], GROMACS[**gromacs4**, **gromacs4.5**, **gromacs-exascale**], Hydra & OP2/PyOP2[**Hydra**, **PyOP2**], CHOMBO[**chombo**], VIS[**vis**]. 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 domain 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 solvers 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 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.

- By contrast, typical low-order partial differential equation solvers typically 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 co-exist with AMR, 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 well known and valid concern in the applications communities about wise utilization of the scarcest of the resources, the developers' time, and protecting the investment already made in the mature production codes of today. The most important consideration for the applications community, therefore, is the time-scale of change in paradigms in the platform architecture and major rewrites of their codes. 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 The 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. Different research groups vary in the extent to which they need to prepare. A great deal of useful science is still done by a small research group with prototype-level software of very limited scope. Like all other software, these too will need to be refactored or rewritten for different platforms, but being small, they are unlikely to be enough of a drain on resources to warrant looking for general portable solutions. Those research efforts were not the focus of the workshop, even though they too will gain from more stability. Instead, the purpose of the workshop was to consider those computation-based science and engineering research efforts that rely upon larger codes with many moving parts that also demand more resources to compute one single model. Many times, the process of scientific discovery requires exploration of the parameter space with a corresponding need to compute several models. 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 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 that they can adopt.

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, PyOP2 [**PyOP2**], STELLA (STEncil Loop LAnguage) [**STELLA14**] and HIPA$^{cc}$ (Heterogeneous Image Processing Acceleration) [**MHTKE12a**] are *embedded domain-specific languages* (<**reference to appropriate place in Ch. 4 "Language ..."**>) that are tailored for a certain field of application and abstract from details of a parallel implementation on a certain target architecture. PyOP2 and the latter approaches use Python and C++ as host language, respectively. PyOP2 targets mesh-based simulation codes over unstructured meshes and uses FEniCS [**fenics**] to generate kernel code for different multicore CPUs and GPUs. Furthermore, it employs runtime compilation and scheduling. STELLA

**Frank:** Ask Mauro whether there is a more appropriate reference to STELLA.

**Frank:** Ask Mauro whether there is a more appropriate reference to 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 [**MHTK12**] and provides code generation for accelerators, such as, GPUs (CUDA, OpenCL, RenderScript) and FPGAs (C code that is suited for high-level synthesis). All the aforementioned approaches can increase *productivity*—e.g., reducing the lines of application code may not only reduce programming but also debugging time—in their target science domain as well as *performance portability* across different compute architectures.

Other efforts, such as 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. None of them provide a complete stable solution that applications can use for their transition to long term viability.

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,[**gromacs4**, **gromacs4.5**, **gromacs-exascale**] 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 latest such efforts in GROMACS 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. If such a DSL could be made widely available for the molecular simulation community, then a great deal of duplicated effort could be avoided, and new applications would benefit immediately.

## 3.2 The Challenges

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 most important one is fear of adding a critical dependency that may not be supported in the long term, or is portable now and in the future. Often the developers will opt for a suboptimal or custom-built solution that does not get in the way of being able to run their simulations. For this reason embedded DSLs, or code transformation technologies are more attractive to the applications. These techniques, because they are not all-or-none solutions, have the added advantage of providing an incremental path for transition. 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 them 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 modelling 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

As mentioned earlier, for the production-grade applications, the biggest non-negotiable requirement is the stability of the paradigm. Code bases that have years of investment in algorithmic development will not adopt technologies that can prevent them from utilizing computational resources as they become available.

For example, even today research groups exist that refuse to use parallel I/O libraries, despite the fact that these have been robust and supported quickly on new platforms for several years. They still prefer to roll their own I/O solution with all the attendant performance and maintenance overhead just to avoid the dependency. It would be even harder to persuade these communities to adopt higher-level abstractions unless there was 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.

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 upon data locality, data layout, or memory alignment. In the list below we outline some abstractions and/or language constructs that would be helpful to applications.

### 3.3.1 The Wish List

- Ability to write dimension independent code easily.
- Depending upon the data access preference of the target architecture, switch the data structures between array-of-structures and structure-of-arrays without having to rewrite the code.
- 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. DSL approaches should try to reverse this direction.
- Source-to-source transformations for mapping from high level language to low level language. If tools were made available for creating DSLs or language extensions, this approach could even allow communities to develop their own domain specific language without having to build the underlying infrastructure.

## 3.4 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. The most important area of research for the applications community is the high level multi-component framework that can maximize data locality in 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 design 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.

From the applications perspective, it is important to understand the impact of different locality models. For instance, with 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.

# Chapter 4

# Data Structures and Layout Abstractions

## 4.1 Introduction

There is a diversity of ways to organize and represent data structures. One source of diversity is among users, based on what they find to be convenient, intuitive and expressive. A second source is differences between what is natural to users and what leads to efficient performance on target architectures. A third source of diversity arises from differences among target architectures in what organizations and representations are most efficient. Rarely does a single abstraction effectively span these three kinds of diversity. Our goals are to enable a range of abstractions, to converge on ways to specify abstractions and map between them, and to enable those abstractions to be freely and effectively layered without undue restriction. Recently, a number of programming interfaces such as Kokkos, TiDA, OpenMP extensions, GridTools, DASH, Array Extensions [**all**] have arisen to give developers more control over data layout and to abstract the data layout itself from the application. In this chapter, we discuss the key points when designing such abstractions, emerging approaches with (im)mature solutions, and potential research areas. Here, we focus on locality management on data-parallel algorithms and leave the discussion on task-oriented abstractions to Chapter 6.

Before going into further discussions, we define relevant terminology. A *data structure* is an organized collection of data residing in one or more *memory spaces*. Memory spaces may have different access characteristics, e.g. 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. *Data decomposition* is the partitioning of a data structure's data into smaller chunks with the intent of assigning each chunk to a memory space or introducing parallelism across chunks. *Data placement* is the mapping of decomposed data structures' chunks to memory spaces. *Data layout* is the arrangement of a data structure according to the addressing scheme used by a given abstraction layer. 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 uses Hilbert curves, and a hardware-managed cache, for example. There must be a mapping between the data layouts in the various abstraction layers, and potentially in each of the memory spaces. A computation accesses memory according to a *data access pattern* which is determined by the composition of (1) how the computation's parallel tasks traverse the data structure, (2) how the computation's tasks are scheduled (i.e., the *execution policy*), (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. *Data locality* is a proxy measure for the relative cost of moving different data to/from a given task as it executes; smaller "distance" implies smaller cost. Locality has both spatial (memory space) and temporal (execution scheduling) considerations.

## 4.2 Key Points

The research communities have been developing data structure abstractions for several years to improve expressiveness in parallel applications. However only recently there is a shift towards a more data centric programing 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 is applied to the distinction between a logical, semantic specification, and lower-level controls over the physical implementation. At the semantic level, scalar 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 computations 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 which 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.

The key point of separation of concerns is of offering, to the user of the interface, an architecture agnostic *abstract machine* (AM). An AM is necessary to express an algorithm, being it a functional language or a Turing machine. An algorithm written for an AM must be translated into program for a *physical machine* (PM), which can be closely related to the AM or completely different. 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 forseeable PMs. This is the well know problem of *portable efficiency*. From this perspective an AM should be distant from PMs for two reasons: first it gives the user an 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

Domain experts specify the work to be accomplished and they map that work onto parallel data collections. They want the freedom to use a representation of data that is natural to them and do not prefer to be concerned with 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 overspecifying 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, and a lower,

physical abstraction layer, at which performance is tuned and the harvesting of parallelism on a particular target machine is controlled. This separation of concerns allows code modifications to be localized. 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.

The interfaces the authors of this report are developing, the already cited Kokkos, TiDA, OpenMP extensions, GridTools, DASH, Array Extensions, provide AMs at widely different abstraction levels. 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 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. 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. **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. **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 user level the locality is taken into consideration only implicitly. Not sure what to say about the others .

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. In the case of C++ libraries, there is the opportunity of naturally building higher level interfaces using lower level ones. For instance, a DASH multidimensional array 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, such as C++.

## 4.4 Discussion

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 at least four options for mitigating this situation: extend standard languages, augment standard languages with embedded domain-specific languages (DSLs), depart from standards with specialized languages, or develop "active libraries" using standard language features.

Different language rules may be required to overcome limitations imposed by current languages. For example, C exposes physical data layout, and limits a compiler's ability to modify the data layout. Source to source translators are still subject to language rules, whereas new languages may remove such limitations through abstraction. The contributors to this chapter of the report largely followed the library-based approach, where the choice of base language to express abstraction varies among different groups. Some prefer standard languages such as Fortran and C/C++ in order to maximize impact and leverage market breadth of the supporting tool chain (e.g., compilers, debuggers, profilers). Wherever profitable, there can be a push to "redeem" existing languages by amending or extending them, e.g. via changes to the specifications or by introducing new ABIs.

C++ seems to provide some of the greatest opportunities for enabling data layout abstractions in an established base language. By contrast, Fortran is relatively inflexible and while dynamically typed scripting languages provide lots of flexibility to users, they make optimization difficult.

C++ meta-programming can cover a significant fraction of the desired capabilities, e.g. polymorphic data layout and execution policy. Specialization can be hidden in template implementation and controlled by template parameters. For example, C++ templates enable users to express data accesses in the body of the code in ways that are natural and convenient, and manage the mapping from that logical abstraction to an underlying layout which may be much better for performance in the template definition. In efforts like Kokkos, Arrow Street and SIMD Building Blocks, add references compilers have demonstrated an ability to "see through" those abstractions to effectively harvest parallelism. In the latter two efforts, the user is required to explicitly provide a bidirectional mapping between abstractions, which can be burdensome and error prone. Efforts to augment the C++ standard with better support for introspection appear to be a promising path for easing that burden. The ISO C++ standardization committee is addressing performant, on-node parallelism for a future standard (e.g., C++17). Even though a huge number of applications are implemented in Fortran, there is a concern about Fortran's lack of extensibility in terms of lambdas and templates.

This chapter of the report focuses on performance controls that are used to manage data to achieve performance, rather than on how parallelism is exposed at the semantic layer. We address language interface issues and provide a taxonomy of performance controls. It would also be good to highlight issues that can impact productivity, performance, and performance portability. We have identified some areas of agreement on where to start in this overall effort.

Performance-related controls pertain to data and to execution policy. These may vary in their function, scope and granularity.

- Data controls may be used to manage:
  - Decomposition, which tends to be either trivial (parameterizable and automatic) or not (explicit)
  - Binding to storage: to a particular type of memory (e.g. read only, streaming), to a phase-dependent depth in the memory hierarchy (prefetching, marking non-temporal), or to memory structures which support different kinds of sharing (SW managed, cached)
  - Mechanisms for, and timing of, distribution to data space/locality bindings
  - Data layout
- Execution policy controls may be used to manage
  - Decomposition of work, e.g. 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, thread or a SIMD lane
- These controls may be applied at different scopes and granularities through a varity of mechanisms
  - Data types - global, fine-grained, can vary by call site
  - Function or construct modifiers - local coarse-grained, can vary within a scope
  - Environmental variable controls - global policies

¡Summarize areas of low-hanging fruit.¿

Libraries? Flavors of functional programming ?

## 4.5 Research Plan

¡Discuss data model: motivation, value, issues.¿

¡removing barriers to freedom of abstraction¿

¡portable efficiency¿

## 4.6 OUTLINE

HIGHLIGHTS

- SCOPE
  - In this chapter, we discuss language and library interfaces that help manage programming abstractions for data locality, particularly for data structures and data layout.
- KEY POINTS
  - Didem: I think this should appear in the introduction of the entire report. What are the *easy* and the *complex* parts of algorithm design? I think the *easy* should include parallelism. From the implementation point of view this may be different, but compared with the challenges of locality, parallelism should be considered easy and high level. What is the level of "algorithmic locality" that we want to be able to specify? Is local/non-local distinction sufficient?
  - From the requirements of the applications (and scalability) we focus on data-parallel algorithms (even though we may use task oriented abstractions and runtimes).
  - The lack of a single abstract machine (programming model) to program for performance imposes the choice of data-structure specific constructs and specific implementations of those for different platforms (for now this is an empirical observation)
  - not sure I got this item, it seems similar to the item right above The diversity of the architectures also forces these constructs to work at level of *means of combination/composition* (it's not sufficient to provide very efficient primitive operations, like BLAS).
  - Separation of concerns is important to maximize expressivity and optimize performance. That principle is applied to the distinction between a logical, semantic specification, and lower-level controls over the physical implementation.
  - At the semantic level, scalar 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.
  - Performance tuners, who may be distinct from the domain experts, may exercise control over performance with a set of interfaces which 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,
    - map parallel work onto underlying hardware mechanisms for supporting parallelism, and
    - exercise control over temporal sequencing of work and movement of data for locality.
  - Some (im)mature solutions implemented in different languages include: Kokkos, TiDA, OpenMP extensions, GridTools, DASH, Array Extensions
- TERMINOLOGY (define each of these up front)
  - Data structure: A organized collection of data residing in one or more memory spaces.
  - Layout (of data structure): The mapping of a data structure into one or more memory spaces, typically with the intent of efficient and performant access by a collection of computations. Example: A 2D matrix of size 100x100 holding integers is a data structure: a semantic/logical entity in which a programmer thinks. Storing and processing the matrix in a computer requires a layout that maps the 2D matrix onto the 1D address space of the computer in some way. The mapping could be a simple linearization or it could be a more complex blocked (tiled) scheme.
  - Memory space: A computer has multiple memory resources. A single computer traditionally has main memory, one or more levels of hardware managed cache memory, and registers; where the main memory is the only memory space explicitly managed by a program. Advanced architectures have heterogeneous memory for performance that must be managed by computations. Examples of these memory spaces include CPUs' non-uniform memory access (NUMA) regions, separate GPU memory, and software-managed on-chip memory.

    * Distributed data structure / parallel data structure: not sure we should use either of those terms, but if we do we should make sure what we mean: is the data distributed over multiple nodes/address spaces? is there concurrent access to the same data structure?
    * Didem: should be part of the report intro Locality: A measure of distance between data on a computer. For example, do the data reside in the same compute node of a cluster, in the same memory space, in the same cache line? [I can't think of a direct definition that doesn't require a whole lot of other definitions up front, so how about the following phenomenological definition?]
    * Execution policy: How a computation's units of work are scheduled and mapped onto data collections. is this the same thing as thread binding?
    * Memory access pattern: A computation accesses data structures according to its execution policy.
    * Memory access pattern: The memory access pattern of a computation is determined by the composition of the computation's algorithm, computation's execution policy, and layout of the computation's data structures. The quality of a computation's memory access pattern is determined by the time or energy consumed accessing its data.
    * Polymorphic layout: The quality of a computation's memory access pattern may depend upon the architecture of the computer on which it executes. A data structure has a polymorphic layout if the layout can be changed to improve the memory access pattern without modifying the source code of computations depending upon that data structure.
    * A data structure (-layout) has good locality (-behavior) with respect to an algorithm if operations on the data structure can be executed efficiently in terms of run-time and/or energy consumption. So locality is not a property of the data structure or layout alone, there is an execution/code component that determines if locality behavior is good.
    * Algorithmic locality vs. actual/implementation locality: Not sure I would know how to define these terms. What is algorithmic locality of matrix multiplication?
    * Logical, semantic level: programmer expresses the WHAT, exposes OPPORTUNITY in a natural expression; portable
    * Portable efficiency
    * Algorithmic locality vs. actual locality or implementation locality
    * Physical, performance level: CONTROL over HOW, so as to meet performance goals; may be implementation specific
    * Polymorphic: may take different forms, depending on the abstraction layer or optimization target
    * Language interface: a spec or API used by a programmer, which may be part of the base language, a compiler-interpreted directive, or a library API [this may need work]
    * Language construct: something that can be identified as a *keyword* in a language interface and the grammar rules that applies to it.
    * Binding: mapping from the logical to physical domain, e.g. for storage
    * *What follows should be either filled in or dropped*
    * control space(use execution space?)
    * binding (or execution policy ?)
    * data layout
    * data decomposition
    * data distribution
    * iteration space traversal (avoid using loop traversal, maybe use domain traversal?) maybe : *iteration patterns* to be paired with *access patterns*?
    * access type
- AGREEMENTS

* Abstractions for performance portability are needed: Data layout, tile sizes, memory access patterns need to be tuned when application is moved between machines. [We may want to tweak that list.]
* The separation of logical from physical concerns enables:
  - Separation of concerns between domain experts and performance tuning experts
  - Maximal exposure of opportunity, vs. hard coding a particular trade-off
  - Getting free of enslavement to restrictions or suboptimalities imposed at the logical level
  - Minimizing, or at least localizing, code modification for control over performance
  - Polymorphism across targets and abstraction layers
* We need a data model to assess the data locality
  - We have a model for parallelism but do not have a model for data locality
  - Optimal trade-offs may shift over time and across target systems.
  - Models need to take into account the capabilities and capacities of computational elements, memory structures, and the network fabric
  - Models should reflect the consequences of various controls, such as the placement of work near data
* It's easier to standardize high-level concepts, suggesting that
  - Extensions for controls over how parallel work and data are managed may best be targeted at libraries and language interfaces, rather than base languages. There's greater freedom for diversity there.
  - Base languages are a good longer-term target for semantically exposing opportuniteis for parallelism (and locality?)
* Low hanging fruit: (1) multidimensional array support (in C and C++) (2) runtime polymorphic data layout: ideally change the layout in runtime (it should always be clear when an operation has performance costs), (3) compile time polymorphic layout: change the layout at compile time with minimal code modifications, support layout in the type system
* Performance-related controls pertain to data and to execution policy. These may vary in their scope and granularity.
  - Data controls may be used to manage: these are under a new itemize
  - Decomposition, which tends to be either trivial (parameterizable and automatic) or not (explicit)
  - Binding to storage: to a particular type of memory (e.g. read only, streaming), to a phase-dependent depth in the memory hierarchy (prefetching, marking non-temporal), or to memory structures which support different kinds of sharing (SW managed, cached)
  - Mechanisms for, and timing of, distribution to data space/locality bindings
  - Data layout end itemize
  - Execution policy controls may be used to manage begin new itemize
  - Decomposition of work, e.g. 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, thread or a SIMD lane end itemize
  - These controls may be applied at different scopes and granularities through a varity of mechanisms begin new itemize
  - Data types - global, fine-grained, can vary by call site
  - Function or construct modifiers - local coarse-grained, can vary within a scope
  - Environmental variable controls - global policies end itemize
* lambdas for domain traversal (or iteration space traversal) [is this now adequately covered below?]

– DISAGREEMENTS

    * Binding [could someone flesh out the opposing points of view?] Depending on the level at which a language/library stands (single thread, multicore chip, parallel distributed system) the binding is responsibility of the user or the system. I don't think we agree on what the level of abstraction is right, so we could not agree in the binding, I guess.
    * support for memory spaces: can be hidden from the programmer or exposed [could someone flesh out the opposing points of view?] Traditional architectures have hidden memory spaces within node through hardware supported movement of data through a cache memory hierarchy. This implicit memory space management strategy has been applied to distributed memory clusters through PGAS programming models and runtimes. There has is not a consensus on PGAS versus explicit management of data movement among clusters' compute nodes. Advanced compute node architectures have heterogeneous memory spaces which must be managed for computational performance. It is an open question as to whether a programming model can be deployed that implicitly manages these memory spaces and provides sufficient memory access quality.
    * Choice of language interfaces
        · Some prefer standard languages, in order to maximize impact and leverage market breadth of the supporting tool chain (e.g., compilers, debuggers, profilers). Wherever profitable, there can be a push to "redeem" existing languages by amending or extending them, e.g. via changes to the spec or by introducing new ABIs. It is noteworthy that the ISO C++ standardization committee intends to address performant, on-node parallelism in a future language standard.
        · Others believe that specialized languages are required, e.g. to "get us to exascale." Different language rules may be required to overcome limitations imposed by current languages. For example, C exposes physical data layout, and limits a compiler's ability to re-layout data. Source to source translators are still subject to language rules, whereas new languages may remove such limitations through abstraction.
        · There's some agreement that C++ metaprogramming can cover a significant fraction of the desired capabilities, and that it's a middle road for being able to implement DSLs that are embedded within standard languages. Specialization can be hidden in template implementation and controlled by template parameters.
        · There's some concern about Fortran's lack of extensibility, e.g. in the direction of lambdas, templates and metaprogramming

– GAPS (what is missing? not covered at the workshop)
    * A data model for which data layout is more suitable for which algorithm? or metric for locality
    * The mentioned distinction between WHAT and HOW is subtle. I think every developer is concerned with WHAT. As library/language developers we want our users to be concerned by certain WHAT that we turn into HOW by stating some lower level WHATs. In my work I can use a PGAS approach (in which locality is expressed as a here-or-there) underneath but hide it altogether to my user. I think we disagree at what level our users should express their programs. We agree that there must be a separation of concerns but not where the separation should be.

– RESEARCH AGENDA
    * If we speak of data locality, is a binary local/remote distinction enough or do we need a more fine-grained differentiation? If so, what is the best way to represent and measure this multi-level locality concept. Is "hierarchy" always the right concept and is a tree always a good representation?
    * What about horizontal locality management vs. vertical locality management? Do they require different abstractions or can they be handled in a uniform way?
    * Identify minimal set of data and execution policy controls
    * Compare/contrast available options for specifying those controls

* Identify gaps and prescribe steps toward closure of those gaps
* Converge on a minimally set of (semi-standard) solutions that provide adequate coverage
* Prove or disprove that an portable efficiency can be achieved with a single language (in order to prove or disprove that data-structure specific approaches are the only beign possible)

# Chapter 5

# Language and Compiler Support for Data Locality

## 5.1 Introduction

*SCOPE*

- *In this chapter, 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.*
- *We explore the proposition that management of locality in parallel programming is a* language *problem.*
- *Language-based solutions may come in the form of new, general-purpose parallel programming languages. As well as supporting intuitive syntax, doing so creates the scope for 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 refer to abstract data types. Multidimensional arrays are a crucial starting point. Many more complex domains can be characterized by the abstract, distributed data structure on which parallel computation occurs. Domain-specific tools are often focussed 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 a fundamental concern for current and future HPC architectures, since the cost of data movement defines a large fraction of overall application performance. The concept of locality, however, 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 the proposition that management of locality in parallel programming is a *language* problem. 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 [**DBLP:journals/corr/math-NA-9810022**]. Active library technologies differ in how this is achieved - examples include template metaprogramming in C++, Lightweight Modular Staging in Scala [**DBLP:journals/cacm/RompfO12**], source-to-source transformation (for example using tools like ROSE [**DBLP:conf/lcpc/YiQ04**]), and run-time code generation, driven either by delayed evaluation of library calls [**DBLP:journals/scp/RussellMKB11**], 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 [**Hudak96buildingdomain-specific**]. 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 [**Erdweg:2011:SLS:2076021.2048099**].

- *Directive-based language extensions:*

- *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 more 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.

## 5.2 Key Points

*KEY POINTS*

- *We need to distinguish between different levels of abstraction, and automation - we need solutions for explicit programmer control over locality and data movement, and we need tools to support composition of parallel components in a way that allows locality and communication to be handled automatically.*

- *We need locality and communication to be robustly evident in the source code, so that programmers have a clear model of execution costs that can be supported by profiling tools.*

- *Language-concepts can support programmers in thinking cleanly about locality (such as distinguishing between local-view and global-view).*
- *Expression of locality needs to be portable across machines.*

## 5.3 State of the Art

*EXAMPLES OF KEY WORK IN THE AREA*

- *HPF (and Brads blog) - vagueness - robust, well-defined, well-understand abstractions*
- *UPC (need to have clear ambitions/capability beyond the low-hanging fruit)*
- *Research vehicles can be narrow but a basis for a standard has to have potential for beyond*
- *Titanium, Global arrays*
- *Chapel (multiresolution concept)*

*STANDARDS*

- *Opportunities for standardisation of mature technologies*
- *What is this standardisable low-hanging fruit?*
- *how to influence standards?*

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 for 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.

Unified Parallel C (UPC) and Co-Array Fortran (CAF) are two 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. Both differ from HPF, ZPL, and Chapel in that programs are written using an explicit 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. There are good reasons to be skeptical about whether such SPMD-based locality models (including MPI) will be sufficient for leveraging the hierarchical architectural locality present in emerging architectures. If not, such models will likely be forced to be part of hybrid programming models in which distinct locality abstractions are used to express finer-grained locality concerns.

## 5.4 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.*
- *Compositional metadata vs explicit dependence/synchronisation/movement (Brad has words)*

*GAPS (what is missing? not covered at the workshop)*

-

## 5.5 Research Plan

*RESEARCH AGENDA*

- *Scope out the opportunities for JIT - plenty of compelling need for infrastructure (specialisation, auto-tuning, ) [[whats the research agenda/challenge here?]]*
- *How do we create a user community?*

*CONCLUSION - RECOMMENDATIONS*

- *Opportunities for standardisation of mature technologies*
- *What is this standardisable low-hanging fruit?*
- *How to influence standards?*
- *Define research agenda*

# Chapter 6

# Data Locality in Task Models

## 6.1 Introduction

SCOPE (**Jesus**)

- Design of interfaces to express locality in a task-based programming model
- Terminology: "task-based", "interfaces" (mostly to specify the levels of the interfaces)
- task-based parallelism is the way to go for many applications (MPI+Task)
- Fast prototyping / productivity
- Redesigning numerical algorithms to express locality and how to express it (nested parallelism, divide and conquer, tree structure)
- Interface betw app/rt/hw to support locality
- Specific performance/debugging tools for task-based applications
- Compiler technology RT Vs Library RT

## 6.2 Key Points

DISCOVERIES/CHALLENGES

- Task granularity (flexibility) (**Miquel**): 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 tasks such as scheduling and work stealing. Task granularity impacts locality because 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 to improve performance and reuse. The optimal granularity depends on data sharing between tasks:
  - In the absence of data sharing one should target a task footprint of $\frac{\text{SharedCacheSize}}{\text{Ncores}}$.
  - In the extreme case, if all data is shared, the target task footprint can become SharedCacheSize. In general it will lie somewhere in between these two cases.

Finding the best granularity is a complex problem as all three metrics to be optimized (overheads, load balance, memory hierarchy) are coupled. Since the optimum configuration may be a data input dependent problem, auto-tuning techniques should be explored. Enabling autotuning involves programmer effort to find ways to partition data sets in a parametric way, allowing to tune the task size. This is also a problem dependent issue. To enable parametric granularity control, program and data decomposition should preferably be performed following a top-down approach rather than bottom-up.

- Scheduling overhead (**Miquel**): Scheduling overhead refers to the time required by the runtime to generate the execution plan of a DAG computation. It includes everything that happens between two execution units except for waiting time (synchronization). Depending on the tasking semantics this involves different amount of work:
  - *T*ask-parallel: In task-parallel runtimes, spawned tasks are ready to be executed (cite: Cilk, TBB task_groups, OpenMP 3.0 Tasks). In this scenario 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. This scheme has low overhead but supports mainly divide-and-conquer and parallel loop style of computation.
  - *T*ask-dataflow: In dataflow schemes (cite: OmpSs, OpenMP 4.0 tasks, TBB dependencies) there are two scheduling levels: 1) resolving dependencies to find ready tasks and 2) scheduling ready tasks to workers. The latter is in general performed as in task-parallel schemes.

  Because of these differences, task-parallel runtimes tend to have smaller scheduling overheads compared to task-dataflow runtimes. It is an open research problem to reduce the overheads of task-dataflow schedulers in order to efficiently support tasks of finer granularity. Some researchers propose to replace the dependency-tracking scheduler with a ticket-based approach (cite: SWAN). Other groups propose to implement the parts of the scheduler in hardware, such as the dependency tracking mechanism (cite: HTSS, Nexus++) or the ready queues and task scheduler (cite: Carbon).
- Efficient debugging/tracing mechanisms/tools (**Jesus**)
- Nested parallelism (recursive formulation) (**Hatem**)
- Work stealing (**Miquel**): Work stealing is how idle cores obtain work in order to balance the load across processors. When work is stolen, the working set of the task is also "stolen". Thus a work steal operation usually results in a data migration. The impact and optimization of work stealing depends on the particular case:
  - Obviously, work stealing can only be optimized for locality if tasks feature sufficient internal locality and are to an extent memory bound. If tasks have very bad locality or the miss rate to main memory is already negligible, then work stealing has little effect beyond the overhead of the stealing itself.
  - When tasks have locality of reference, things become interesting. 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 sequential task in the set of cores sharing the cache and only resorting to global work stealing when the task queues become empty.
  - In general we want to minimize the number of work steals. This works well if the application can be rapidly partitioned into almost equivalent sets of work that can then proceed independently. This works well 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 generally not efficient. This is 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 worksharing partitioning of the loop can be a more efficient method.

  - A big challenge is for the runtime to learn the hierarchical data properties of an application and exploit them to generate efficient schedules. Classical random work stealing (e.g. Cilk-like) does not do any attempt to exploit this. Socket-aware policies exist (eg, Qthread) that perform hierarchical work stealing: 1) first among cores in a socket and 2) then among sockets. Some programming models expose an API that allows programmers specify on which NUMA node/socket a collection of tasks should be executed (cite: OmpSs). Configurable work stealers which can be customized with *scheduling hints* have also been developed (cite: Pheet/PPoPP'13). Finally, a more extreme option is to allow the application programmer to attach a custom work stealing function to his application (e.g., MassiveThreads/ROSS13). How to effectively specify this information in a programmer-friendly way is an open topic of research.

- Scheduling priority, DAG critical path (**Romain**)
- Socket-aware scheduling (**Hatem**)
- Detection of overlapped memory-region (**Jesus**)
- API Standardization (**Romain**)
- DAG Composition (**Hatem**)
- Handling data locality in presence of co-processors and accelerators (**Jesus**)

## 6.3 The State-of-the-Art

A few examples of work in that area (**Hatem**)

- OmpSs
- Charm++
- QUARK
- StarPU
- PaRSEC
- SuperMatrix
- ParalleX
- OCR

APPLICATION DOMAINS (**Hatem**)

- Dense linear algebra
- Computational astronomy
- Uncertainty Quantification
- Seismic
- Weather/Climate modeling
- Krylov-based solvers
- Nbody problems

## 6.4 Discussions

AREAS OF AGREEMENT (**Romain**)

- 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 hardware dynamism. [TODO: Maybe phrase it to say that the assembly interface does not guarantee anything in terms of performance and/or energy].

  In light of this, toolchains will become more and more 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 programming model: dynamic task based programming models need to be a part of the solution.

- Task base programming models, however, rely on the computation and data being split into chunks ("tasks" for the computation which implies a size of the data these tasks operate on). The size of these chunks (the granularity) is difficult to determine as it needs to balance the overheads of the task-based system with the need to expose sufficient parallelism to fully occupy the ever increasing number of parallel resources. A static granularity will be sub-optimal for future machines [TODO: Do I need to add more to this]

AREAS OF DISAGREEMENT (**Romain**)

- There is agreement on the fact that a standardization needs to happen for task based programming model but there is disagreement as to the level at which this standardization needs to 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 what can be expressed by the user at the programming model level [TODO: Jesus, I would like to understand your position a bit more on this so I can clarify things a bit more].

- Another area of disagreement is how best to deal with the expression of granularity; specifically, is it better for the programmer to break-down tasks and data 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 a sort of "recipe" for the runtime to be able to stich smaller tasks or chunks of data into larger ones. There is debate as to which approach is simpler and more likely to yield positive results.

## 6.5 Research Plan

Identify Gaps / What is missing (**Miquel**)

- Better understanding of the performance features of task-parallel systems. Need to convince developers of the value of these systems.
  - Better performance tools. Better understanding of the impact of locality
  - Case studies that show performance benefit in certain conditions (eg, in the presence of faults?). Focus on data aspects, i.e. show that a good management can be done even when there are a lot of work steals.

# Chapter 7

# System-Scale Data Locality Management

Parallel computers are becoming more and more complex. Indeed, they 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. Today it is already more challenging to write an application that efficiently accesses its data than one that efficiently processes the data. In other words arranging bytes is more complex than computing flops. As it is expected that the memory per core will decrease in the coming years this problem will become even more important and harder to attack.

To address the data locality issue system-wide, one approach consists of working on the application ecosystem and how the application is executed (e.g., its interaction with the batch-scheduler and the storage, optimization on the way the application uses the resources, its relation with the topology, or the interaction with other executing application). More precisely, the important issue lies in the strong need for new models and algorithms, 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 are tremendous efforts for optimizing an application statically in the literature. Among these are data layout optimizations, compilation optimization, and parallelism structuring. However, once an application is written and compiled there are several additional factors that play an important role, that could not be known before the execution and have a dramatic impact on its performance. Among these factors are:

- The allocated resources for the specific run and their interconnection,
- the other running applications and the network traffic they induce,
- the topology of the target machine,
- the data accessed by the application and their location on the storage system,
- dependencies of the input on the execution,

and many more.

It is important to remark that these runtime factors are orthogonal to the static optimizations that can be performed. In other words, no matter how the application is written and optimized, the way it is executed and the interaction with its ecosystem have a huge impact on its behavior. For instance, it has been shown that the way resources are allocated to an application plays a crucial role in the performance of the execution. Recent works [**cui2013accelerating**, **kramer13**] have demonstrated that a non-contiguous allocation can slowdown the performance by more than 30%. However, a batch scheduler cannot always provide a contiguous allocation and even in the case of such allocation the way processes are mapped to the

allocated resources have a big impact on the performance [**DBLP:conf/ics/ChenCHRK06**, **jm10**]. 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.

Futhermore, 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. A leading contender for a new interconnect is a multi-level direct network such as Dragonfly [**ibm-percs-network**, **4556717**]. Such networks are formed from highly-connected parts, placing every node within a few hops of all others. This may benefit unstructured communications that often occur in graph algorithms, but limited connections between parts can be bottlenecks for structured communication patterns and the network topologies have not been completely specified. At the node level, a promising approach for fully utilizing higher core counts on next-generation architectures is over-decomposed task parallelism [**Kale:1993:CPC:165854.165874**], 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, models of the application and algorithms and tools to optimize its execution with its whole ecosystem. The literature provides a lot of models and abstraction on how to write a parallel code. However, even with sufficient parallelism, future applications may not scale due to the data traffic and coherence management. Current models and abstraction are geared towards computations and ignore 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 control locality. Modeling the future large-scale parallel machines will require work in the following directions: (1) better describe the memory hierarchy (2) provide an integrated view with the nodes and the network (3) exhibit qualitative knowledge and, (4) provide 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 [**hoefler-mpi-2.2-scal-topo**]. However, while this interface is a good first step, it is essentially limited to BSP-style applications. To optimize execution at system scale, we need to provide mechanisms, tools and algorithms that are based on the environment (given by the network model) and the application requirements (given by application models and abstractions). Based on that, several optimizations can be performed such as: improving storage access, mapping processes onto resources based on their affinity [**hjm14**, **DBLP:conf/ics/HoeflerS11**, **Navauxandal2009**], 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.

## 7.2 State of the Art

Concerning topology mapping TreeMatch [**jmt14b**], provides mapping of processes onto computing resources in the case of a tree topology (such as current NUMA nodes and fattree network). LibTopoMap [**DBLP:conf/ics/HoeflerS11**] addresses the same problem as TreeMatch but for arbitrary topology such as torus, grid, and others. The general problem is NP-hard and can thus only be approximated. However, several specialized versions of the problem can be solved near-optimal in polynomial time, for example, mapping Cartesian topologies to Dragonfly networks [**prisacari-dragonfly-mapping**].

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 [**scotch-man**] or ParMetis [**karypis2003parmetis**] can address the problem even though they might require more precise information than specific tools and do not always provide good solutions [**jmt14b**]. After processes are allocated to an application, Zoltan2 [**zoltan2**, **drl+14**] is a toolkit that can map processes to the allocated resources depending on the geometry of the target machine and process affinity.

Hardware Locality (HWLOC) [**hwloc**] is a library and a set of tools aiming at discovering and exposing the hardware topology of machines, including processors, cores, threads, shared caches, NUMA memory nodes and I/O devices. NETLOC [**netloc**] is a network model extension of HWLOC to account for locality

requirements of the network. 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 [**geimer'ea:2010:scalascaarchitecture**]. It can also be done by tracing data exchange at the runtime level with a system like OVIS [**ovis**, **SystemMonitoring**], 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 [**yoo2003slurm**], OAR [**capit2005batch**], LSF [**zhou1992lsf**] or PBS [**henderson1995job**] 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.

**TODO:**

- Storage (e.g. parallel file system (???)), etc.

## 7.3 Discussion

To address the locality problem at system scale, several challenges are required to 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 manage 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. However, some optimizations can be conflicting. It is required to identify how the different approaches interact with each-other and propose integrated solutions that provide a global optimizations crossing the different layers.

Ultimately, the validation of the models and solutions to the key points and challenges above will be a key challenge.

## 7.4 Research Plan

To solve the above problems, we propose co-design between application communication models, specific network structures, and algorithms for allocation and task mapping. We will determine the effect of over-decomposition on application communication. To benefit structured communication patterns on multi-level networks, we will complete the topology (specifying pairs of nodes to be connected) and techniques to optimize allocations and task mappings.

# Chapter 8

# Conclusion