Trends in Data Locality Abstractions for HPC Systems

Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, *Senior Member, IEEE*, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H J Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, and Miquel Pericás

Abstract—The cost of data movement has always been an important concern in high performance computing (HPC) systems. It has now become the dominant factor in terms of both energy consumption and performance. Support for expression of data locality has been explored in the past, but those efforts have had only modest success in being adopted in HPC applications for various reasons. them However, with the increasing complexity of the memory hierarchy and higher parallelism in emerging HPC systems, locality management has acquired a new urgency. Developers can no longer limit themselves to low-level solutions and ignore the potential for productivity and performance portability obtained by using locality abstractions. Fortunately, the trend emerging in recent literature on the topic alleviates many of the concerns that got in the way of their adoption by application developers. Data locality abstractions are available in the forms of libraries, data structures, languages and runtime systems; a common theme is increasing productivity without sacrificing performance. This paper examines these trends and identifies commonalities that can combine various locality concepts to develop a comprehensive approach to expressing and managing data locality on future large-scale high-performance computing systems.

Index Terms—Data locality, programming abstractions, high-performance computing, data layout, locality-aware runtimes

1 Introduction

The computing industry has entered a period of technology transition as we strive for the next 1,000 x performance improvement over the previous generation of petaflops-scale computing platforms. Over the past 30 years, we have come to expect a 1,000 x increase in HPC system performance via technology scaling. With the end of conventional improvements to technology (Dennard scaling), which started in approximately 2004, single processing core performance has ceased to improve with each generation. The industry has adopted a new approach to performance scaling by packing more cores into each processor chip. This multicore approach continues to drive up the theoretical peak performance of the processing chips, and the computing industry is on track to have chips with thousands of cores by 2020 [1]. The other consequence of the new technology scaling trend is that the energy efficiency of transistors is improving as their sizes shrink, but the energy efficiency of wires is not improving. Therefore,

- *D. Unat is with the Department of Computer Engineering, Koç University, Istanbul 34450, Turkey. E-mail: dunat@ku.edu.tr.*
- *A. Dubey is with Argonne National Laboratory, Lemont, IL 60439. E-mail: adubey@anl.gov.*
- *T. Hoefler is with ETH Zürich, Zürich 8092, Switzerland. E-mail: htor@inf.ethz.ch.*
- *J. Shalf is with Lawrence Berkeley National Laboratory, Berkeley, CA 94720. E-mail: jshalf@lbl.gov.*
- *M. Abraham is with KTH Royal Institute of Technology, Solna 17121, Sweden. E-mail: mjab@kth.se.*
- *M. Bianco is with Swiss National Supercomputer Centre, Lugano 6900, Switzerland. E-mail: mauro.bianco@cscs.ch.*
- *B. Chamberlain is with Cray Inc., Seattle, WA 98164. E-mail: bradc@cray.com.*
- *R. Cledat is with Intel Cooperation, Santa Clara, CA 95050.*
- *C. Edwards is with Sandia National Laboratories, Albuquerque, NM 87185. E-mail: hcedwar@sandia.gov.*
- *H. Finkel is with Argonne National Laboratory, Argonne, IL 60439. E-mail: hfinkel@anl.gov.*
- *K. Fuerlinger is with Ludwig-Maximilians-Universität München, Munich D-80538, Germany. E-mail: Karl.Fuerlinger@nm.ifi.lmu.de.*
- *F. Hannig is with University of Erlangen-Nuremberg, Erlangen 91058, Germany. E-mail: frank.hannig@fau.de.*

Manuscript received 2 June 2016; revised 12 Apr. 2017; accepted 14 Apr. 2017. Date of publication 10 May 2017; date of current version 13 Sept. 2017. (Corresponding author: Didem Unat.)
Recommended for acceptance by U.V. Catalyurek.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2703149

- *E. Jeannot is with INRIA Bordeaux Sud-Ouest, Talence 33405, France. E-mail: emmanuel.jeannot@inria.fr.*
- *A. Kamil is with University of Michigan, MI 48109, and also with Lawrence Berkeley National Laboratory, Berkeley, CA 94720. E-mail: akamil@umich.edu.*
- *J. Keasler is with Lawrence Livermore National Laboratory, Livermore, CA 94550. E-mail: keasler1@llnl.gov.*
- *P. Kelly is with Imperial College London, London, United Kingdom. E-mail: p.kelly@imperial.ac.uk.*
- *V. Leung is with Sandia National Laboratories, Albuquerque, NM 87185. E-mail: vjleung@sandia.gov.*
- *H. Ltaief is with King Abdullah University of Science and Technology, Thuwal 23955, Kingdom of Saudi Arabia. E-mail: Ltaief_hatem@yahoo.fr.*
- *N. Maruyama is with RIKEN, Kobe, Hyogo 650-0047, Japan. E-mail: nmaruyama@riken.jp.*
- *C. J. Newburn is with Nvidia Corporation, Santa Clara, CA 95050. E-mail: chris.newburn@intel.com.*
- *M. Pericás is with Chalmers University of Technology 41296, Göteborg, Sweden. E-mail: miquelp@chalmers.se.*

the relative cost of computation to data movement has become further skewed in favor of computation.

By 2018, further improvements to compute efficiency will be undercut by the energy required to move data to the computational cores on a chip [2] and are manifested in substantial bandwidth tapering at every level of the memory and communication hierarchy. Bandwidth tapering has been a challenge since the dawn of cache hierarchies, and the remedies (loop blocking, strip-mining, tiling, domain decomposition, and communication optimizations/topology mapping) have been studied for decades. Although the research community has developed extensive compiler and library solutions, only a fraction of these are available in general-purpose systems. Furthermore, with the increase in the parallelism and memory hierarchy going from system to node to compute unit level, the already difficult task of managing parallelism has become much more complex. The solutions for bandwidth tapering challenges now need their counterparts at the intranode, internode and global system level communication. Moreover, the dual tension of increasing levels of parallelism and core heterogeneity create an intractable explosion in complexity. There is an urgent need for higher level of abstraction in order to shield the applications developers from this complexity and reduce the effort needed to port codes to different computing platforms.

One critical abstraction needed for future tractability of the application space is *data locality*; a way of expressing computations so that information about proximity of data to be used can be communicated to the optimizing software stack. The impact of data locality optimization has moved from being a tuning option to a central feature of code writing to get *any* performance improvement at all. There needs to be a formalization of commonly used approaches to make the implementations reusable and parametrizable so that a common data abstractions can be used portably and flexibly across multiple architectures, without manually re-tuning for each new system. The need for performance portability is on the rise in direct correlation with the rise in platform heterogeneity.

Application developers have begun to realize the enormity of the challenge facing them and have started a dialogue with researchers in programming abstractions to look for effective solutions. This development has opened up a real opportunity for the higher level abstractions to gain traction in the applications communities, especially when the application developers are kept in the loop. We conducted a series of workshops on the topic of programming abstractions for data locality for high performance computing (HPC) that gather practitioners and researchers from all applicable areas, including the computational scientists from multiple science domains [3], [4], [5]. This survey paper distills the outcomes of the series thus far. The objective of this effort is to facilitate the development of this critical research area by; (1) defining a common terminology to facilitate future exchange of ideas in the field, (2) describe the current trends in various research domains that directly influence data locality, and (3) recommend directions for future research. We do not claim to have solved or covered every aspect of this enormous challenge, however, the interdisciplinary exchanges between domain scientists and computer scientists at the workshop, and dissemination of the gathered knowledge plays an essential role in maintaining forward progress in this area.

Locality can be expressed and managed at various level in the computational ecosystem. The bulk of the paper is divided into sections corresponding to research areas that are actively engaged in exploring the issues of locality. Section 2 defines common terminology used to describe the state of the art in concerned research areas. We examine data locality in the context of data structures and library support in Section 3, language and compiler support in Section 4, runtime approaches in Section 5, and systems level support in Section 6. All of these research areas have one goal in common, to help applications effectively use the machine for computational science and engineering. Section 7 serves two purposes, it describes challenges and expectations from application developers, which in turn provide perspective and cohesiveness to the research areas discussed in earlier sections. We summarize our findings in Section 8.

2 Terminology

We begin by defining commonly used terminology in describing efforts aimed at addressing data locality.

Data locality is indicative of how close data is to where it needs to be processed, shorter distance imply better data locality. A *data structure* is the organization of a *data type* onto some particular memory architecture. The memory subsystem is composed of several memory arrays, which can be defined as *memory spaces*. Not all memory spaces can be managed directly by the programmer (e.g., caches). However, new architectures tend to have multiple user-manageable memory spaces with varying performance characteristics and usage restrictions (e.g., constant memory of GPUs).

Application performance is constrained by both time and energy costs of moving data in service of the computation, which is directly affected by the data access pattern. The data access pattern is a composition of data layout, data decomposition, data placement, task placement, and how the parallel tasks traverse the data structure. Fig. 1 illustrates these concepts.[1] Given a data type and memory space (e.g., an array of memory cells), we define *data layout* as an injective mapping from the elements of the data type to the cells of the single memory space. By extension, we define a *distributed layout* as the mapping of the elements to multiple memory spaces. Usually a layout can be considered a parameter of a data structure. Layout affects data access patterns, and hence performance, therefore, selecting an appropriate map to data structures is important.

Data decomposition is the way that data is partitioned into smaller chunks that can be assigned to different memory spaces for introducing data parallelism or improving data locality. *Data placement* is the mapping of the chunks of data from a domain-decomposed data structure to memory spaces. *Task placement* is the assignment of threads of execution to a particular physical processor resource and its related set of memory spaces. Many contemporary programming environments do not offer an automatic method

[1] The figure is inspired by Fuchs and Fuerlinger [6].

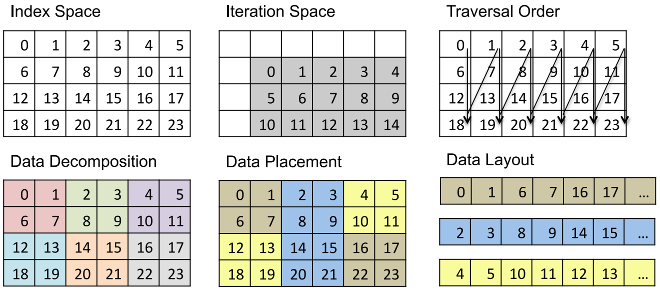

Fig. 1. illustration of concepts that are important for data locality for a dense two dimensional array. Example iteration space, traversal order, decomposition, data placement and data layout are shown.

to directly relate the task placement to the data placement, aside from loose policies such as first touch memory affinity. *Index space* defines the index domain for data. *Iteration space* refers to the set of points in a multi-dimensional loop nest irrespective of traversal order. The dimensionality of the iteration space is typically defined in terms of the number of loop nests (e.g., an N-nested loop defines an N-dimensional iteration space). *Traversal order* indicates the order in which the loop nest visits these indices.

A *tiling* layout is often used to exploit the locality of hierarchical memories. This layout can be viewed as adding additional dimensions to the iteration space in order to identify the tiles and the elements within the tiles. Thus a fully tiled D-dimensional array will have $2 * D$ dimensions. An array implementation may retain a D-dimensional interface by using integer division and modulo (or integral bit shift and mask) operations to map a D-dimensional tuple to a $2 * D$ tuple and then to the tile and element. Algorithms are usually written in terms of the higher-dimensional layout, so loop nests are deeper than D levels. A benefit of tiling is because of their explicitly defined sizes, the compiler can perform optimizations that would not be available otherwise.

An array may have a recursive tile layout where the member is itself a tile, for example, *hierarchical tiled arrays* [7]. Such hierarchical layouts may be used to better manage the locality of data in a hierarchy of memory spaces; for example, the top level mapped to a set of distributed memory spaces and the nested levels corresponding to the cache hierarchy within a local memory space.

A library is an *active library* if it comes with methods for delivering library-specific optimizations [8], for example, template metaprogramming in C++ or lightweight modular staging in Scala. An *embedded domain-specific language* (DSL) is a technique for delivering a language-based solution within a host general-purpose language. *Directive-based language extensions* such as OpenMP and OpenACC use annotations to drive code transformation. *Object visibility* determines whether data objects are visible globally from anywhere or are visible only locally on the node. *Multiresolution language philosophy* is a concept in which the programmers can move from language features that are more declarative, abstract, and higher level to those that are more imperative, control oriented, and low level as required by their algorithm or performance goals.

The task-based runtime implements a *scheduling strategy* that imposes a partial ordering on the tasks within each queue and optionally among queues as well. The runtime may use the specific knowledge of the underlying machine. Scheduling can be *static* so that data locality can be enforced. Scheduling can also be *dynamic*, using *work sharing* where a single queue lists all the work to be done, or *work stealing*, where a queue is assigned to each computing resource and an idle process can steal work from another's queue.

3 Data Structures and Library Support

The traditional view of flat memory is not consistent with the actual memory subsystems of modern computers. Memory is organized into banks and NUMA regions, cache hierarchies, specialized memories such as scratchpad storage, read-only memories, etc. This disconnect makes development of efficient data structures very challenging.

3.1 Key Points

We identified two design principles as important and desired by application programmers: Algorithmic execution dependencies and separation of concerns. Note that, in this Section, we use the term *application* as the user of a library, which in a layered software architecture may be another, higher-level, library or domain-specific language.

3.1.1 Algorithmic Execution Dependence

In general determining what layout an algorithm should use is difficult. The implementation of an algorithm is written by accessing data elements through some interfaces, for instance using a tuple of indices to access a multi-dimensional array. An implementation can leverage temporal locality by accessing the same data elements multiple times, and spatial locality by accessing nearby data elements, where *nearby* is here a logical concept related to the abstract data type, and not to the implementation of the data structure. We refer to this locality as *algorithmic locality*. The optimal data locality of the implementation is reached when the data structure layout, in memory, lets the algorithm find the corresponding elements in the closest possible location (relative to the processing elements used by the threads). Different implementations have different algorithmic localities and therefore require different data structures layouts.

Typically, the algorithmic locality also depends upon input, so the layout can be chosen only for the likelihood of locality. For certain applications such as linear algebra or finite-difference stencils, the trace can be determined by a few simple parameters such as array sizes, which can be used to determine the best layout effectively. These cases are well represented in HPC applications, and several solutions have been implemented to exploit locality, especially in computing nodes. Multi-node implementations require the integration with the system-scale locality management discussed in Section 6.

Note that we highlight the usefulness of picking the best layout by analyzing the abstract algorithm instead of restructuring an existing code. The latter would usually lead to convoluted, non-portable, and unmaintainable code. A library of algorithms and data structures should enable an efficient coupling of algorithms and data structures mappings. Libraries differ on how this coupling can be specified or found, and how concerns are separated between application programmers and library programmers.

3.1.2 Separation of Concerns

Separation of concerns is a fundamental motivation for developing libraries of algorithms and data structures to be used within applications. A well-defined separation of concerns clearly identifies who (library or application) is responsible for what. Our focus is on managing data locality, so we limit this *what* to the mapping of data structures to memory spaces and algorithms to execution spaces.

Separating Explicitly by Threads. A parallel-enabling library should provide concurrent threads. Different solutions differ on the guarantees they provide for safety, progress, and placement of threads. Low level libraries like *pthreads* leave all these concerns to the application, others offer different levels of guarantees depending on the applications (and programmers) they are targeting. An emerging trend in HPC is to delegate responsibilities to libraries, compilers, and runtime systems.

Separating by Parallel Patterns. A common separation is a parallel pattern and code body that is executed within a pattern. This can also be explained as decoupling loop iteration space from the loop body itself. For example, a loop over a range of integers (e.g., FORTRAN do-loop, C/C++ for-loop) is a serial loop pattern that executes a loop body (codelet). Depending on the inter loop actions performed by the loop body this serial loop pattern can often be simply translated to the *foreach*, *reduce*, or *scan* (i.e., prefix sum) data-parallel pattern. Other patterns are possible, as stencil-like iterations on multidimensional arrays.

In this strategy the application is responsible for identifying the parallel pattern and providing the codelet that is (thread) safe to execute within that pattern. The library is then responsible for mapping execution of that codelet onto the execution space according to the pattern and for managing the pattern's inter thread interactions. For example, a parallel reduce requires thread-local temporary values and inter thread reduction of those temporary values.

Separating Location Policies. The mapping of application code bodies via parallel patterns has a spatial and temporal scheduling consideration: For example, on which core or when the execution of the code body will occur and whether the mapping is static or dynamic. We label the set of parameters that govern the answers to these questions as an *location policy*. The number and extensibility of such parameters that a parallel-enabling library has and exposes define the flexibility of that library.

Separating Data Structure Layout. Within a memory space a computer language hard codes the layout of their data structures, for example FORTRAN arrays, C/C++ arrays, or C/C++ classes. A library can define data types that abstracts the layout specification from the data type mapping. The parameter(s) of this specification may be static (defined at compile time) or dynamic (defined at runtime) and affect the compiler's ability to optimize code accordingly. A library can also define data types with distributed layouts that span multiple memory spaces and can define operations for moving data between memory spaces. The flexibility of this strategy is limited by the layout capabilities and their extensibility.

Integrating Separations. These separation-of-concerns strategies can provide significant flexibility through high-level abstractions (spaces, patterns, policies, layouts). However, the data locality and thus the performance of a parallel algorithm is determined by the mappings (data and execution) to hardware that are implemented by these abstractions. Thus, the integrated set of parameters for these abstractions must be chosen appropriately for the algorithm and underlying hardware in order to achieve locality and thus performance. A well-designed parallel-enabling library will provide and expose these parameters such that changing the underlying hardware requires no changes to the application codelets and trivial changes to the abstractions' parameters. Such parameter changes could even be chosen automatically based on the target hardware architecture.

3.2 State of the Art

Within the confines of existing language standards one is constrained to leveraging market breadth of the supporting tool chain (e.g., compilers, debuggers, profilers). Wherever profitable, the research plan can redeem existing languages by amending or extending them (e.g., by changing the specifications or by introducing new APIs). Examples include Kokkos [9], TiDA [10], GridTools [11], hStreams [12], and DASH [13].

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 abstract machine 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 algorithmic 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 efficient mapping on cores. Like Kokkos, the metadata describing the layout of each array is carried throughout the program and into libraries, thereby offering a pathway to better library composability. TiDA is currently packaged as Fortran and C++ libraries and adopted by the BoxLib AMR framework [14].

GridTools provides a set of libraries for expressing distributed memory implementations of regular grid applications, such as stencils on regular and icosahedral grids. It is not meant to be universal, in the sense that non regular grid applications should not be expressed using GridTools libraries. 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 [15]. It expects the application to use its patterns. The hStreams library provides mechanisms for expressing and implementing data decomposition, distribution, data binding, data layout, data reference characteristics, and location policy on heterogeneous platforms. DASH is built on a one-sided communication substrate and provides a Partitioned Global Address Space (PGAS) abstraction in C++ using operator overloading. The DASH abstract machine is basically a distributed parallel machine with the concept of hierarchical locality. It is a very general library designed to address scaling of applications at system scale, while leaving the managing of threads in a node to the application.

Table 1 offers a quick comparison between the libraries presented in this Section. This is intended to be a simple sketch and should not be treated as a comprehensive comparison of these quite complex and rich libraries.

Clearly, no single way of treating locality concerns exists, nor is there 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. The opportunity arises for naturally building higher-level interfaces by using lower-level ones. For instance, TiDA or DASH multidimensional arrays could be implemented using Kokkos arrays, or GridTools parallel algorithms could use the DASH library and Kokkos arrays for storage. This is a potential benefit from interoperability that arises from using a common language provided with generic programming capabilities. One outcome of the survey is to initiate efforts to explicitly define the requirements for a common runtime infrastructure that could be used interoperably across these library solutions.

TABLE 1
Comparison of the Libraries Discussed in Section 3.2 with Respect to the Separation of Concerns

Lib.	Scale	Threads	Patterns	Policies	Layout
Kokkos	Node	Yes	Yes+	Yes	Yes
TiDA	Node+	Yes	No	Yes	Yes
GridTools	Node	Yes	Yes	Yes	Yes
hStreams	Node	Yes	No	Yes	Future
DASH	System	No	Yes+	No	Yes

"Scale" refer to the ability of the library to handle inter-node communication. TiDA has ongoing development to do so. "Threads" tells if the library handles low level thread managing. In "Pattern", a "Yes+" entry symbolizes the fact that the library provide patterns but also direct managing by the application.

4 Language and Compiler Support for Data Locality

While significant advances have been seen in libraries for existing programming languages, especially C++, in facilities that allow for data-locality optimizations, significant limitations remain in what can be accomplished with libraries alone. C/C++ and Fortran, which dominate the high-performance computing landscape, offer limited facilities for compile-time introspection. By contrast, custom languages are designed to present language-intrinsic abstractions that allow the programmer to explicitly expose parallelism and locality. Such abstractions in turn significantly simplify compiler analysis and optimization and also assist locality management at both runtime and system levels discussed in Sections 5 and 6.

4.1 Key Points

The following features are key to understanding and designing for data locality from the language and compiler perspective.

4.1.1 Object Visibility

One of the most significant axes in the relevant design space is the choice between local-by-default and global-by-default object visibility. Local-by-default visibility (or local-only visibility) is familiar to any user of MPI, and message-passing is still often an effective way to optimize for data locality. MPI, however, is not the only common example; most GPU-targeted programming models (OpenCL, CUDA, etc.) explicitly represent local memory domains and force the programming to arrange any necessary transfers. The disadvantage of local-by-default visibility, however, is that it tends to be cumbersome to use. Furthermore, programmer productivity can be low because data locality must be managed in every part of the code, even where performance is not critical or the necessary management logic is boilerplate.

Two commonplace language-design techniques improve upon this local-by-default situation. The first, exemplified by Loci [16], provides a programming environment in which declarative annotations, and other functional programming techniques can be employed to drive the automated generation of the communication-management and task-scheduling logic. Declarative solutions tend to have much greater semantic freedom than those embedded in imperative programming languages, allowing more invasive transformations between the input and the resulting implementation. The disadvantage of such systems tends to be generality, and such systems tend to be domain specific.

The second commonplace technique to improve upon the local-by-default situation is to move toward a global-by-default model, at least for certain classes of objects. PGAS models, now widely available from Fortran Co-Arrays [17], Chapel [18], Julia [19], and many other languages, provide some differentiation between local and global objects but allow global access without explicit regard for locality considerations. The compiler and/or runtime system might optimize layout and placement of objects based on their global access pattern, but the degree to which this optimization can be usefully done is still an open question.

On the far end of the spectrum are solutions that do not expose any data-locality information to the user directly but depend solely on compilers and runtime libraries to perform any desirable data-locality optimizations. OpenMP falls into this camp, and current experience suggests that the more advanced data locality optimizations sought might prove indefinitely out of reach for its trivial user-facing locality model. One might argue that such optimizations are more important for tools not restricted to the shared-memory part of the hierarchy; but experience suggests that between NUMA and the proliferation of cores per node, data-locality optimizations are important both on-node and over distributed-memory systems.

4.1.2 Requirements

Effective abstractions for data locality need to have low overhead and high-level semantic information, including information about data dependencies needed by the compiler's optimizer and runtime library. Dealing with side-effects is key to dependence analysis, and this is an area in which declarative solutions and novel languages often hold distinct advantages because traditional languages make conservative assumptions about the behavior of external function calls. The abstractions need to cover data movement, be it automatic (via caching or similar) or explicit; different levels of control are desirable for different use cases. Profiling, auto-tuning and user feedback are important additions to purely static determinations, and while user-provided hints will remain important, only tools using these more automated measurement-driven techniques are likely to scale to large codes. Finally, the abstractions have to be composable as no convergence exists yet on what are the most productive paradigms for portable high-performance codes. While the hierarchical nature of modern hardware is well established, the extent and semantics of exposure to the users are not yet settled; and the optimal answer may be domain specific. Some solutions may be specific to parts of the hierarchy; and an overall solution may require separate tools for different parts of the solution, making composability a key requirement.

A generic goal, at a programmatic level, is to encourage programmers to expose all available parallelism in their source code and let the compiler and/or runtime system choose how to best use that freedom on a particular hardware architecture. In practice, this means that the parallelism often needs to be coarsened into larger task units. For example, even if all discretized grid points are independent, having one dispatched task per grid point is likely impractical. The space of potential coarsenings often grows quickly, and so some combination of profile-driven feedback and auto-tuning, user-provided grouping preferences, and heuristics are necessary in practical tools. We also note that even within a particular coarsening scheme, task-execution ordering is important to preserve locality, and ordering considerations must be part of the relevant cost model.

4.1.3 Adoption

Regardless of the flavor of the solution, widespread adoption can be supported only if the implementations are treated as proper software engineering projects. It is critical to have invested stakeholders because these projects often involve long time horizons and considerable infrastructure work. They also need a coherent support model and quick bug fixes. Adoption is also greatly enhanced for tools with a small initial learning curve and those that enable incremental transitioning from existing codebases to new ones.

4.2 State of the Art

Advances are being made in both C++ and FORTRAN. In C++ memory-aliasing attributes and parallel-algorithm abstractions are being designed, while in FORTRAN PGAS-style Co-Arrays [20] are now part of the standard. New languages, both general-purpose languages such as Chapel and Julia and domain-specific languages such as Loci, have production-quality implementations and growing user communities. Custom languages have also benefited from strong community compiler infrastructures, which enable functionality reuse. Higher-level tools need standardized, or at least well-supported, lower-level interfaces upon which to build. We also note that the line between the language and library is fuzzy in terms of capability and responsibility, and successful programming models often combine a targeted set of language capabilities with strong libraries built on top of those facilities.

Chapel [18] is an emerging language that uses 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 runtime queries. Chapel follows the PGAS philosophy, supporting direct access to variables stored on remote locales based on traditional lexical scoping rules. Chapel also follows the multiresolution philosophy by supporting low-level mechanisms for placing data or tasks on specific locales, as well as high-level mechanisms for mapping global-view data structures or parallel loops to the locales. Advanced users may implement these data distributions and loop decompositions within Chapel itself and can even define the model used to describe a machine's architecture in terms of locales.

X10 [21] is another PGAS language that uses *places* as analogues to Chapel's locales. In X10, execution must be colocated with data. Operating on remote data requires spawning a task at the place that owns the data. The user can specify that the new task run asynchronously, in which case it can be explicitly synchronized later and any return value accessed through a future. Thus, X10 makes communication explicit in the form of remote tasks. Hierarchical Place Trees [22] extend X10's model of places to arbitrary hierarchies, allowing places to describe every location in a hierarchical machine.

Unified Parallel C (UPC), Co-Array Fortran (CAF), and Titanium [23] are three of the founding PGAS languages. UPC supports global-view data structures and syntactically invisible communication while CAF has local-view data structures and syntactically evident communication. Titanium has a local-view data model built around ZPL-style multidimensional arrays [24]. Its type system distinguishes between data guaranteed to be local and data that may be remote, using annotations on variable declarations. On the other hand, access to local and remote data is provided by the same syntax. Thus, Titanium strikes a balance between the HPF and ZPL approaches, making communication explicit in declarations but allowing the same code fragments to operate on local and remote data.

Recent work in Titanium has replaced the flat SPMD model with the more hierarchical Recursive Single-Program, Multiple-Data (RSPMD) model [25]. This model groups together data and execution contexts into teams that are arranged in hierarchical structures, which match the structure of recursive and compositional algorithms and emerging hierarchical architectures. While the total set of threads is fixed at startup as in SPMD, hierarchical teams can be created dynamically, and threads can enter and exit teams as necessary. Titanium provides a mechanism for querying the machine structure at runtime, allowing the same program to target different platforms by building the appropriate team structure during execution.

Other work has been done to address the limitations of the flat SPMD model in the context of Phalanx [26] and UPC++ [27], both active libraries for C++. The Phalanx library uses the Hierarchical Single-Program, Multiple-Data (HSPMD) model, which is a hybrid of SPMD and dynamic tasking. The HSPMD model retains the cooperative nature of SPMD by allowing thread teams, as in RSPMD, but it allows new teams of threads to be spawned dynamically.

Unlike SPMD and RSPMD, the total set of executing threads is not fixed at startup. Both RSPMD and HSPMD allow expression of locality and concurrency at multiple levels, although through slightly different mechanisms, allowing the user to take advantage of hierarchical architectures. The UPC++ library uses RSPMD as its basic execution model but additionally allows X10-style asynchronous tasks to be spawned at remote locations. This allows execution to be moved dynamically to where data are located and adds a further degree of adaptability to the basic bulk-synchronous SPMD model.

Compilations of both local-by-default and global-by-default languages can be facilitated with recent development in polyhedral analysis, which allows the compiler to model the iteration space and all data dependencies for so-called affine code regions. An affine region is a block of code where all loop iteration variables and array accesses can be modeled by affine functions in Presburger arithmetic [28]. The polyhedral program representation can be used to automatically parallelize programs [29] and more recently automatically map them to complex accelerator memory hierarchies [30], [31].

5 Task-Based Runtime Approaches for Data Locality

Traditionally, task-based runtime systems have been used to enable a problem-centric description of an application's parallelism while hiding the details of task scheduling to complex architectures from the programmer. This separation of concerns is probably the most important reason for the success of using runtime environment systems for task models. It enables developers to taskify their applications while focusing on the scientific algorithms they are most familiar with. This paradigm breaks the standard bulk synchronous programming model inherent to runtimes supporting many state-of-the-art languages (e.g., PGAS), as previously mentioned in Section 3. Programmers delegate all responsibilities related to efficient execution to the task scheduling runtime thereby achieving higher productivity and portability across architectures. In light of the growing importance of locality management, runtime systems will need to move past only considering task-centric attributes (load balance, etc.) to ones that take into account data-centric attributes (data movement, memory bandwidth, etc.).

5.1 Key Points

At a very basic level, a locality-aware runtime is responsible for mapping the abstract expression of tasks and data at the application level to hardware resources, both compute and memory. The important question, however, is where one draws the line between the programmer (or higher-level abstraction), the runtime and the hardware. Traditionally, hardware has managed a lot of the locality (through the use of cache), but this is shifting as, necessarily, hardware can implement only a limited number of schemes that may not be adapted to all application patterns. Although the exact borders of a locality-aware runtime remain the subject of healthy research, researchers agree that with exascale systems, locality-aware runtimes will need greater cooperation between software and hardware.

5.1.1 Runtime Involvement in Data Locality

Data locality is relevant at three levels: The expression of parallelism in the application, the association of this expressed parallelism and the data, and the mapping of the tasks and data to computing and memory resources. Parallelism can be expressed in either a data centric or a task centric view. In the former case, parallelism is expressed mostly through the chunking of data into subsets that can independently be operated on, whereas in the latter case, parallelism is obtained through the chunking of the computation into independent subsets. The expression of parallelism is usually done outside the runtime either directly by the programmer or with the help of higher-level toolchains.

Whether the application has been divided in a task-centric or a data-centric manner, data and tasks need to be respectively associating tasks and/or data with the chunks and making runtime aware of them with the respective task chunks and data chunks identified in the first step. Whether the chunking is task-centric or data-centric, the association between specific data or task and their respective chunks must be made known to the runtime. The question of task and data granularity also comes up at this level, but additional information is available to answer it: At this stage, tasks and data are associated so the runtime has more information to determine the optimal granularity level taking into account both computing resources and memory resources. For example, the presence of vector units and GPU warps may push for the coarsening of tasks to be able to fully occupy these units; this will be, in turn, limited by the amount of memory (scratchpads, for example) that is close by to feed the computation units. Considerations such as *over-provisioning* which provides parallel slack and helps ensure progress, will also factor in granularity decisions at this level.

The third level involving data locality is scheduling. Previous efforts in resource allocation have frequently focused on improving the performance of an application for a particular machine, for example, by optimizing for cache size, memory hierarchy, number of cores, or network interconnect topology and routing. This is most efficiently done with static scheduling. Task-based runtimes that schedule statically may still make use of specific knowledge of the machine to perform their scheduling decisions. Static scheduling of tasks has several advantages over dynamic scheduling provided a precise enough model of the underlying computing and networking resources is available.

The static approach will become more difficult with increase in machine variability. Therefore, while static or deterministic scheduling enables offline data locality optimizations, the lack of a dependable machine model may make the benefits of dynamic scheduling, namely, adaptability and load-balancing, more desirable. Of course, in the presence of a machine model, the dynamic scheduling may also take advantage of the machine hardware specifications by further refining its runtime decisions. This holistic data-locality strategy at the system level is further explained in Section 6.

5.1.2 Abstractions for Locality

Task-based programming models are notoriously difficult to reason about and debug given that the parameters specified by the programmer to constrain execution (dependences) purposefully allow for a wide range of execution options. Certain task-based runtime systems, which allow the dynamic construction of the task-graph (such as OCR), only exacerbate these problems. Tools allowing the programmer to understand the execution of a task-based program need to be developed. This is particularly true when the decisions taken by these runtimes will be more and more impacted by data locality considerations that may be obscure to the end user.

These tools will need to cover two broad areas: 1) information on the execution flow of the application in terms of the tasks and data-elements defined by the user and, more importantly 2) information about the mapping of those tasks and data-elements to the computing and memory resources. For instance, OmpSs [32] and its dynamic runtime Nanos++ comes with substantial supports for performance analysis in the form of instrumentation tools for tracing and profiling of the task executions. In particular, the core instrumentation package *Extrae* [33] and the flexible data browser *Paraver* [34] provide useful insights on task scheduling and hardware usage in order to help the application developer identifying potential performance bottlenecks.

Nested or recursive algorithmic formulation as in cache oblivious algorithms [35] is a well-known technique to increase data reuse at the high levels of the memory hierarchy and, therefore, to reduce memory latency overheads. This often requires slight changes in the original algorithm. Nested parallelism can also enable a smart runtime to determine an optimal level of granularity based on the hardware available. This does require, however, that the runtime be made aware of the hierarchical nature of the tasks and data so that it may properly co-schedule iterations that share the same data. This approach is well suited for applications that have well defined data domains that can be easily divided (spatial decomposition, for example).

For algorithms that do not expose as much structure, nested parallelism may not be suited. A more generalized notion of closeness is needed: Programmers need to be able to express a certain commonality between tasks in terms of their data. The reason is that reducing data movement needs to happen at all levels of the system architecture in order to be effective: From the single CPU socket within a multi-socket shared-memory node up to multiple distributed-memory nodes linked through the high-performance network interconnect. This bottom-up approach highlights the need for programmers to expose various levels of closeness so that a runtime can map the application's abstract structure to the concrete hardware instance it is executing on, as detailed by the machine model.

Many numerical algorithms are often built on top of optimized basic blocks. For instance, dense eigensolvers require three computational stages: matrix reduction to condensed form, an iterative solver to extract the eigenvalues, and back transformation to get the associated eigenvectors. Each stage corresponds to an aggregation of several computational kernels, which may already be optimized independently for data locality. However, the ability to express locality constraints across the various steps is important. In other words, the way locality can be composed is important to express.

5.2 State of the Art

Standardization is widely considered desirable, but there is a disagreement as to the level at which this standardization should happen. One option is to standardize the APIs at the runtime level in a way similar to the Open Community Runtime (OCR) [36]. Another option is to standardize the interface of the programming model, as OpenMP or OmpSs [32] do. Currently there is no clear reason to decide for a particular scheme, so both approaches are being actively researched.

A locality-aware runtime needs to know about associations of data and tasks in order to simultaneously enable scheduling tasks and placing data. This association can be explicitly specified by the user (for example in OCR), discovered by an automated tool (for example, with the RStream compiler [37]), extracted from a more high-level specification from the user (Legion [38], HTA [7], RAJA [39], OpenMP, etc.), or from the application metadata as in Perilla [40].

Another big challenge is how to communicate the hierarchical data properties of an application to the runtime so that they can be exploited to generate efficient schedules. Classical random work stealers (e.g., Cilk) do not exploit this. Socket-aware policies exist (e.g., Qthread [41]) that perform hierarchical work stealing: First among cores in a socket and then among sockets. Some programming models expose an API that allows programmers to specify on which NUMA node/socket a collection of tasks should be executed (e.g., OmpSs [32]). Configurable work stealers that can be customized with *scheduling hints* have also been developed [42]. A more extreme option is to allow the application programmer to attach a custom work-stealing function to the application [43].

6 System-Scale Data Locality Management

The highest level in the stack is the whole system, which usually comprises a complex topology ranging from on-chip networks to datacenter-wide interconnection topologies. Optimizing for locality during program execution at this level is equally important to all other levels.

6.1 Key Points

System-scale locality management consists of optimizing application execution, taking into account both the data access of the application and the topology of the machine to reduce node-level data movement. Therefore, in order to enable such optimization two kinds of models are required: An application model and an architecture model. At system scale one must describe the whole ecosystem. Among the relevant elements of the ecosystem are: The cache hierarchy; the memory system and its different operating modes such as slow and large versus fast and small, or persistent versus volatile; the operating system; the network with concerns such as protocol, topology, addressing, and performance; the storage with its connected devices and strategy; and the batch scheduler which has the knowledge of available resources, and other applications running that may interfere with execution.

Applications need abstractions allowing them to express their behavior and requirements in terms of data access, locality and communication at runtime. For these, we need to define metrics to capture the notions of data access, affinity, and network traffic. Defining metrics to describe the application behavior in a concise and precise manner is still a research topic. Often, an affinity graph describing how the different parts of the application interact is useful in managing locality. However, such affinity can be positive (components need to be mapped close together due to shared data) or negative (components need to be spread across the system because of potential contention when accessing shared resources: Memory, storage, network, etc.). A good model of affinity is not yet available in the literature.

A hardware model is needed to control locality. Modeling future large-scale parallel machines will have to describe the memory system better, provide an integrated view with the nodes and the network. The models will also need to exhibit qualitative knowledge, and provide ways to express the multiscale properties of the machine.

6.1.1 Trends and Requirements

We can see different trends affecting the way locality is managed at system scale.

Concerning node and topology modeling, we note that even if most NUMA systems are mostly hierarchical, this is no longer true when we consider the network. Moreover, manycore architecture such as the Intel Knights Landing do not feature a strict hierarchical memory. This means that process placement algorithms need to be able to address arbitrary topologies.

Moreover, even large networks often have low diameter (e.g., diameter-3 Dragonfly [44] or diameter-2 Slim Fly [45] topologies). Therefore, topology mapping could become less important in some cases as the placement may have a smaller impact on the performance or simple random strategies provide close-to-highest performance. Yet, this is not generally true for low-diameter topologies [46]. Precise models of application behavior and the underlying platform are needed in order to understand how placement and data layout impact performance.

In the case of very large machines such as top-end supercomputers featuring millions of cores, the algorithmic cost of process placement becomes very high. Being able to design hierarchical algorithms is required in that setting.

Another important consideration is the ability to deal with dynamic behavior. Topology-aware dynamic load balancing is a hot topic [47], [48], which concerns itself with managing change in application behavior and coping with affinity dependence in the input dataset. This requires modification of the affinity modeling from a static model (e.g., the same communication matrix for the whole application execution) to a dynamic model (e.g., instantiating the communication matrix at runtime).

At system scale it is important to manage affinity for the whole application ecosystem. Currently, locality is managed independently for the volatile memory, the NVRAM, the storage, the network, etc. It is crucial to account for these different resources at the same time to perform global locality optimizations. For instance, optimizing storage access and memory access simultaneously results in good performance gain as shown in early results [49].

Additionally, research into the layer above the parallel file system is beginning to uncover methods of orchestrating I/O *between* applications [50]. This type of high-level coordination can assist in managing shared resources such as network links and I/O gateways and is complementary to an understanding of the storage data layout itself. It can also enable optimization of locality management for several applications at the same time.

6.2 State of the Art

No strict standard way exists to describe and enforce process and thread mapping. For example, techniques for thread binding depend on the underlying operating system, the runtime system (MPI, PGAS, etc.), and even the implementation (e.g., OpenMPI versus MPICH).

Arguably some progress has been made, for example, MPI-3 provides an interface that allows one to detect which processes are in a shared-memory domain (i.e., on the same node) [51]. Other interfaces, for example, thread binding at startup, are not standardized, but MPI allows them to be implemented at the `mpiexec` level.

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 [52] for distributed memory or performance counter analysis for shared-memory systems. It can also be done by tracing data exchange at the runtime level with a system such as OVIS [53], [54], by monitoring the messages transferred between MPI processes, for instance.

Hardware locality (hwloc) [55], [56] is a library and a set of tools for discovering and exposing the hardware topology of machines, including processors, cores, threads, shared caches, NUMA memory nodes, and I/O devices. Netloc [57], [58] is a network model extension of hwloc to account for locality requirements of the network, including the fabric topology. For instance, the network bandwidth and the way contention is managed may change the way the distance within the network is expressed or measured.

The problem is even more important if we consider the way applications are allocated to resources and how they access storage. This requires optimizations between applications. Currently, resource managers or job schedulers such as SLURM [59], OAR [60], LSF [61], or PBS [62] allocate nodes to processes. However, none of them can match the application requirements in terms of communication with the topology of the machine and the constraints incurred by already mapped applications. Similarly, parallel file systems such as Lustre [63], GPFS [64], PVFS [65], and PanFS [66] and I/O libraries such as ROMIO [67], HDF5 [68], and Parallel netCDF [69] are responsible for organizing data on external storage (e.g., disks) and moving data between application memory and external storage over system networks.

7 Applications Expectations from Abstractions

An application developer is concerned with end-to-end parallelization and may be faced with different parallelization needs in different parts of the application [70]. Data locality for applications is often a direct map from their modeling and discretization methods. We can loosely map the applications along two dimensions: Spatial connectivity and functional connectivity. In this map the lower end of the spatial connectivity axis would have applications that are embarrassingly parallel and the top end would have dynamic connectivity such as adaptive meshing. The functional connectivity axis would have single physics applications at the lower end, whereas at the high end would be applications where the components are swapped in and out of active state. Being placed higher along an axis implies greater challenges in achieving locality. HPC applications typically fall into the fourth quadrant, both spatial and functional connectivities are high [5].

Applications communities have well known and valid concerns about wisely utilizing the developers time and protecting the investment already made in the mature production codes of today [71], [72]. An 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. Even with those constraints, however, many possibilities exist in application infrastructure design to expose the potential for data locality, and therefore performance, if appropriate abstractions can be made available. A stable programming paradigm with a lifecycle that is several times the development cycle of the code must emerge for sustainable science. It can take any of the forms under consideration, such as embedded domain-specific languages, abstraction libraries, or full languages, or some combination of these, as long as long term support and commitment are provided, as well as a way to make incremental transition to the new paradigm.

7.1 Overview of Concerns

Abstractions often apply easily to simple problems; but where the computation deviates from the simple pattern, the effectiveness of the abstraction decreases. A useful abstraction would allow itself also to be ignored or turned off as needed. In the context of data locality that might mean an ability to express the inherent hierarchical parallelism in the application in a declarative instead of imperative way, leaving the code translators (compilers or autotuners) to carry out the actual mapping.

Other less considered but possibly equally critical concerns relate to expressibility. Application developers can have a clear notion of their data model without finding ways of expressing the models effectively in the available data structures and language constructs. There is no theoretical basis for the analysis of data movement within the local memory or remote memory. Because of this lack of formalism to inform 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 focuses on correctness and stability first, and then performance. By the time performance analysis tools are applied, it can be 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 to let the design be informed by the expected performance model throughout the process. Such a modeling tool would need to be highly configurable, so that its conclusions might be portable across a range of compilers and hardware and valid into the future, in much the same way that numerical simulations often use ensembles of input-parameter space in order to obtain conclusions with reduced bias. Below we discuss application developers' concerns that tie into the data locality abstractions discussed in earlier sections.

7.2 Data Structures

Data layout and movement have a direct impact on the implementation complexity and performance of an application. Since these are determined by the data structures used in the implementation, this is an important concern for the application. Any effort that moves in the direction of allowing the application to describe the working set through a library or an abstraction is likely to prove useful.

Most languages provide standard containers and data structures that are easy to use in high-level code; yet few languages or libraries provide interfaces for the application developer to inform the compiler about expectations of data locality, data layout, or memory alignment. For example, a common concern for the PDE solvers is the data structure containing multiple field components that have identical spatial layout. Should it be an array with an added dimension for the field components or a structure; and within the array or structure, what should be the order for storing in memory for performance [73], [74]. There is no *one* best layout for every platform. State of the art abstractions and tools described in Section 3 are working towards making that a programming abstraction concern instead of an application concern. Other abstractions that could be helpful for performance include allowing persistence of data between two successive code modules.

7.3 Languages and Compilers

The state of the art in parallel programming models currently used in applications is a hybrid model such as MPI+OpenMP or MPI+CUDA/OpenCL. The former is both local-by-default (MPI) and global-by-default (OpenMP), while the latter is local-by-default only (object visibility defined in Section 4). Since the two models target different classes of platforms, they do not really overlap. PGAS models have much less penetration in the field than do the above two models. In general a global-by-default model is easier to adopt but it is much harder to make it performant. The real difficulty in designing for parallelism lies in finding the best hierarchical decomposition inherent to the application. That is basically the hierarchical version of the local-by-default approach. Abstractions such as tiling can be helpful in expressing hierarchical parallelism. Because of being explicitly aware of locality, local-by-default design can be more easily mapped to a performant global design.

The transition to a new programming language, although likely to be optimal eventually, is not a realistic solution in the near term. In addition to the usual challenge of sustainability (it might go away), need for verification dictates incremental adoption for existing codes. Therefore, either embedded DSLs or new languages with strong interoperability with the existing languages are likely to have better chance at being adopted. The amount of effort required by the applications to transition to the new programming model will be another factor in its success. Irrespective of which solution emerges, it must provide a robust and clean way of handling threads for interoperability among various components of the application. Also, just-in-time compilation will be helpful to many applications with highly variable runtime characteristics.

7.4 Runtime

The vast majority of applications in computational science and engineering continue to operate in largely bulk-synchronous mode, with a few notable exceptions such as Uintah [75], and applications built upon Charm++ [76] such as NAMD [77]. As the applications see it, this approach has two major benefits: many applications have a built in regularity, and therefore map well to the bulk-synchronous mode, and it takes care of dependencies within the application trivially. Evidence indicates, however, that this state of affairs may not remain attractive or even feasible because heterogeneity in hardware is unfavorable to regulate lock-step execution. Additionally, capability additions in applications make them more heterogeneous. However, the jury is still out on whether the overheads of asynchronicity will be outweighed by the benefits of pipelining and overlapping permitted by the task-based runtime. A good API that allows articulating the hierarchical decomposition and dependencies easily is likely to be helpful to applications to think about runtime locality, and to reason about their code functionally without implementing it in a functional language. Such an approach is needed to their way away from bulk synchronism.

7.5 System-Scale

System-wide scalability is an important cross-cutting issue since the targets are very large-scale, high-performance computers. On the one hand, application scalability will depend mostly on the way data is accessed and locality is managed. On the other hand, the proposed solutions and mechanisms have to run at the same scale as the application which limits their inner decision time. That, in turn, makes it 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). Novel approaches are needed to control data locality system wide, by integrating cross-layer I/O stack mechanisms with cross-node topology-aware mechanisms. Another challenge is that often each layer of the software stack is optimized independently to address the locality concerns with the result that outcomes sometime conflict. It is therefore important to observe the interaction of different approaches and propose integrated solutions that provide a global optimization across different layers. An example of such an approach is mapping independent application data accesses to a set of storage resources in a balanced manner. This approach requires an ability to interrogate the system regarding what resources are available, some distance metric in terms of application processes, and coordination across those processes (perhaps supported by a system service) to perform an appropriate mapping. Ultimately, the validation of the models and solutions to the concerns and challenges will be a key challenge.

8 Summary

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

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

Acknowledgments

Authors would like to thank other PADAL14 and PADAL15 workshop participants: Maciej Besta, Jed Brown, Cy Chan, Sung-Eun Choi, Jack Choquette, Brice Goglin, Jesus Labarta, Leonidas Linardakis, Edward Luke, Satoshi Matsuoka, Peter Messmer, Lawrence Mitchell, Kathryn O'Brien, David Padua, Robert B. Ross, Marie-Christine Sawley, Robert Schreiber, Thomas Schulthess. James Sexton, Suzanne Michelle Shontz, Adrian Tate, Gysi Tobias, Engo Toshio, Mohamed Wahib, Chih-Chieh Yang. This work was partially supported by the German Research Foundation under contract TE 163/17-1. This work was partially supported by the Grant 655965 by the European Commission.

References

[1] P. M. Kogge and J. Shalf, “Exascale computing trends: Adjusting to the new normal' for computer architecture,” *Comput. Sci. Eng.*, vol. 15, no. 6, pp. 16–26, 2013.

[2] J. A. Ang, et al., “Abstract machine models and proxy architectures for exascale computing,” in *Proc. 1st Int. Workshop Hardware-Software Co-Des. High Perform. Comput.*, 2014, pp. 25–32.

[3] P. Participants, “Workshop on programming abstractions for data locality, PADAL ’14,” 2014. [Online]. Available: https://sites.google.com/a/lbl.gov/padal-workshop/

[4] P. Participants, “Workshop on programming abstractions for data locality, PADAL ’15,” 2015. [Online]. Available: https://sites.google.com/a/lbl.gov/padal-workshop/

[5] D. Unat, et al., “Programming Abstractions for Data Locality,” Joint report of LBNL/CSCS/ETH/ANL/INRIA, Tech. Rep., Lugano, Switzerland, 2014.

[6] T. Fuchs and K. Fuerlinger, “Expressing and exploiting multidimensional locality in DASH,” in *Proc. SPPEXA Symp.*, 2016, pp. 341–359.

[7] G. Bikshandi, et al., “Programming for parallelism and locality with hierarchically tiled arrays,” in *Proc. 11th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2006, pp. 48–57.

[8] T. L. Veldhuizen and D. Gannon, “Active libraries: Rethinking the roles of compilers and libraries,” *CoRR*, vol. math.NA/9810022, 1998.

[9] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, 2014.

[10] D. Unat, et al., in *Proc. TiDA: High-Level Program. Abstractions Data Locality Manage.*, 2016, pp. 116–135.

[11] M. Bianco and B. Cumming, “A generic sfor multi-stage stencils,” *Proc. Euro-Par Parallel Process. 20th Int. Conf. Porto Portugal* 2014, pp. 584–595.

[12] Intel Open Source, “Hetero Streams Library,” (2017). [Online]. Available: https://01.org/hetero-streams-library

[13] K. Fuerlinger, et al., “DASH: Data structures and algorithms with support for hierarchical locality,” in *Proc. Euro-Par Workshops*, 2014, pp. 542–552.

[14] W. Zhang, A. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, “Boxlib with tiling: An adaptive mesh refinement software framework,” *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. S156–S172, 2016.

[15] T. Gysi, T. Grosser, and T. Hoefler, “Modesto: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures,” in *Proc. 29th ACM Int. Conf. Supercomput.*, 2015, pp. 177–186.

[16] Loci, “Language,” (2016). [Online]. Available: https://sourceforge.net/projects/loci-framework/

[17] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin, “A new vision for coarray fortran,” in *Proc. 3rd Conf. Partitioned Global Address Space Program. Models*, 2009, pp. 5:1–5:9.

[18] B. L. Chamberlain, D. Callahan, and H. P. Zima, “Parallel programmability and the chapel language,” *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, 2007.

[19] Julia, “Language,” (2017). [Online]. Available: http://julialang.org/

[20] R. W. Numrich and J. Reid, “Co-array fortran for parallel programming,” *ACM SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[21] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, *X10 Language Specification Version 2.4*, San Jose, CA, USA: IBM Research, May 2014.

[22] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, “Hierarchical place trees: A portable abstraction for task parallelism and data movement,” in *Proc. 22nd Int. Workshop Languages Compilers Parallel Comput.*, 2009, pp. 172–187.

[23] K. Yelick, et al., “Titanium: A high-performance Java dialect,” in *Proc. Workshop Java High-Perform. Netw. Comput.*, 1998, pp. 825–836.

[24] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and W. D. Weathersby, “ZPL: A machine independent programming language for parallel computers,” *IEEE Trans. Softw. Eng.*, vol. 26, no. 3, pp. 197–211, Mar. 2000.

[25] A. Kamil and K. Yelick, “Hierarchical computation in the SPMD programming model,” in *The 26th International Workshop on Languages and Compilers for Parallel Computing*, Sept. 2013.

[26] M. Garland, M. Kudlur, and Y. Zheng, “Designing a unified programming model for heterogeneous machines,” in *Proc. Int. Conf. High Perform. Comput. Netw. Starage Anal.*, 2012, pp. 1–11.

[27] Y. Zheng, A. Kamil, M. Driscoll, H. Shan, and K. Yelick, “UPC++: A PGAS extension for C++,” in *Proc. 28th IEEE Int. Parallel Distrib. Process. Symp.*, 2014, pp. 1105–1114.

[28] R. M. Karp, R. E. Miller, and S. Winograd, “The organization of computations for uniform recurrence equations,” *J. ACM*, vol. 14, no. 3, pp. 563–590, Jul. 1967.

[29] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” in *Proc. 29th ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2008, pp. 101–113.

[30] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, “Polyhedral parallel code generation for CUDA,” *ACM Trans. Archit. Code Optimization*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[31] T. Grosser and T. Hoefler, “Polly-ACC: Transparent compilation to heterogeneous hardware,” in *Proc. 30th Int. Conf. Supercomput.*, 2016, Art. no. 1.

[32] Barcelona Supercomputing Center, “The OmpSs Programming Model,” (2017). [Online]. Available: https://pm.bsc.es/ompss

[33] Barcelona Supercomputing Center, “Extrae: A Paraver trace-files generator,” (2017). [Online]. Available: https://tools.bsc.es/extrae.

[34] Barcelona Supercomputing Center, “Paraver: A flexible performance analysis tool,” (2017). [Online]. Available: https://tools.bsc.es/paraver.

[35] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-oblivious algorithms,” in *Proc. 40th Annu. Symp. Found. Comput. Sci.*, 1999, pp. 285–297.

[36] Intel Open Source, “Open Community Runtime,” (2017). [Online]. Available: https://01.org/open-community-runtime

[37] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, “R-stream compiler,” in *Proc. Encyclopedia Parallel Comput.*, 2011, pp. 1756–1765.

[38] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 66:1–66:11.

[39] R. Hornung and J. Keasler, “The raja portability layer: Overview and status,” Tech. Rep. LLNL-TR-661403, 2014.

[40] T. Nguyen, D. Unat, W. Zhang, A. Almgren, N. Farooqi, and J. Shalf, “Perilla: Metadata-based optimizations of an asynchronous runtime for adaptive mesh refinement,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 81:1–81:12.

[41] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, “OpenMP task scheduling strategies for multicore NUMA systems,” *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 2, pp. 110–124, May 2012.

[42] M. Wimmer, D. Cederman, J. L. Träff, and P. Tsigas, “Work-stealing with configurable scheduling strategies,” in *Proc. 18th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2013, pp. 315–316.

[43] J. Nakashima, S. Nakatani, and K. Taura, “Design and implementation of a customizable work stealing scheduler,” in *Proc. Int. Workshop Runtime Oper. Syst. Supercomput.*, 2013, Art. no. 9.

[44] J. Kim, W. J. Dally, S. Scott, and D. Abts, “Technology-driven, highly-scalable dragonfly topology,” *SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 77–88, Jun. 2008.

[45] M. Besta and T. Hoefler, “Slim fly: A cost effective low-diameter network topology,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 348–359.

[46] B. Prisacari, G. Rodriguez, P. Heidelberger, D. Chen, C. Minkenberg, and T. Hoefler, “Efficient task placement and routing of nearest neighbor exchanges in dragonfly networks,” in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 129–140.

[47] L. L. Pilla, et al., “Asymptotically optimal load balancing for hierarchical multi-core systems,” in *Proc. IEEE 18th Int. Conf. Parallel Distrib. Syst.*, 2012, pp. 236–243.

[48] E. Jeannot, E. Meneses, G. Mercier, F. Tessier, and G. Zheng, “Communication and topology-aware load balancing in Charm++ with TreeMatch,” *2013 IEEE Int. Conf. Cluster Comput.*, pp. 1–8, Sep. 2013, doi: 10.1109/CLUSTER.2013.6702666.

[49] F. Tessier, P. Malakar, V. Vishwanath, E. Jeannot, and F. Isaila, “Topology-aware data aggregation for intensive I/O on large-scale supercomputers,” in *Proc. 1st Workshop Optimization Commun. HPC Runtime Syst. Held Conjunction ACM/IEEE SuperComput. Conf.*, 2016, Art. no. 10.

[50] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim, “CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination,” in *Proc. Int. Parallel Distrib. Process. Symp.*, 2014, pp. 155–164.

[51] T. Hoefler, et al., “MPI + MPI: A new hybrid approach to parallel programming with MPI plus shared memory,” *J. Comput.*, vol. 93, no. 12, pp. 1121–1136, 2013.

[52] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The Scalasca performance toolset architecture," *Concurrency Comput. Practice Exp.*, vol. 22, no. 6, pp. 702–719, Apr. 2010.

[53] Ovis, "Main Page - OVISWiki," (2017). [Online]. Available: https://ovis.ca.sandia.gov

[54] M. Showerman, et al., "Large scale system monitoring and analysis on blue waters using OVIS," in *Proc. Cray User's Group*, 2014, pp. 1–12.

[55] B. Goglin, "Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc)," in *Proc. Int. Conf. High Perform. Comput. Simulation*, 2014, pp. 74–81.

[56] Hwloc, "Portable Hardware Locality," (2010). [Online]. Available: http://www.open-mpi.org/projects/hwloc/

[57] B. Goglin, J. Hursey, and J. M. Squyres, "Netloc: Towards a Comprehensive View of the HPC System Topology," in *Proc. 5th Int. Workshop Parallel Softw. Tools Tool Infrastructures Conjunction*, 2014, pp. 216–225.

[58] Netloc, "Portable Network Locality," http://www.open-mpi.org/projects/netloc/

[59] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Proc. Job Scheduling Strategies Parallel Process.*, 2003, pp. 44–60.

[60] N. Capit, et al., "A batch scheduler with high level components," in *Proc. IEEE Int. Symp. Cluster Comput. Grid*, 2005, pp. 776–783.

[61] S. Zhou, "Lsf: Load sharing in large heterogeneous distributed systems," in *Workshop on Cluster Computing*, 1992.

[62] R. L. Henderson, "Job scheduling under the portable batch system," in *Proc. Job Scheduling Strategies Parallel Process.*, 1995, pp. 279–294.

[63] P. J. Braam, "The lustre storage architecture," Cluster File Systems, Inc., Tech. Rep., 2003. [Online]. Available: http://lustre.org/docs/lustre.pdf

[64] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002.

[65] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Proc. 4th Annu. Linux Showcase Conf.*, 2000, pp. 317–327.

[66] B. Welch, et al., "Scalable performance of the panasas parallel file system," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 17–33.

[67] R. Thakur, W. Gropp, and E. Lusk, "On implementing MPI-IO portably and with high performance," in *Proc. 6th Workshop Input/Output Parallel Distrib. Syst.*, 1999, pp. 23–32.

[68] "HDF5," (2017). [Online]. Available: http://www.hdfgroup.org/HDF5/.

[69] J. Li, et al., "Parallel netCDF: A high-performance scientific I/O interface," in *Proc. ACM/IEEE Conf. Supercomput.*, 2003, pp. 39–39.

[70] J. Segal and C. Morris, "Developing scientific software," *IEEE Software*, vol. 25, no. 4, pp. 18–20, Jul. 2008.

[71] L. Hochstein and V. R. Basili, "The ASC-alliance projects: A case study of large-scale parallel scientific code development," *Computer*, vol. 41, no. 3, pp. 50–58, 2008.

[72] J. C. Carver, "Software engineering for computational science and engineering," *Comput. Sci. Eng.*, vol. 14, no. 2, pp. 8–11, 2012.

[73] A. Dubey, et al., "A survey of high level frameworks in block-structured adaptive mesh refinement packages," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3217–3227, 2014.

[74] A. Dubey, et al., "Software abstractions and methodologies for HPC simulation codes on future architectures," *J. Open Res. Softw.*, vol. 2, no. 1, 2014.

[75] S. G. Parker, "A component-based architecture for parallel multi-physics PDE simulation," *Future Generation Comput. Syst.*, vol. 22, pp. 204–216, 2006.

[76] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *Proc. 8th Annu. Conf. Object-Oriented Program. Syst. Languages Appl.*, 1993, pp. 91–108.

[77] J. C. Phillips, et al., "Scalable molecular dynamics with NAMD," *J. Comput. Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

Didem Unat is an assistant professor of computer science and Engineering at Koç University, Istanbul, Turkey. Previously she was at the Lawrence Berkeley National Laboratory. She is the recipient of the Luis Alvarez Fellowship in 2012 at the Berkeley Lab. Her research interest lies primarily in the area of high performance computing, parallel programming models, compiler analysis and performance modeling. Visit her group webpage for more information parcorelab.ku.edu.tr.

Anshu Dubey received the PhD degree in computer science from Old Dominion University, in 1993 and the BTech degree in electrical engineering from Indian Institute of Technology, New Delhi. She is a computer scientist in the Mathematics and Computer Science Division at Argonne National Laboratory, and a senior fellow at the Computation Institute. From 2013 to 2015 she was at Lawrence Berkeley National Laboratory, where she served as work lead and computer systems engineer. Prior to that she was associate director and computer science/applications group leader in Flash Center For Computation Science at the University of Chicago.

Torsten Hoefler is an assistant professor of Computer Science at ETH Zürich, Switzerland. He is active in the Message Passing Interface (MPI) Forum where he chairs the "Collective Operations and Topologies" working group. His research interests revolve around the central topic of "Performance-centric Software Development" and include scalable networks, parallel programming techniques, and performance modeling. Additional information about Torsten can be found on his homepage at htor.inf.ethz.ch.

John Shalf received the MS degree in electrical and computer engineering from and Virginia Tech. He is CTO of the National Energy Research Supercomputing Center and head of the Computer Science Department, Lawrence Berkeley National Laboratory. His research interests include parallel computing software and high-performance computing technology. He is a member of the American Association for the Advancement of Science and the Optical Society of America, and coauthor of the whitepaper The Landscape of Parallel Computing Research: A View from Berkeley (UC Berkeley, 2006). Contact him at jshalf@lbl.gov. He is a member of the IEEE.

Mark Abraham is a research scientist at the KTH Royal Technical University in Stockholm, Sweden. He manages the worldwide development of the high-performance molecular simulation package GROMACS.

Mauro Bianco is currently a computer scientist at Swiss National Supercomputing Centre. His focus is on the design and development of domain specific C++ libraries for parallel and portable scientific simulations.

Bradford L. Chamberlain received the PhD from the Department of Computer Science and Engineering, the University of Washington and remains associated with the department as an affiliate professor. He is a principal engineer at Cray Inc. where he serves as the technical lead for the Chapel parallel programming language project.

Romain Cledat graduated in 2011 from the Georgia Institute of Technology with the PhD degree in computer science. He also received the MS degree in electrical and computer engineering from Georgia Tech and a Masters in Engineering from the Ecole Centrale de Lyon (France). He is currently a leading developer on the Open Community Runtime (OCR) as part of DoE's XStack project which aims to develop the software infrastructure for Exascale computing.

H. Carter Edwards is a principal member of Technical Staff at Sandia National Laboratories in Albuquerque, New Mexico, where he leads research and development for performance portable programming models on next generation manycore architectures. He lead the Kokkos (github.com/kokkos) project and represents Sandia on the ISO/C++ standards committee.

Hal Finkel is a computational scientist at Argonnes Leadership Computing Facility. He works on the LLVM compiler infrastructure, the Hardware/Hybrid Cosmology Code (HACC), and represents Argonne on the C++ standards committee.

Karl Fuerlinger is a lecturer and senior researcher at the Ludwig-Maximilians-University (LMU) Munich, working in the area of parallel and high performance computing. His research is focused on tools for program analysis and parallel programming models.

Frank Hannig (M'01–SM'12) leads the Architecture and Compiler Design Group in the CS Department at Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany. His main research interests are the design of massively parallel architectures, ranging from dedicated hardware to multi-core architectures, domain-specific computing, and architecture/compiler co-design. He is a senior member of the IEEE.

Emmanuel Jeannot is a senior research scientist at Inria, in Bordeaux, France. He leads the Tadaam (Topology-Aware System-Scale Data Management for High-Performance Computing) project team and works on runtime system and process placement.

Amir Kamil received the PhD degree at UC Berkeley, where his research focused on program analysis and optimization for parallel languages and programming models for hierarchical machines. He is a Lecturer at the University of Michigan and a Computer Systems Engineer at Lawrence Berkeley National Laboratory (LBNL). He has continued his research on programming models at LBNL, while also investigating communication optimizations for dense grid applications and contributing to UPC++, a C++ library for distributed computation.

Jeff Keasler worked for over a decade as a member of the ALE3D application team at LLNL. He is co-PI with Rich Hornung for both the LULESH proxy application and the RAJA portability layer.

Paul H. J. Kelly is Professor of Software Technology at Imperial College London, where he has been on the faculty since 1989. He leads Imperial's Software Performance Optimization research group, and is co-director of Imperial's Centre for Computational Methods in Science and Engineering. His research focus is domain-specific program optimization.

Vitus Leung is a principal member of Technical Staff at Sandia National Laboratories in Albuquerque, New Mexico, where he leads research in distributed memory resource management. He has won R&D 100, US Patent, and Federal-Laboratory-Consortium Excellence-in-Technology-Transfer awards for work in this area. He is a senior member of the ACM and has been a Member of Technical Staff at Bell Laboratories in Holmdel, New Jersey and a Regents Dissertation Fellow at the University of California.

Hatem Ltaief is a senior research scientist in the Extreme Computing Research Center at KAUST. His research interests include parallel numerical algorithms, fault tolerant algorithms, parallel programming models, and performance optimizations for multicore architectures and hardware accelerators. His current research collaborators include Aramco, Total, Observatoire de Paris, NVIDIA, and Intel.

Naoya Maruyama is a team leader at RIKEN Advanced Institute for Computational Science, where he leads the HPC Programming Framework Research Team. His team focuses on high-level parallel frameworks for computational science applications.

Chris J Newborn received the PhD degree at Carnegie Mellon University. He serves as an HPC architect, focused on Intel's Xeon Phi product family. He has contributed to a combination of hardware and software technologies over the last twenty years. He has a passion for making the richness of heterogeneous platforms easier to use. He has more than 80 patents. He wrote a binary-optimizing, multi-grained parallelizing compiler as part. He's delighted to have worked on volume products that his Mom uses.

Miquel Pericás is an assistant professor of computer science and engineering at Chalmers University of Technology in Gothenburg, Sweden. From 2012 to 2014 he was a JSPS postdoctoral fellow at the Tokyo Institute of Technology, and before that a senior researcher at the Barcelona Supercomputing Center. His research focuses on parallel runtime systems and communication-efficient computer architectures.