Programming Abstractions for Data Locality

author list goes here

June 17, 2014

Programming Abstractions for Data Locality

author list goes here

abstract goes here

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

Motivating Applications and Their Requirements

SKELETON In this section we discuss data locality from the applications perspective, covering a range of modeling methods. While the applications space itself is very large, the set of applications/applications experts included in the workshop were fairly representative of the numerical algorithms and techniques used in majority of state-of-the-art application codes (i.e. [**cosmo**], [**gromacs**], [**Hydra**, **op2**], [**chombo**], [**vis**]. Additionally, among the included applications were those that model multi-physics phenomena combining several different numerical and algorithmic technologies into one tightly coupled whole. These applications demonstrate the challenges of characterizing the behavior of individual solvers when embedded in a larger code base with multiple and heterogeneous solvers. Because of the presence of many tightly coupled solvers these applications also highlight the importance of interoperability among solvers and libraries by implication. 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, protien 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.

The highly used algorithms in scientific computing have varying degrees of arithmetic intensity and potential for data locality inherent to them. For example, Gromacs has a small working set size for (? short-range) .... therefore benefits from easily obtained data locality. Where modeling needs to consider global effects, the required resolution is low enough that a judicious decomposition and mapping of the work load can confine these spatially long range computations to a small subset of processes, thereby retaining reasonable data locality. By contrast typical low order partial differential equation solvers with structured mesh or particles typically have very few operations per data item and therefore struggle with achieving a high degree of data reuse. Unstructured meshes have an additional layer of indirection that exacerbates this problem. Others fall somewhere in between. Multiphysics applications further add a dimension in the data locality challenge; that of the different data access patterns in different solvers. Here, increasing locality for one solver can decrease it for another, thereby necessitating carefully considered trade-offs. There are well known and valid concern among 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.

2.1 The State-of-the-Art

Among the domain science communities relying on modeling and simulation to obtain results there is huge variance in awareness and preparedness for the ongoing hardware platform architecture revolution. This huge variance also exists in the need that different research groups have to be prepared. A great deal of useful science is still done with very limited scope prototype level software in use by a small research group. 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 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 in the compilers, programming abstractions and runtime systems in order to be better prepared for the platform heterogeniety. At the workshop examples of many such efforts were presented. For example (HIPA^{cc} [18]), OP2 [], and Stella [] represented the use of embedded domain specific languages being used to good effect in their target science domains. Other efforts such as use of tiling within patches in AMR for exposing greater parallelism rely on a library based approach. Still others such as Gromacs using SIMD operations for non-bonded iterations or doing ensemble simulations are targetted at utilizing vectorization possibilities presented by the accelerators. 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.

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. 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 solution,s have the added advange of providing an incremental path for transition. The trade-off is the possibly 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 the incrementally 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.

As mentioned earlier, for the production grade applications the biggest non-negotiable 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. There exist research groups even today that refuse to use parallel IO libraries, which have been robust and early to get on new platforms for a few years now. They still prefer to roll their own IO solution with all the attendant performance and maintainance 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 easy to install and link library. With this model the code can always run in the high level language, while better translation tools can provide performance as they become available.

2.2 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: (1) what methodology should be used to determine what constitutes a component, (2) what degree of locality awareness is appropriate in a component, (3) what is the optimum level abstraction in the component based design, i.e. who is aware of spatial decomposition, and who is aware of functional decomposition it it exists, (4) how to design so that numerical complexity does not interleave with the complexity arising out of locality management, and (4) 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 perspectives 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 value proposition for the applications.

Chapter 3

Data Structures and Layout Abstractions

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
  - 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)
  - 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)

- 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 computation. Advanced architectures have heterogeneous memory that, or performance, must be managed by computations. Examples of these memory spaces include CPUs' non-uniform memory access (NUMA) regions, separate GPU memory, and GPU's on-chip shared memory.
- Data structure: A organized collection of datum 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.
- 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?
- Locality: A measure of distance between datum on a computer. For example, do the datum 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?]
- Memory access pattern: A computation accesses data structures according to its execution policy.
- Execution policy: How a computation's units of work are scheduled and mapped onto data collections [in line with C++ committee].
- 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 datum.
- 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:
      - 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
  - 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 datum 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 datum 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 4

Language and Compiler Support for Data Locality

HIGHLIGHTS

- 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, octtrees, structured adaptive-mesh multigrids etc. While Chapter 3 tackles particular data stuctures, in this chapter we look at how to build tools to support programmers who build such abstractions.
- 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.
- TERMINOLOGY (define each of these up front)

  - *Active library:* a library which comes with a mechanism for delivering library-specific optimizations [23]. Active library technologies differ in how this is achieved - examples include template metaprogramming in C++, Lightweight Modular Staging in Scala [21], source-to-source transformation (for example using tools like ROSE [24]), and run-time code generation, driven either by delayed evaluation of library calls [22], 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 [12]. 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 [6].
  - *Directive-based language extensions:*

- 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)

- AGREEMENTS
  - PRINCIPLES
    - Avoid losing information through premature "lowering" of the program representation.
    - 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
  - Do what I say vs do what I mean [[PK: I thnk this is *agreement* on the need for "multiresolution" solutions, see earlier]]
  - Compositional metadata vs explicit dependence/synchronisation/movement (Brad has words)

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

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

- RESEARCH AGENDA
  - Scope out the opportunities for JIT - plenty of compelling need for infrastructure (specialisation, autotuning,) [[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 5

Data Locality in Task Models

HIGHLIGHTS

- 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
- RELATED WORK: a few examples of work in that area (**Hatem**)
  - OmpSs
  - Charm++
  - QUARK
  - StarPU
  - PaRSEC
  - SuperMatrix
  - ParalleX
  - OCR
- DISCOVERIES/CHALLENGES
  - Task granularity (flexibility) (**Miquel**)
  - Scheduling overhead (**Miquel**)
  - Efficient debugging/tracing mechanisms/tools (**Jesus**)
  - Nested parallelism (recursive formulation) (**Hatem**)
  - Work stealing (**Miquel**)
  - 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**)
- 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 targetted 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.
- 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.
- APPLICATION DOMAINS (**Hatem**)
  - Dense linear algebra
  - Computational astronomy

- Uncertainty Quantification
- Seismic
- Weather/Climate modeling
- Krylov-based solvers
- Nbody problems

Chapter 6

System-Scale Data Locality Management

HIGHLIGHTS

- SCOPE
  - topology mapping
  - storage and data
  - Resource management and batch scheduler
- KEY POINTS
  - However the application is written and optimize, the way it is executed has a huge impact on its performance
  - Mapping of application to resources is important even in the case of contiguous allocation. But joint allocation and mapping (application expresses its need and the batch scheduler tries to fulfill it) provides better optimization
  - there is a need for models of the machine at different scales (cache, node, memory, network, etc.) Such models are necessary to design algorithms to enhance performance
  - Usually, each part of the application ecosystem (storage, runtime, batch scheduler, etc.) act independently. There is a need for joint decisions both vertically (from node to storage) and horizontally (e.g. between compute nodes)
  - need model of application (same reason as above)
    - shift in application communication models from BSP to over-decomposition model
    - graph-based analytic applications
    - applications using stencil communication patterns
    - applications using transpose communication patterns
  - ...
- RELATED WORK
  - Topology mapping: TreeMatch, Libtopomap, MPIPP? scotch, metis
  - Geometric mapping: Zoltan2
  - Storage: parallel file system (???)
  - resource management: queuing systems 'SLURM, LSF, PBS, etc.)
  - machine models: HWLOC, NETLOC, etc.
  - building app model: profilers (scalasca), compilers, runtime monitoring (openMPI), miniApps, etc.

- CHALLENGES
  - scalable management of the hierarchy and topology
  - globality (tackle the problem system wide)
  - interaction between different layers
  - alternative node architectures
  - alternative topologies, bottlenecks
  - validation of models
  - ...

6.1 Introduction

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 (flash, non-volatile, RAM, etc), elaborate topologies (both at the shared-memory level and at the distributed-memory one with state-of-the-art interconnection networks) and advanced parallel storage systems and resource management tools.

Nowadays it is already more challenging to write an application that efficiently accesses its data than efficiently processes the data. In other words bytes feature more issues than flops. As it is expected that the memory per core will decrease in the coming years this problem will become even more important.

To address the data locality issue system-wide, one approach consists of working on the application ecosystem and how the application executed (interaction with the batch-scheduler and the storage, optimization on the way the application uses the resources, its relation with the topology, the interaction with other executing application, etc.).

More precisely, the important issues 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, (3) data locality and transfers for applications.

6.2 Key points

In the literature there are tremendous efforts for optimizing an application statically (better data layout, compilation optimization, parallelism structuration, etc.). However, once an application is written and compiled there are several 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, we have:

- 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,
- ...

It is important to remark that these runtime factors are orthogonal to the static optimizations that can be perform. In other words, whatever, the application is written and optimized the way it is executed and it interacts with its ecosystem have a huge impact on its behavior. For instance, it has been shown that the way resources are allocated to application plays a crucial role in the performance of the execution. Recent work [4, 17] 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 [3, 14]. The reason is that the topology of a HPC machine is hierarchical and that the process affinity is not uniform (some pairs of processes exchange more data than some other pairs).

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 a huge parallelism, future applications will 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 procide 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 to 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.

Application models require abstractions on how to express its behavior and its requirement in terms of data access, locality and communication. It is require to define metrics to capture the notions of data access, affinity, network traffic, etc.

To optimize execution at system scale, it is required 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 [10, 11, 20], 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

TODO:

- shift in application communication models from BSP to over-decomposition model
- graph-based analytic applications
- applications using stencil communication patterns
- applications using transpose communication patterns

6.3 Related Work

Concerning topology mapping TreeMatch [15], provides mapping of processes onto computing resources in the case of a tree topology (such as current NUMA nodes and fattree network). LibTopoMap [11] addresses the same problem as TreeMatch but for arbitrary topology such as torus, grid, etc. Zoltan [1, 5] is a tool chain where processors are first allocated to the application and then processes are mapped to the allocated resources depending on the geometry of the target machine and process affinity. Topology mapping is basically a graph embedding problem where a application graph is embedded into a machine graph. Therefore, graph partitioners such as Scotch [7] or ParMetis [16] can address the problem even though they might require more precise information than specific tools and do not always provide good solutions [15].

Hardware Locality (HWLOC) [13] 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 [19] is 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 [8]. It can also be done by tracing data exchange at the runtime level by, for instance monitoring the messages transferred between MPI processes. 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 [25], OAR [2], LSF [26] or PBS [9] 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.

6.4 challenges

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 (storage, resource manager, etc.) 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.

TODO:

- alternative node architectures
- alternative topologies, bottlenecks
- validation of models
- ...

Chapter 7

Conclusion

Bibliography

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

[2] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard. "A batch scheduler with high level components". In: *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on.* Vol. 2. IEEE. 2005, pp. 776–783.

[3] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. "MPIPP: an Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters". In: *Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006.* Ed. by G. K. Egan and Y. Muraoka. ACM, 2006, pp. 353–360. ISBN: 1-59593-282-8.

[4] Y. Cui, E. Poyraz, J. Zhou, S. Callaghan, P. Maechling, T. Jordan, L. Shih, and P. Chen. "Accelerating CyberShake Calculations on the XE6/XK7 Platform of Blue Waters". In: *Extreme Scaling Workshop (XSW), 2013.* IEEE. 2013, pp. 8–17.

[5] M. Deveci, S. Rajamanickam, V. J. Leung, K. Pedretti, S. L. Olivier, D. P. Bunde, U. V. Catalyürek, and K. Devine. "Exploiting geometric partitioning in task mapping for parallel computers". In: *IPDPS.* PHOENIX (Arizona) USA, May 2014.

[6] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. "SugarJ: Library-based Syntactic Language Extensibility". In: *SIGPLAN Not.* 46.10 (Oct. 2011), pp. 391–406. ISSN: 0362-1340. DOI: `10.1145/2076021.2048099`. URL: `http://doi.acm.org/10.1145/2076021.2048099`.

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

[8] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. "The Scalasca performance toolset architecture". In: *Concurrency and Computation: Practice and Experience* 22.6 (Apr. 2010), pp. 702–719. DOI: `10.1002/cpe.1556`.

[9] R. L. Henderson. "Job scheduling under the portable batch system". In: *Job scheduling strategies for parallel processing.* Springer. 1995, pp. 279–294.

[10] T. Hoefler, E. Jeannot, and G. Mercier. "Chapter 5: An Overview of Process Mapping Techniques and Algorithms in High-Performance Computing". In: *High Performance Computing on Complex Environments.* Ed. by E. Jeannot and J. Žilinskas. To be published. Wiley, 2014, pp. 65–84.

[11] T. Hoefler and M. Snir. "Generic Topology Mapping Strategies for Large-Scale Parallel Architectures". In: *Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31 - June 04, 2011.* Ed. by D. K. Lowenthal, B. R. de Supinski, and S. A. McKee. ACM, 2011, pp. 75–84. ISBN: 978-1-4503-0102-2.

[12] P. Hudak. "Building Domain-Specific Embedded Languages". In: *ACM Computing Surveys* 28 (1996).

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

[14] E. Jeannot and G. Mercier. “Near-optimal Placement of MPI Processes on Hierarchical NUMA Architectures”. In: *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference*. Ed. by P. D’Ambra, M. R. Guarracino, and D. Talia. Vol. 6272. Lecture Notes on Computer Science. Ischia, Italy: Springer, Sept. 2010, pp. 199–210.

[15] E. Jeannot, G. Mercier, and F. Tessier. “Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.4 (Apr. 2014), pp. 993–1002. ISSN: 1045-9219. DOI: 10.1109/TPDS.2013.104.

[16] G. Karypis, K. Schloegel, and V. Kumar. “Parmetis”. In: *Parallel graph partitioning and sparse matrix ordering library. Version* 2 (2003).

[17] B. Kramer. “Is Petascale Completely Done? What Should We Do Now?” joint-lab on petsacale computing workshophttps://wiki.ncsa.illinois.edu/display/jointlab/Joint-lab+workshop+Nov.+25-27+2013. Nov. 2013.

[18] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. “Generating Device-specific GPU Code for Local Operators in Medical Imaging”. In: *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Shanghai, China: IEEE, May 21–25, 2012, pp. 569–581. ISBN: 978-0-7695-4675-9. DOI: 10.1109/IPDPS.2012.59.

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

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

[21] T. Rompf and M. Odersky. “Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs”. In: *Commun. ACM* 55.6 (2012), pp. 121–130.

[22] F. P. Russell, M. R. Mellor, P. H. J. Kelly, and O. Beckmann. “DESOLA: An active linear algebra library using delayed evaluation and runtime code generation”. In: *Sci. Comput. Program.* 76.4 (2011), pp. 227–242.

[23] T. L. Veldhuizen and D. Gannon. “Active Libraries: Rethinking the roles of compilers and libraries”. In: *CoRR* math.NA/9810022 (1998).

[24] Q. Yi and D. J. Quinlan. “Applying Loop Optimizations to Object-Oriented Abstractions Through General Classification of Array Semantics”. In: *LCPC*. Ed. by R. Eigenmann, Z. Li, and S. P. Midkiff. Vol. 3602. Lecture Notes in Computer Science. Springer, 2004, pp. 253–267. ISBN: 3-540-28009-X.

[25] A. B. Yoo, M. A. Jette, and M. Grondona. “SLURM: Simple linux utility for resource management”. In: *Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.

[26] S. Zhou. “LSF: Load Sharing in Large Heterogeneous Distributed Systems”. In: *I Workshop on Cluster Computing*. 1992.