A Design Proposal for a Next Generation Scientific Software Framework

Anshu Dubey[1] and Daniel T. Graves[1]

Computational Research Division
Lawrence Berkeley National Laboratory,
Berkeley, CA 94720
adubey@lbl.gov

Abstract. High performance scientific software has many unique and challenging characteristics. These codes typically consist of many different stages of computation with different algorithms and components with diverse requirements. These heterogeneous algorithms, coupled with with platform heterogeneity, create serious performance challenges. To retain performance, portability and maintainability of the software on heterogeneous platforms, more abstractions have to be integrated into the software design. Most of these abstractions are still in the research stage and scientific codes have barely started using them. However, it is urgent that we start considering the abstraction interplay in designing the next generation of software architecture. We propose a software architecture for PDE-based scientific codes that combines three abstractions in a code framework suitable for expected heterogeneity in platforms, while retaining separation of concerns, performance and portability of the software. We support our proposal with an example design for an adaptive mesh refinement based framework.

1 Introduction

Scientific software used for simulation of complex multiphysics phenomena has many characteristics that make it uniquely challenging to architect and maintain. The codes have many different stages of computation, each with different algorithms and components. The components can have diverse requirements from the system hardware and software. Often these requirements conflict with one another, where an optimization for one is a detriment to another. The many moving parts in the application need to interoperate, but often the mechanisms used to achieve interoperability are awkward and inefficient. Furthermore, the codes typically run on high-end high-performance-computing (HPC) platforms which are expected to become increasingly more heterogeneous. The codes are expected to use the platforms effectively because these are extremely expensive and rare resources. It will take a combination of carefully thought out design, hard-nosed trade-offs, and good orchestration capabilities in the framework for a code to meet the demands placed upon it.

The main features of almost all successful frameworks of today are components that encapsulate a functionality, composability of those components, and

a separation of concerns. The frameworks are designed so that the sections of the code that pertain to the science of the application can be written and verified as sequential code. These sections are then plugged into the framework with a wrapper. The internals of the framework handle the parallelization and data management. There are several examples of such application codes and frameworks [5, 16, 20, 4, 15] that have served their respective scientific communities for a decade or more, and have even branched out to serve other communities.These codes are continuing to serve their communities, but with a change in the hardware paradigm, they are facing a crisis. In addition to heterogeneity in hardware architecture, the codes will have to deal with higher failure rates and deeper and more varied memory hierarchies. Furthermore, they have to do all this with steadily growing solver diversity which is the inevitable result of increasing the model fidelity. The abstractions that may help to tame some of the diversity are still mostly in the research stage. The application code developers have barely begun experimenting with these abstractions. As a result, there is little understanding about what design choices are likely to yield a long-lasting, nimble and portable high performance code that will stand the test of time.

For the forthcoming exascale era, the design space is bigger than ever with more programming abstractions to be incorporated into the software architecture. There has been investment in research on programming models and abstractions that could make it easier to code for the increasingly heterogeneous platforms of the future. There has also been some investment in algorithms that might better deal with new challenges such as decreasing memory per processing unit and the need to minimize data movement. How to architect scientific software with many moving parts into a composable whole, however, remains an open question.

Some consensus is beginning to emerge about which abstractions might prove to be the most useful. Here, we take note of three front runners among programming abstractions; namely, tasking, tiling and embedded domain-specific languages (eDSL). We propose a general software architecture that combines these abstractions in a way that separation of concerns is maintained or even enhanced. The abstraction lifting provided in the proposed architecture applies to a broad class of applications that primarily use partial-differential-equation solvers. We illustrate the applicability of this general purpose architecture with a more specific example of a framework for adaptive mesh refinement (AMR) based codes.

2 Requirements

Many scientific codes that are operating at petascale today have had years of investment in solvers for the physical phenomena they model. Typically, such codes tend to support a large community of researchers which makes code maintainability an important requirement. Even with the relatively uniform fat-node architectures that are still around, there is enough variability in code behavior that performance portability has been an important consideration. Most codes

avoid too many platform specific optimizations except in very limited performance critical kernels, where multiple alternative implementations might have existed for different platforms. As the diversity in the platform architecture is increasing this is becoming a bigger challenge. In the past, it was only a matter of performance, now it has also become a matter of being able to use the platform at all. In short, the need for platform-specific code appears to be increasing. Mitigating this challenge would require a framework to support code transformations for different target platforms while allowing the high level code to remain unchanged.

Many long standing obstacles to meeting the above requirements still exist. There has always been a trade-off between modularity and performance. Modularity is necessary for maintainable and reusable code, but it compromises performance. Even today, inter-procedural analysis is unable to deliver the performance across invocations of different functions. Flattening the code by fusing functions manually has proven to increase the execution speed considerably. Similarly, easy adaptability to various different architectures requires a small and nimble code base. This is at odds with the increasing code size from adding more capabilities for higher model fidelity. Some of these conflicting requirements were met in the past through a holistic approach to optimization, where trade-offs were carefully considered and adopted [6]. For example, most composable codes run at a very small fraction (3-8%) of the peak performance of the machine because they sacrifice performance for multiphysics and composability. In planning of simulation campaigns, the focus is on achieving the overall science objective. Sometimes this means using suboptimal options for individual components. The framework should be able to facilitate such unorthodox approaches and therefore should provide hooks for being able to make these choices.

With the advent of heterogeneity within and across platforms, code and performance portability have become much more critical. At the same time they have become much more difficult to achieve. This is tractable only if coding is done to higher levels of abstraction, with many more tools providing the intermediate levels of support. Normally, it would be more sensible to wait until the current research in abstractions has been hardened into products before considering how to use them to redesign the codes. However, the codes that need to change can be huge and take many person-years to refactor, while the platforms are already becoming more heterogeneous. Therefore, code redesign must begin as soon as possible.

3 Approach

One of the very first choices to make in designing a framework is which abstractions to use, and how to provide footholds for the abstractions in the framework so that they maintain separation of concerns. Every multiphysics high performance computing code has several levels of complexity, each one of which may need a different expertise. While it is true that such interdisciplinary projects demand overlap in the expertise of the participants, it is impossible for any in-

dividual to be an expert in everything. Instead, the software design has to be orthogonalized so that various complexities are untangled and experts can focus on what they know best. The complexity of the physical model is typically expressed in terms of parametrized equations. This is the knowledge that the domain scientists provide. The discretization of the equations and the numerical algorithms to solve them are best done by the applied mathematicians. In multi-component codes, where more than one numerical solver is in use, the issues of data movement and solver interoperability are best managed by the software engineers. The portability and performance of the code on various platforms are best handled by high performance computing experts. A good framework design would allow mostly independent development of these various aspects of the code without significant overlap in expertise.

One way to look at separation of concerns is to abstract the knowledge of the resources from the computation and vice-versa, and differentiate the physical view from the logical view. To illustrate this abstraction with an example, we consider domain decomposition of structured meshes. Here the logical view of the mesh is a self contained collection of discrete points where the "active" points are updated during solution evolution, while the "passive" points are only used in the computations, but are not updated. In rectangular grids the passive points are referred to as "halo" cells surrounding the active grid Here the physical view could be the entire physical domain where the halo points are filled with the values derived from the physical boundaries, or it could be an arbitrarily-sized section of the domain. The section itself could come from one or more physical boundaries or it could come entirely from the interior of the domain. The halo cells of the interior face of the domain section are, in effect, virtual points that hold the same values as the real points in the corresponding neighboring section of the domain, therefore they are also referred to as "ghost" cells in literature. The important point to note here is that the solver concerns itself only with the logical view, while under the hood, the infrastructure can manipulate the physical view to orchestrate the computation to best suit the target platform.

Similarly, it should be possible for a numerical algorithm to be expressed in a way that is essentially oblivious to the underlying micro-architecture of the platform. This means not just the layout of the processing units (cores or accelerators), but also the network, the coherency domains and other aspects of memory hierarchy. This can be achieved by writing solvers without explicit indexing into the data structures, but with the flexibility to operate on arbitrary chunks of provided data. One example of such an approach is stencil-based abstractions for mitigating the hardware dependence of solvers [11, 14]. This approach cleanly shields the numerics from the vagaries of the memory hierarchy and data movement. The orchestration of the data to be operated on can be left to the infrastructure.

In the next few sections we describe how we can leverage three of the abstractions that are gaining momentum to design a highly portable, extensible and performant scientific software. These abstractions are embedded domain-specific languages, tiling, and runtime dynamic tasking.

3.1 Embedded Domain-specific-languages

The idea behind eDSL's in our context is to provide abstractions in a high level language that address some domain's repeatedly-used functionality. The abstractions make the job of the compiler easier by constraining the semantics, allowing compilers to generate more optimized code. While it is difficult to persuade a community to switch over to an unproven language, eDSL's provide a way out. They live embedded within high-level languages, and therefore provide an on-ramp. An eDSL can be particularly useful to its target community if it takes care of abstraction lifting for the numerical solvers.

The current generation of computational kernels follow strictly prescriptive semantics. This is true not because of model or algorithmic requirements, but because the available programming models dictate it, the developers understand it, and it was effective on the machines of the time when the codes were developed. In truth, it is often completely unnecessary to be prescriptive in the description of operations, and DSL's such as the tensor-contraction-engine in NWChem, and Nebo-wasatch, an eDSL for updating stencils in Uintah, provide higher level constructs for precisely and concisely specifying the solver algorithm. Another example of successful use of eDSL on heterogeneous platforms is Stella [8]. In all these instances the back-ends transform the human-readable eDSL code into unreadable but highly performant code for the target platform. For each new platform one or more back-ends must be developed, but that is not a concern of the algorithm implementer. This results in a very robust separation of concerns between performance tuning and algorithm development.

3.2 Tiling

Because future machines will be more hierarchical, software design should orient itself to benefit from the hierarchy rather than be challenged by it. This means that the applications must identify the granularities within the models and the algorithms, and match the application granularity with the appropriate level in the machine hierarchy. To some extent, applications are already doing this with the mixed MPI-threading model. As a simple example one can consider a finite difference solver for a partial differential equation on a uniformly discretized mesh. Here the coarse-grain parallelism is achieved by decomposing the domain spatially, where each section of the decomposed domain is mapped to one MPI rank. The most prevalent way to achieve fine-grain parallelism is to exploit the relative independence of operations within loops of the computational kernels evolving the solution by threading the loops. In current multi- or many- core architectures, an MPI rank is typically the node, whereas threads map to different cores within the node. Thus different parallel granularities of the application are cleanly mapped to the two levels of platform architecture hierarchy. This approach is suboptimal because the resulting parallelism is fragile. It can easily break if the loop is modified without exercising sufficient care for it to remain thread-safe. It also violates the separation of concerns between computational kernel and parallelism. Furthermore, this approach also does not

address the issue of machine heterogeneity; the code has to change with every different architecture.

A more robust way to expose fine-grained parallelism is to use tiling, especially when one is operating with rectangular domains. A library like TIDA [19] provides hierarchy within tiles and assumes the responsibility of mapping tiles to the most suitable hardware resources. TIDA not only maintains the isolation of kernels from parallelism, but also separates the "within node" parallelism and memory layout handling from macro-parallelism.

3.3 Task Based Runtime Support

The currently dominant bulk-synchronous model is suitable for the distributed memory parallelization model. Enough computation is allotted to each processing unit that the cost of bulk-synchronization is amortized. Scientific codes have been designed for bulk-synchronization not because of any fundamental algorithmic necessity, but because that is a relatively easy way to satisfy dependencies. However, this mode of computation is known to have limitations when different components of the code demand different load distribution or when there is heterogeneity in the performance of individual processing units because of error correction or clock drift. The high performance scientific community is reaching consensus that dynamic runtime task scheduling could be a more desirable model. Dynamic runtimes have been demonstrated to be useful in many situations, such as dense linear algebra [9] and grid workflows [1]. There has been some success in large scientific codes as well, most notably Uintah [13, 12] using their own runtime and NAMD [17] using charm++ [10]. However, the majority of codes have either not integrated dynamic runtimes, or are in the early stages of experimentation. Many independent runtime systems and task based execution models are being developed in the community. Gilmanov et al. [7] provide a good summary.

3.4 Proposed Architecture

Figure 1 provides an overview schematic of scientific code architectures as they exist in many codes now (the left panel), and the modifications to the architecture with abstraction lifting that could help tackle the platform heterogeneity and massive parallelism. Note that in both views different virtual views (effectively programming abstractions) map to different concerns of the code. The boxes in the schematic are color coded to indicate the type of role played by the entity in the box in the overall scheme. Orange boxes represent the base code and its physical view. The pink boxes represent logical or virtual view of the corresponding quantities. The green boxes are for the tools that enable the added abstractions in the code. The blue boxes are for the target optimization. In the figure the top left hand corner box represents the whole application code covering the entire domain, including all the operators that need to be applied to it. Note that we make no assertions about either the discretization or the type of operators. The two branches out of this box represent the spatial and

functional decompositions. Again we place no constraints on what the decompositions ought to be for individual applications, each application will find its own granularities and decompose accordingly. The decompositions provide means for employing the logical views of the application in the framework design. Thus the spatial decomposition results in a logical view which allows the numerics to ignore the physical layout of the problem domain in the implementation. With appropriate abstractions in parallelization, for example tiling, and an eDSL for supporting higher level expression of the numerical algorithms, code transformation tools can be employed to optimize the computation and handling or memory hierarchy. The same abstraction also helps in alleviating the challenges posed by platform heterogeneity since code transformation tools can have different processing elements as their targets.

Similar to spatial decomposition, functional decomposition can be done at a granularity best suited for each specific application. In some instances, it would be at the level of one operator. In others, the operator itself may have inherent parallelism capable of finer decomposition. Here two different abstractions can come into play. One is a different class of code transformation techniques that allow operator fusion for memory access and compute optimization. The other is treating each decomposed function object as a task with some dependencies on other tasks. When viewed as tasks, the functions can be assigned to any resource at any time as long as their dependencies have been met. Thus this virtualization makes it possible to optimize for parallelization and scaling if a good dynamic scheduler can be built. In the next section we take these general principles and apply them to a framework for applications that use structured adaptive mesh refinement (SAMR).

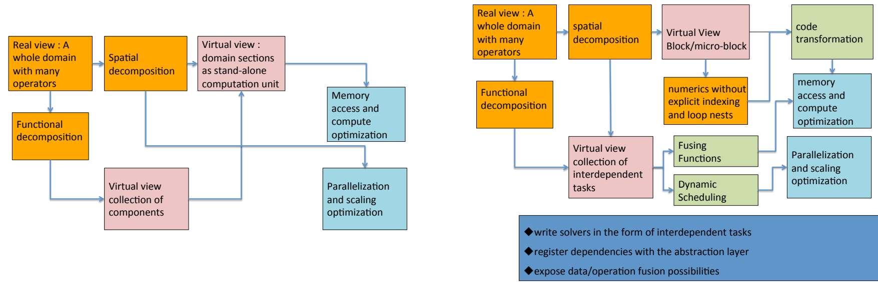

Fig. 1. Schematics of scientific code architecture. The left panel shows the current architecture, while the right panel shows one set of possible modifications to the architecture that could tackle heterogeneity.

4 Example: Structured AMR

Adaptive mesh refinement is a technique applied to logically Cartesian meshes when the target application has many length scales that live in different parts of the domain. Since the length scale dictates the spacing between the consecutive points in the discretized mesh, clearly both memory and computation are saved when fine resolution is used only where the physical conditions of the solution demand it (see [2]).

4.1 Granularities and Decomposition

In any multiphysics AMR code, there are two clear granularities that stand out: a block/patch and the physics operators. A block with its halo of virtual or ghost cells (as they are more generally known) is already used as the logical view of the domain by physics operators in almost all codes that rely on explicit or semi-implicit methods for solving partial differential equations. Thus a block readily maps to the third box on top in Figure 1. Similarly, most time integration follows operator splitting, which means that operators are applied on the solution data sequentially. Therefore they provide a readily available means of functional decomposition. The current generation code architectures don't view them as such, Uintah being one of the few exceptions. AMR Codes typically iterate over their collection of blocks for one operator, updating the block data as each operator finishes. Often this generates a false dependence, because in most situations the order of application of operators does not matter physically. A simple reformulation where the updates from physics operators are accumulated in a scratch space instead of being applied to the state data in the blocks immediately removes this artificial dependency. Now the application of individual operator on an individual block can be treated as a stand-alone task that can be scheduled when its dependencies are met. This provides us a map into the bottom left box in the schematic of Figure 1.

Another granularity that is inherent in AMR codes is that of a "level". Recall that each level has all blocks/patches that have same spatial resolution (dx). In most instances, the timestep dt also refines with the same ratio as dx, and is uniform across the level. There is some cross-level interaction. For example, reconciliation of physical quantities such as fluxes at the fine-coarse boundaries, ghost cell filling at those same boundaries and time synchronization of adjacent levels when needed. But levels manage their own meta information and time evolution. We can exploit these characteristics of the levels to enhance the macro-parallelism flexibility. The mapping of boxes in a level to different processing elements can be abstracted from the logical view of all boxes within the level. Once this is done, the processing elements allocated to levels can be adjusted as needed altering the logical view, and therefore bulk of the code.

4.2 Micro-parallelism

So far we have focused on virtualization of the decomposed space and functions along with macro-parallelism. In the many-core and accelerators based hetero-

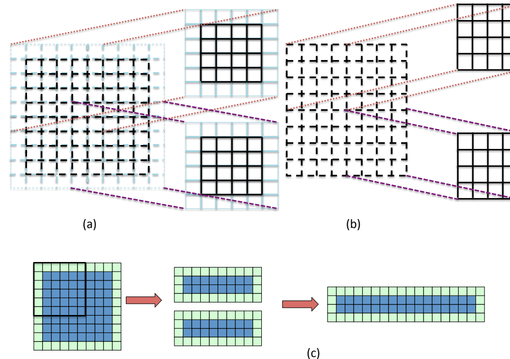

Fig. 2. Exposing micro-parallelism in a parameterizable way

geneous world, this is not enough. Finer-grained parallelism potential needs to be exposed in ways that can be harnessed flexibly for different architectures. In AMR, since we already have the abstraction of a "block", in principle it should be possible to break a block into smaller blocks and provide arbitrarily fine parallelism. The problem with that approach is that as the block size becomes smaller, the memory used for halo cells begins to dominate. The more common approach of explicit loop threading is not a very desirable solution because that makes the code harder to maintain, and needs to be retuned for every platform. A better approach is to eliminate the in-place update of the state within a block (recall that this is also desirable for functional decomposition). Figure 2 explains how this concept can be exploited to expose greater spatial parallelism through tiling. Here, if we parametrize the index space of the block in order to implement the operators, then the logical view of the tiles within the source block looks like panel (a) in the figure. Note that though the tiles have overlapping cells, they are read only, so they don't present any obstacle to threading. The destination block has similar logical and physical views with no overlap between tiles as shown in panel (b). Because of lack of overlap in the destination tiles there are no spurious dependencies or write conflicts and therefore no obstacle to threading.

Panel (c) shows how this parametrization also allows customization of tile shapes to best match the target processing element. A tile can be arbitrarily small, rectangular or thin, as long as its index space is provided as a parameter, the tiling abstraction can manage it. This is particularly useful when faced with processing elements that prefer to use large vectors or those that have small caches with small cache lines. Depending on the depth of memory hierarchy within the node, it may even be useful to coalesce some of the blocks (similar to Figure 2(c)) to map them to NUMA/coherence domains.

In our design we use dynamic scheduling within the node. This choice maintains the separation of macro-parallelism from micro-parallelism. Here, the load-balancing generated by regridding of AMR continues to be managed at the macro/distributed memory parallel level. The tasks are generated from the combination of spatial and functional decompositions described earlier. In order for the task based execution model to be useful there should be enough load within the node to amortize the overheads. Confining the dynamic scheduling to within the node limits the size of the dependence graph which can grow non-linearly if done globally. To enable dynamic scheduling very few changes needs to occur in the solver code. One change is that if the solver iterates over blocks explicitly, it has to either let that go completely or make it non-prescriptive (i.e. change from "for" to "while"). The second change is that the solver must articulate any dependencies that cannot be inferred by the framework from the index space information available to it. For example, if order of application of operators matters then the framework has to be informed of it, and the operators have to express it.

4.3 Solvers

AMR codes mostly rely upon explicit or semi-implicit solvers. If need for implicit methods arises, codes typically use dedicated linear algebra libraries which assume control for the duration of the solve. Hence for this discussion we ignore the challenges of implicit solve and focus entirely on explicit and semi-implicit solvers. The chief characteristic of these solvers is that an update of a point in the mesh can be described as a stencil. Stencils can be small or large, but they all collect information from their neighborhood which can be precisely specified geometrically. A stencil update is essentially a weighted sum of specified points in the neighborhood of the point being update. The neighboring points can be specified by shift relative to the target as shown in an example of five point stencil in Figure 3 (no weights are shown). Stencil computations and their optimization [18] is an active research area with many stencil DSL or eDSL languages under development. Some, such as Stella [8] and the Nebo-Wasatch in Uintah [3] have been successfully deployed in production codes with different back-ends for supporting different hardware. The motivation is the same, they seek to eliminate explicit indexing and dimensioning in expressing the involved arithmetic. Since solvers are the most computation-intensive parts of the code, a good code transformation tool from high level eDSL to optimized code will be critical in achieving performance portability across heterogeneity.

5 Conclusions

We propose a software architecture for the next generation of platforms by using a set of programming abstractions that are gaining ground in the community. Our proposed methodology focuses on using these abstractions in ways that function orthogonally to one another, and therefore do not diminish each others'

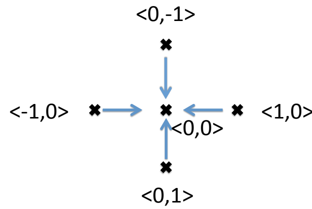

Fig. 3. Example of shifts in a five-point stencil update

impacts. Additionally, they maintain the separation of concerns which allows a team to map each member's expertise to the most appropriate aspect of code development. We cite efforts where a code has integrated one or two of these abstractions and has had success. We present a way of incorporating these abstractions seamlessly in a framework and we believe we have provided ample reasoning to support that claim. We firmly believe that the scientific codes of today must begin to be refactored to be prepared for future machines, otherwise scientific discovery through simulations will meet with a serious setback.

Acknowledgments: This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computer Research.

References

1. Ayyub, S., Abramson, D.: Gridrod: a dynamic runtime scheduler for grid workflows. In: Proceedings of the 21st annual international conference on Supercomputing. pp. 43–52. ACM (2007)
2. Berger, M., Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations. Journal of Computational Physics 53, 484–512 (1984)
3. Berzins, M., Luitjens, J., Meng, Q., Harman, T., Wight, C., Peterson, J.: Uintah - a scalable framework for hazard analysis. In: TG '10: Proc. of 2010 TeraGrid Conference. ACM, New York, NY, USA (2010)
4. Case, D., Babin, V., Berryman, J., Betz, R., Cai, Q., Cerutti, D., Cheatham Iii, T., Darden, T., Duke, R., Gohlke, H., et al.: Amber 14 (2014)
5. Dubey, A., Antypas, K., Ganapathy, M., Reid, L., Riley, K., Sheeler, D., Siegel, A., Weide, K.: Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. Parallel Computing 35(10-11), 512–522 (2009)
6. Dubey, A., Calder, A., Daley, C., Fisher, R., Graziani, C., Jordan, G., Lamb, D., Reid, L., Townsley, D.M., Weide, K.: Pragmatic optimizations for better scientific utilization of large supercomputers. International Journal of High Performance Computing Applications 27(3), 360–373 (2013)
7. Gilmanov, T., Anderson, M., Brodowicz, M., Sterling, T.: Application characteristics of many-tasking execution models. In: Proc. of the 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA). Citeseer (2013)

8. Gysi, T., Fuhrer, O., Osuna, C., Cumming, B., Schulthess, T.: Stella: A domain-specific embedded language for stencil codes on structured grids. In: EGU General Assembly Conference Abstracts. vol. 16, p. 8464 (2014)
9. Haidar, A., Ltaief, H., YarKhan, A., Dongarra, J.: Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. Concurrency and Computation: Practice and Experience 24(3), 305–321 (2012)
10. Kale, L.V., Bohm, E., Mendes, C.L., Wilmarth, T., Zheng, G.: Programming petascale applications with Charm++ and AMPI. Petascale Computing: Algorithms and Applications 1, 421–441 (2007)
11. Maruyama, N., Nomura, T., Sato, K., Matsuoka, S.: Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In: High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for. pp. 1–12. IEEE (2011)
12. Meng, Q., Luitjens, J., Berzins, M.: Dynamic task scheduling for the uintah framework. In: Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10) (2010), http://www.sci.utah.edu/publications/meng10/Meng_TaskSchedulingUintah2010.pdf
13. Notz, P.K., Pawlowski, R.P., Sutherland, J.C.: Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software. ACM Trans. Math. Softw. 39(1), 1:1–1:21 (Nov 2012)
14. Orchard, D.A., Bolingbroke, M., Mycroft, A.: Ypnos: declarative, parallel structured grid programming. In: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming. pp. 15–24. DAMP '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1708046.1708053
15. O'Shea, B.W., Bryan, G., Bordner, J., Norman, M.L., Abel, T., Harkness, R., Kritsuk, A.: Introducing Enzo, an AMR cosmology application. In: Plewa, T., Timur, L., Weirs, V. (eds.) Adaptive Mesh Refinement – Theory and Applications. Lecture Notes in Computational Science and Engineering, vol. 41. Springer (2005)
16. Parker, S.G.: A component-based architecture for parallel multi-physics PDE simulation. Future Generation Comput. Sys. 22, 204–216 (2006)
17. Phillips, J.C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R.D., Kale, L., Schulten, K.: Scalable molecular dynamics with namd. Journal of computational chemistry 26(16), 1781–1802 (2005)
18. Stock, K., Kong, M., Grosser, T., Pouchet, L.N., Rastello, F., Ramanujam, J., Sadayappan, P.: A framework for enhancing data reuse via associative reordering. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 10. ACM (2014)
19. Unat, D., Chan, C., Zhang, W., Bell, J., Shalf, J.: Tiling as a durable abstraction for parallelism and data locality. http://sc13.supercomputing.org/sites/default/files/WorkshopsArchive/pdfs/wp118s1.pdf (2013)
20. Valiev, M., Bylaska, E.J., Govind, N., Kowalski, K., Straatsma, T.P., Van Dam, H.J., Wang, D., Nieplocha, J., Apra, E., Windus, T.L., et al.: Nwchem: a comprehensive and scalable open-source solution for large scale molecular simulations. Computer Physics Communications 181(9), 1477–1489 (2010)