1 Introduction

The size and availability of data sets has increased significantly over the course of the past decade. Several technologies have been developed specifically to enable the analysis of very large data sets. These technologies have been dubbed "big data" platforms (Hadoop, Spark, Flink [1, 2, 3]). Big data platforms provide users the ability to compute count-based summaries, through database-inspired interactions, over large sets of information. The continued development and interest in Hadoop, the rise of the Spark platform, and growing interest in Flink, Theano [4], and Google's Tensorflow [5, 6] libraries demonstrate a concerted effort by academic, open source, and commercial interests to develop systems supporting large scale statistical computing. Another solution, Spartan [7] is a platform that takes a step toward providing a general purpose system for computations on distributed arrays for applied statistics on commodity cloud systems. Spartan extends Theano and Tensorflow in an effort to generalize array operations specifically to support distributed computing. Spartan decomposes array computations into a predefined set of parallel operations and employs a greedy algorithm to optimize execution and data layout using a user provided expression graph. The expression graph is provided to the Spartan runtime which optimizes the expression graph and schedules work, and infers the data layout on each compute locale using a greedy algorithm and user execution profile arguments.

This project will develop ***Phylanx***, an array-based and distributed framework targeted to HPC systems using the HPX runtime, which will overcome some of the limitations of existing systems such as Hadoop, Spark, and Flink.

***Phylanx* will build upon Spartan's ideas** and adds (1) more sophisticated algorithms to optimize data layout, distribution, and tiling on HPC systems by applying combinatorial optimization algorithms relying on submodularity functions which improves optimality guarantees beyond Spartan, and (2) the use of cache-oblivious data layouts (based on space filling curves) to improve the overall performance and scalability of the provided expression evaluations.

The overlap and interest in building a platform with statistical compute capabilities (note: many general purpose big data and Tensorflow platforms use "NumPy style operations" at scale) using these technologies along with a number of similar efforts hints at technical challenges associated with efficiently scaling these systems. Various publications confirm (see for instance [8]) that these existing big-data platforms face serious performance challenges. As an example, the distributed and local disk I/O (associated with Spark's distributed disk shuffle operation) and concurrency management overhead in Spark severely constrains the system's performance. Also in HPC environments, the challenge of scaling big data platforms requires a significant amount of runtime level modifications as outlined by [9]. ***Phylanx* will provide a generic framework** to develop a wide variety of big-data algorithms targeting HPC platforms aiming to overcome the mentioned limitations.

Consumers and developers of big data technologies continue to engineer solutions in order to increase the scope of their technologies to encompass problem sets beyond count-based summarizations. Big data technologies aim to enable statistical computing, machine learning, and data science capabilities matching the increase in processing and analytics requirements. The popularity of big data platforms spans several research and commercial ventures, including but not limited to marketing and financial analysis, bioinformatics, medical diagnostics, particle physics, autonomous vehicles, and image and natural language processing. However, a general purpose solution to scale sophisticated statistically oriented algorithms continues to elude the big data community. On the other hand, derivative technologies have been developed that graft new features on top of existing platforms to support scaling statistical algorithms. These efforts demonstrate that the existing systems require users to perform heroic engineering efforts in order to support each of the technologies' promises of resiliency for execution on commodity hardware. The challenges associated with scaling statistical algorithms are often related to the underlying "big data" computational paradigms and models. Scaling big data software ecosystems to perform more general purpose computations is an ongoing effort. Spark and Flink are modern endeavors to offer more general purpose processing capabilities built on top of the Hadoop File system. ***Phylanx* will provide programming methods, environments, and**

tools for these kind of computations which aims at reducing the overheads and improving scalability for a set of big-data algorithms by utilizing (1) static optimization techniques improving data layout, distribution, and tiling, and minimizing communication overheads based on the concrete expressions evaluated and by (2) building on top of HPX, a parallel runtime system which serves a flexible and portable implementation platform.

Continued interest and development directed toward finding a one-platform-for-all solution invites questions about performance limitations. A system providing research into performance/scaling trade-offs (for statistical algorithms and runtimes on HPC) does not currently exist. ***Phylanx* will provide an experimental platform** to the wider research community enabling the data and statistical computing capability necessary to investigate performance and scalability trade-offs for these algorithms at scale.

Big data systems achieve in-memory processing by using sophisticated cache policies to minimize distributed disk I/O. Local disk storage is used to "spill data" locally, when data set sizes exceed the heap size of the JVM. The overhead associated with caching, spill files, and the reliance on distributed disk storage as the mechanism for interprocess communication, imposes significant limitations that impact the performance of statistical algorithms operating on big data platforms. Developers have to invent techniques under several constraints imposed by the platform's implementation. Creating complex algorithms on these platforms requires a level of detailed understanding and access to the platform that can easily exceed the target user's expertise and level of access through the platform's APIs. ***Phylanx* will provide a higher-level interface** to the user from which it infers an optimized data placement and execution strategy, thus allowing domain scientists to focus on their work instead of having to delve into system specific idiosyncrasies.

Additional scaling complexity faced by big data platforms is associated with the implementations using third party libraries to provide primitives that enable asynchronous and distributed computing (e.g. Java's Netty library). Many third party libraries that provide asynchrony primitives use *epoll*, an operating system service, to detect asynchronous completions. The *epoll* system call consumes file descriptors. As big data technologies are required to scale to larger data set sizes, more file descriptors are consumed to maintain state in spill files on disk. This has a side effect of limiting *epoll's* functionality. There is a book-keeping cost associated with the use of epoll. Bugs related to the consumption of file descriptors have impacted the scalability of big data systems. The "garbage collection" of file descriptors to avoid leaks, a situation in which file descriptors are opened and not closed, has been an issue in these platforms. Applications leaking file descriptors can quickly reach operating system limitations causing the operating system hosting the technology to fail. The use of these libraries (based on *epoll*) is ingrained into the technology's implementations, and resides in places users are unable to touch directly. ***Phylanx* will build upon the dynamic task scheduling capabilities of HPX**, a modern, asynchronous task-based parallel runtime system which completely alleviates this problem. The dataflow-style capabilities exposed by HPX also help with optimally scheduling tasks, thus guaranteeing the preservation of all data-dependencies even for complex distributed workflows.

Using *Phylanx*, we will implement three benchmarks (out of the twelve benchmarks provided by Spartan) and three additional benchmarks not covered by Spartan, all of which have been chosen such that the infrastructure required for their implementation has to expose a wide variety of functionalities necessary for the implementation of a broader set of algorithms in the future. *Phylanx* will make use of a submodularity-based optimizer to layout, distribute and tile the used data in the best way achievable, and will use space-filling curves where such optimization is beneficial and appropriate. The use of dynamic, runtime-based adaptation through application specific policies in APEX will complement the implemented static optimizations. The broader research goals of the ***Phylanx*** project are to: (1) enable **best possible performance** capabilities for array-based big-data applications, both current and future, (2) develop and deliver a **practical and easy to use computing framework** for future practical big-data analysis, and (3) **provide programming methods, environments, and tools** for effective means of expressing array-based big-data applications for portable HPC system execution.

2 Background, State of the Art, and Preliminary Work

The data science and deep learning communities have embraced high productivity, rapid-development languages such as MATLAB, Julia, Scala, R, and Python. Data set sizes required to construct effective machine learning models exceed the capabilities provided by rapid-development languages. In an effort to improve the performance of machine learning algorithms implemented using rapid-development languages, domain specific libraries and runtime systems such as Theano, Tensorflow, and Spartan have been developed to support local and distributed data centric computations on commodity hardware.

Improved performance targets, and in some cases distributed processing, are achieved by Theano, Tensorflow, and Spartan by leveraging a mixture of capabilities offered by interpreters and computer algebraic systems. Each platform provides users with the ability to define computations using a NumPy interface. The NumPy interface provides users a level of indirection to a software system that encodes the user's NumPy-expressions into an expression graph. At execution time each system interprets and evaluates the user's expression graph. The interpreter component attempts to find an optimal evaluation of the expression graph using a combination of compiler techniques, heuristics, and computer algebraic simplification operations.

Theano users are able to define, symbolically, with "place holder variables," mathematical expressions using multi-dimensional arrays. Theano expressions are optimized at evaluation time using functionality blending computer algebraic optimization and compiler style optimizations. Theano makes optimization choices using rule-based pattern matching over the expression graph. Theano boasts GPU support, constant folding, subgraph merging, arithmetic simplification, BLAS function scheduling, memory aliasing, and loop fusion. Theano's optimizations target single-node algorithms for commodity hardware.

Tensorflow provides a system of computation similar to Theano. Users implement an algorithm in Python using placeholder variables. At evaluation time, users provide to the interpreter the expression graph and values, or files representing values, as inputs for placeholder variables. For distributed algorithms, Tensorflow's optimization choices are driven by heuristics related to when network copies are implicitly scheduled by the user's expression graph. For local computation, Tensorflow includes a *just-in-time* compilation technique to optimize execution of the user's expression graph. Tensorflow's optimizations target both distributed and single-node commodity clouds executing optimization algorithms for neural networks using unpublished heuristics. The Tensorflow user community has started several open source efforts to develop libraries extending the platform's support for non-neural-network oriented algorithms. All of the Tensorflow extension libraries are built for single locality computations. An opportunity exists for the development of a system built from conception for local and distributed execution of user-generated expression graphs.

Spartan is a platform that takes a step toward providing a general purpose distributed array system for applied statistical computing for commodity cloud systems. Spartan extends Theano and Tensorflow in an effort to generalize array operations specifically to support distributed computing. Spartan decomposes array computations into 5 parallel operations (filter, map, fold, scan, join_update) and provides a greedy algorithm to optimize execution and data layout using a user provided expression graph. Spartan operates in a manner similar to Theano and Tensorflow. Users interact with a NumPy interface to encode an expression graph. The expression graph is provided to the Spartan runtime which optimizes the expression graph and schedules work, and infers the data layout—or tiling—on each compute locale using a greedy algorithm and user execution profile arguments.

The HPX runtime system to be used by *Phylanx* provides the necessary abstractions to build efficient code which is oblivious to local and remote operation while maintaining data locality.

2.1 The HPX Runtime System

HPX is a general purpose C++ runtime system for parallel and distributed applications of any scale. It has been described in detail in other publications [10, 11, 12, 13, 14, 15]. In the context of this research we plan to use HPX because of its dynamic scheduling and global data addressing capabilities.

HPX represents an innovative mixture of components as shown in Figure 1. HPX is built using long-known ideas and concepts such as static and dynamic dataflow, fine-grained futures-based synchronization,

and continuation-style programming. However, it is the combination of these ideas and their strict application that form overarching design principles making HPX unique [12]. HPX aims to resolve the problems of scalability, resiliency, power efficiency, and runtime adaptive resource management that continue to grow in importance as computer architectures evolve from petascale to exascale, as the industry is facing increasing demands in supporting highly distributed, heterogeneous systems. Modern applications have to run on a varying set of resources that are mostly unknown at compile time. To achieve these goals, HPX departs from today's prevalent parallel programming models with the aim of mitigating traditional limitations, such as implicit and explicit (global and local) barriers, coarse-grained parallelism, and lack of easily achievable overlap between computation and communication.

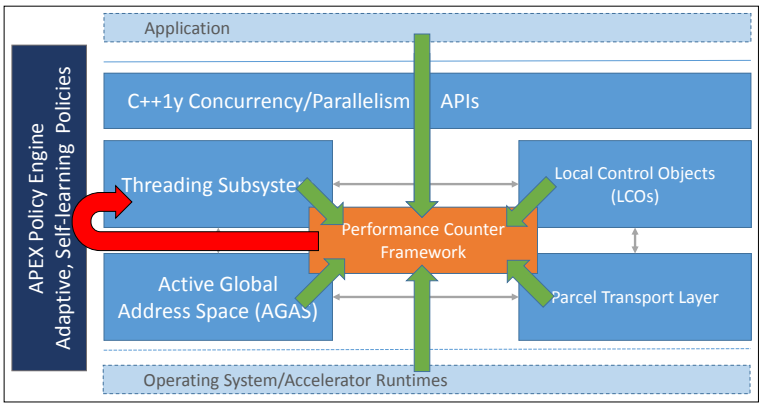

Figure 1: Architecture of the HPX runtime system. HPX implements the supporting functionality for all of the modules to be built for the *Phylanx* project. HPX represents an innovative mixture of a global system-wide active address space (AGAS), fine-grained parallelism, and lightweight synchronization combined with implicit, work queue based, message-driven, asynchronous computation; full semantic equivalence of local and remote execution; and support for hardware accelerators.

HPX exposes a coherent programming model unifying all the different types of parallelism available in today's computer systems. By modeling the API after the interfaces defined by the C++ standards, programmers are able to write fully asynchronous code using hundreds of millions of HPX-threads (tasks) in a familiar environment. This ease of programming extends to both parallel and distributed applications. HPX is the first open source runtime system to implement the concepts of the ParalleX execution model [16, 17] on conventional systems including Linux clusters, Windows, Macintosh, Android, Intel Xeon/Phi, and the IBM Blue Gene/Q. Further, HPX provides services and APIs allowing it to coordinate and manage code execution on GPUs and accelerators in distributed systems, such as Nvidia and AMD GPUs, and Intel Xeon Phis (Knight's Corner and Knight's Landing architectures). Support for other accelerators will be added in the context of the *Phylanx* project. HPX is published under a liberal, open-source license and has an open, active, and thriving developer and user community.

2.1.1 Active Global Address Space

The AGAS dynamically assigns system wide unique global addresses to objects. As such it unifies local and remote operations: code looks the same regardless of whether the target object is local or remote. Therefore, objects can be migrated to other localities (nodes), or migrated out of main memory to persistent storage and back into memory, a technique which is useful for load-balancing and optimized data distribution, but also to make the execution state of the system persistent. In the context of *Phylanx* we will rely on AGAS for efficiently accessing the partitions of the distributed arrays. Thus the algorithms we will design for working with those arrays can be locality-agnostic which simplifies their implementations significantly.

2.1.2 Threading/Synchronization

HPX uses a highly efficient threading system based on C++11/C++14 standardized/proposed facilities, it enables executors (a set of predefined or user supplied policies defining how to schedule threads and where to execute them), and various scheduling policies (FIFO, LIFO, priority based schemes, etc.). This threading system enables continuation style (data-flow style) programming supporting full asynchrony of execution which avoids global barriers and improves system utilization. The threads interact with a highly efficient constraint-based synchronization model which waits only for results that are necessary before continuing

a particular operation thus avoiding global barriers. Finally, the threading system allows fine control over which resources on a node are used by the *Phylanx* processing pipeline. The lightweight threading and synchronization provided by HPX do not rely on operating system consumables and will thus allow programmers to overcome the limitations of systems which rely on epoll (libev) for their synchronization needs.

2.1.3 Parcel Transport

HPX includes a modular networking (parcel) layer which has back-end modules for different network fabrics. Existing ports are TCP, MPI, and IBverbs parcel-ports. Others ports such as shared-memory, portals, or proprietary networks are readily implementable. This transport layer allows for dynamic expansion and shrinking of the number of localities connected together at any point in time (not possible with the current MPI parcel-port). This abstraction creates semantic equivalence between local and remote operations: local operations just create a new thread, remote operations create a parcel describing a new thread to be created on another locality.

2.1.4 Performance Counters/Policies

HPX includes a performance counter framework with a uniform interface for extracting arbitrary information including performance metrics, queue lengths, execution times of crucial functions, memory footprint, etc. There is also a policy based decision engine (see Section 2.2) that uses user defined (or predefined) policies to make decisions about parameter changes based on performance counters evaluated at user defined (or predefined) events that may be recurring, one time, or custom. As part of *Phylanx* we will perform research with the goal of determining the type of policies needed to dynamically adapt the runtime parameters of the processing pipeline (dynamic data placement, replication, caching polices, etc.) towards best possible IO bandwidth and minimal network overheads.

2.1.5 C++1y Concurrency and Parallelism APIs

HPX exposes a full set of higher-level parallelization facilities. These higher-level parallelization APIs have been designed to overcome limitations of today's commonly used programming models in C++ codes. The constructs exposed by HPX are well-aligned with the existing C++ standard [18, 19] and the ongoing standardization work (see for instance [20, 21, 22, 23, 24]). However, HPX goes beyond those with the goal of providing a flexible, well-integrated, and extensible framework for parallelizing applications using a wide range of types of parallelism [13]. Because HPX is designed for use on both, single and multiple nodes, the facilities we describe are also available in distributed use cases, which further broadens their usability. HPX exposes a uniform higher-level API which gives the application programmer syntactic and semantic equivalence of various types of on-node and off-node parallelism, all of which are well-integrated into the C++ type system. These facilities are not only fully aligned with modern C++ programming concepts, easily extensible and fully generic, they also enable highly efficient parallelization on par or better than existing equivalent applications based on bulk synchronous coding that is typical of OpenMP and/or MPI applications.

The differences between HPX and other parallel models and runtime systems like X10, Chapel, Charm++, Habanero, OpenMP and MPI have been discussed in detail in [12]. To summarize one comparison, the pragma-based constructs in OpenMP (such as `#pragma omp`) often feel misplaced in a modern C++ application since they operate outside of the C++ type system and are restricted operating within a locality. HPX, on the other hand, operates within the C++ standard and offers distributed parallelism. Other notable (and partially comparable) solutions are provided by the Intel Threading Building Blocks [25] and Microsoft's Parallel Patterns Library [26] which expose similar APIs as HPX. However, like OpenMP, all these solutions fall short when it comes to applications for distributed memory. Alternatively, X10, Chapel, Charm++, and Habenero operate in a distributed setting, but rely on providing a new language. We expect that many of the core algorithms for *Phylanx* can be implemented elegantly using HPX's higher-level API, which also opens up a natural upgrade path to acceleration.

As a complement to the distributed algorithmic constructs, HPX also supports distributed data structures

like `hpx::partitioned_vector` and `hpx::unordered_map`. These are modeled as closely as possible to the well-known container types of the C++ Standard library `vector` and `unordered_map` except that the underlying data storage allows for a flexible distributed data layout and placement across the nodes associated with a running application. Many of the algorithmic constructs in HPX directly support these distributed data container types.

2.2 APEX

The transition to extreme-scale computing poses new challenges in performance analysis and optimization because of the anticipated high concurrency and dynamic operation that will be required to make systems operate efficiently. Increasingly heterogeneous hardware, deeper memory hierarchies, reliability concerns, and constraints posed by power limits will contribute to a dynamic environment in which hardware and software performance may vary considerably during an application's execution. Furthermore, emerging exascale programming models like ParalleX emphasize message-driven computation and finer-grained parallelism, resulting in more asynchronous computation. Within this context, there is a compelling case for runtime performance observation that merges *first-person* (application perspective) with *third-person* (resource perspective) introspection, and for *in situ* performance analytics to identify bottlenecks and their impact on specific sections of code. This information drives online dynamic feedback and adaptation techniques that can be integrated with an exascale runtime system. The goal is to create an autonomic capability in the system that can direct the application performance to more productive execution outcomes. In addition, there is a need to provide applications with an adaptable feedback and control mechanism, so that they can trigger restructuring, algorithmic changes, or load balancing when key conditions are met.

We have implemented such a capability for asynchronous tasking models, called *Autonomic Performance Environment for eXascale (APEX)* [27, 28], a component of the HPX runtime system. APEX supports both introspection and policy-driven adaptation for performance and power optimization objectives. Autonomic behavior requires both performance awareness (introspection), and performance control and adaptation. APEX provides lightweight, low-overhead introspection from timers, counters, node- or machine-wide resource utilization data, energy consumption, and system health, all accessed in real-time. The introspection results are analyzed at runtime by policy rules in order to provide the feedback control mechanism. In addition, the performance measurements can be buffered out to performance measurement libraries like TAU [29], generating profiles or event traces for detailed post-mortem analysis and visualization.

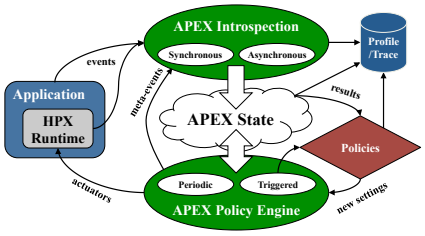

Figure 2: The APEX system design. Events are passed from the application and runtime to APEX, which updates the internal performance state and optionally buffers the trace events to the filesystem for post-mortem analysis and visualization. *Phylanx* proposes new trace events and semantics for post-mortem analysis of asynchronous tasking runtimes, including from the APEX policy engine.

The most distinguishing component in APEX is the *policy engine*. The policy engine provides controls to an application, library, runtime, and/or operating system using the aforementioned introspection measurements. *Policies* are rules that provide controls based on the observed state captured by APEX. The rules are encoded as callback functions that are periodic or triggered by events. The policy functions query the APEX state, extracting updated profile values from any measurement collected by APEX. The functions can change application or runtime behavior by whatever means available, such as throttling threads, changing algorithms, changing task granularity, or triggering data movement such as mesh refinement or repartitioning. We hypothesize there will be new policies that emerge from our experiences with the *Phylanx* project. Any of the proposed benchmark applications or underlying components may have faster performance or execute

more efficiently given an opportunity for dynamic behavior. The tiled array optimizer could benefit from APEX policies that define performance expectations and evaluate the solver progress to validate and/or correct them. The *Phylanx* framework or even the HPX runtime itself could benefit from policies with respect to parcels, parallel algorithms or auto-chunked execution.

2.3 The Tensor Contraction Engine

The Tensor Contraction Engine (TCE) [30, 31, 32, 33, 34, 35] is a source-to-source translator that generates optimized code for a variety of targets from source code consisting of properly structured loop nests in which arrays are accessed using affine combinations of loop indices. Starting with a *polyhedral* representation of the original loop nests a variety of mathematical techniques are applied to find transformations of the loop nests into forms which facilitate parallel implementation, improve locality, and realize other benefits. TCE has been used to optimize locality levels beyond main memory [36]. It has been used with a DSL targeting computational chemistry, and related efforts provide more general DSLs [37]. One of the important issues addressed in TCE in data layout selection in the context of tensor operations [35]. This approach combines the best features of empirical optimizations, namely, the incorporation of complex behavior of modern architectures, and a model-driven approach that enables efficient exploration of the search space through modeling the cost of constituent operations. We will integrate this model-driven approach with the algorithmic techniques described in Section 2.4.

In *Phylanx* TCE, along with an existing DSL, will provide a general loop nest facility that will be processed by a polyhedral model based compiler such as Pluto [38, 39, 40, 41]. TCE will be adapted to transform suitably formed IR derived from other *Phylanx* DSLs, providing the code generator with more powerful transformation facilities. TCE code generation capabilities, including those for Intel vector units, will be integrated into the code generator. In addition, the data layout selection techniques used in TCE [35] will be further developed for *Phylanx*.

2.4 Performance guarantee for data tiling

Spartan employs a greedy algorithm to determine a tiling scheme for the data that attempts to minimize the cost of communication between nodes. It is known [7] that finding the minimum-cost solution exactly is an NP-hard problem, so the authors of Spartan employ a simple greedy algorithm with no quality guarantees.

We observe that, even in the case of NP-hard problems, it is often possible to say something about the quality of the solution. We can either produce an optimal solution that is found in reasonable time for most instances (using *Smoothed Analysis* [42]), or produce a solution in polynomial time whose cost is within a constant fraction of the optimum. Since smoothed analysis requires assumptions on the input that we cannot make (our input is user-produced code), we will focus on the latter: *Find an efficient algorithm that assigns data tilings to an expression tree, such that the output tiling has cost at most $c \cdot OPT$, where c is a constant and OPT is the cost of the optimal tiling.*

The theory of *Approximation Algorithms* [43] provides us with various tools for this, including *Primal-Dual methods* [44] (see [45] for an application) and *Submodular Function Optimization* [46].

A third route that we wish to explore is that of *Fixed-Parameter Tractability* [47], where we seek to identify a parameter of the input that is responsible for the hardness of the problem. Controlling such a parameter would yield a polynomial-time algorithm for the problem.

2.5 Submodular function optimization

Many optimization problems take on the form "$\max_{S \subseteq V} f(S)$ such that S is feasible," where V is a finite set and $f : 2^V \to \mathbb{R}$ assigns values to subsets. We will focus on the case where f is *submodular*, which intuitively means that selecting an element earlier will increase the objective value more than selecting it later: the *marginal return* of selecting an element is diminishing as the selected set grows. Submodular function maximization problems have been studied extensively, and though the problem is NP-hard, successful approximation strategies exist. Notably, the celebrated result of Nemhauser and Wolsey [46] provides an algorithm that produces a set S with $f(S)$ within a $(1 - 1/e)$ factor of the subset while maximizing f.

Our interest in submodular function maximization stems from the hypothesis that the data tiling problem from the Spartan project [7] exhibits submodular properties.

The classic submodular optimization problem requires random access to all input data. This may not be feasible for our application. In that case, an online algorithm, or a streaming algorithm such as that of Badanidiyuru et al. [48] could give us the desired outcome, at a modest cost to the optimality guarantee.

We will also consider parallelizing the optimization step in order to mitigate possible long execution times, and the effect on the approximation ratio of such an approach. Given that users' code does not always match the capabilities of the available hardware, if our first objectives proceed smoothly, we may investigate alternative expression tree optimizations.

3 Proposed Research

Phylanx proposes to create a Spartan-like tool targeted to HPC systems using the HPX runtime. Our compute graph interpreter will compare Spartan's greedy scheduling algorithm against a scheduler using a greedy submodular optimization algorithm. The applied statistical computing interpreter will provide users the ability to implement NumPy-styled expression-graphs using Python or C/C++. Expression graphs will be serialized and sent to an HPX-enabled interpreter for lazy optimization and evaluation.

Spartan's tiled array optimizer uses a greedy selection algorithm, informed by data access patterns present in the user-submitted expression graph, to find a "best layout" format for an array tile stored on a locale. Spartan's tiled array optimizer provides no optimality guarantees.

Matroid constrained submodular functions provide a mathematical construct for describing diminishing returns. Submodularity-based techniques provide a mathematically provable guarantee of optimality approaching $(1 - 1/e)$ in machine learning problem domains and in sensor placement problems [49, 48, 50] Given a graph representation of a user's computation and a model of the cost of constituent operations [35], a submodular driven version of the tiled array optimizer would be able to select array tile storage format based on the diminishing return valuation related to a particular format given previously-selected storage formats and user-selected access operations. Submodularity can also be extended into the scheduling of operations represented in the user's compute graph by greedy prioritization of distributed operations informed by a diminishing return associated with operator performance. Including a mathematician with a strong background in combinatorics into our team of researchers will ensure a solid theoretical background for applying submodular functions to our optimization strategies. Including a computer scientist with significant array-based compilation and DSL technology expertise will ensure choice of good optimization strategies.

Spartan makes 50+ of the most commonly employed NumPy functions available for use on distributed arrays, and sizes and locates the data on these arrays based on the kinds of operations which will be applied to the data [51]. Beyond that, the HPX interpreter will optimize the user's expression graph, construct cost models of constituent operations, and determine array data layout and indexing scheme using a submodular optimization technique. Submodular optimization exhibits a diminishing return property which has demonstrated usefulness in several learning and summarization problems. Submodular optimization will be used to select distributed array layout based upon array access patterns (constraints) encoded by the users compute graph. Data layout and tiling schemes will extend Spartan's dense and sparse column/row formats by including the space filling curves (Peano, Z-order, Hilbert) storage options.

A mixture of algorithms have been selected to demonstrate the platform's goal of achieving a general purpose solution. The platform will provide implementation of the 3 benchmarks mentioned in the Spartan paper as well as 3 additional benchmarks that have been selected for their domain specificity in text, image, and graph applications. The platform will provide an implementation of the following benchmarks: (1) **Cholesky Decomposition**, is an algorithm with $O(N^3)$ complexity and very strong data dependencies; (2) **Singular Value Decomposition**, which is a very fundamental algorithm in scientific and statistical computing; (3) **Logistic Regression with Conjugate Gradient**, is a representative algorithm from the family of linear models used in statistics; (4) **Latent Dirichlet Allocation**, an algorithm popular for it's ability to identify themes associated with a large set of documents; (5) the basic **neural network algorithm**, underlying

the family or related algorithms popularized by the deep learning community; (6) inspired by GraphBLAS, **Jaccard Index** and **breadth-first-search** (BFS) will be provided.

As with any high performance computing project, computing speed and efficiency are primary goals. To that end, we will incorporate periodic performance regression testing with APEX and TAU, tracking design implementations to validate expected performance gains. New APEX policies will be developed to steer the HPX runtime with respect to communication efficiency, auto-chunking of parallel loops and guided execution policy selection. We hypothesize that the benchmark applications and the execution graph optimizer will also benefit from dynamic adaptation and feedback and control policies.

In summary, this project will provide an implementation of Spartan and three of its benchmarks that will make use of a greedy submodular optimizer, will make use of space-filling curves where such optimization is beneficial and appropriate, and implement three benchmarks not covered by Spartan. The use of dynamic, runtime-based adaptation through application specific policies in APEX will complement the implemented static optimizations.

The broader research goals of the *Phylanx* project are to:

- enable **best possible performance** capabilities for array-based big-data applications, both current and future,
- develop and deliver a **practical and easy to use computing framework** for future practical big-data analysis, and
- **provide programming methods, environments, and tools** for effective means of expressing array-based big-data applications for portable HPC system execution.

3.1 Proposed System Architecture

In the context of *Phylanx*, we will rely on the HPX runtime system which is an example of what we call a 'target independent programming' system. The programmers define the parallelism in the program, not just how one might want to exploit the machine. This is specified in an abstract way in which units of work are scheduled by explicitly describing data dependencies between them to the underlying runtime system. In response, the system creates a dynamic task graph corresponding to the real data dependencies as extracted from the algorithms. The task graph is then scheduled – either automatically or based on additional information as provided by the expression tree annotations, runtime based information, or extracted from a machine learning module – and run on the available computational resources with the best possible performance. The system does this in a way which maximizes the use of the available memory hierarchies and the use of the available heterogeneous resources.

One of the main goals of the *Phylanx* project is to develop a software infrastructure for efficiently processing large amounts of array data. We envision for this infrastructure to be designed as a set of software libraries bridging the gap which today exists between two traditional ways of processing "Big Data." On one hand, in academia and the high-performance community, we have seen the implementation of distributed algorithms using hand-crafted solutions in C/C++ using MPI. These code are usually fairly efficient, but often difficult to write and to maintain. On the other hand many companies have created their own software ecosystem. Prominent examples for this are Google's MapReduce [52], or Apache's Hadoop, Spark, and Flink projects [2, 3]. These frameworks provide an interface which is simpler and promises automatic scheduling, data distribution, and fault tolerance. Unfortunately, these systems often lack scalability with the CPU as the bottleneck [53, 54].

We propose to build a system which exposes a library of high-performance algorithmic primitives which will be dynamically tied together at runtime to achieve the desired effect. The system will use a higher-level description of the desired data processing algorithm in Python, possibly syntactically derived from the well-known NumPy and SymPy Python libraries [55, 56]. The user produces a set of NumPy/SymPy-style expressions in accordance to the required data processing algorithms. In addition to the elementary operations as provided by NumPy arrays (numeric operations, array transformations, etc.), we will design and develop higher-level algorithmic blocks as described in Section 3.2 which the user can employ.

As a first step in the processing, the *Phylanx* libraries will create a storable (as a file) representation of the expression tree corresponding to the original set of Python expressions. This expression tree serves several purposes: a) it provides a possible API for other potential tools, such a GUIs or direct C++ applications which produce such expression trees as part of their workflow; b) it will be used as the medium connecting the Python front-end library with a C++ framework which *Phylanx* will design and which will serve as the platform to optimize the expression tree and—given the appropriate data sets—to perform the required operations; and c) it provides a means of archiving arbitrarily complex expression trees for later use. The nodes of the expression tree will represent operations to perform on the data arrays which are produced either by an input operation or by a preceding computational step. These dependencies are represented by the edges of the expression tree. *Phylanx* will identify, select, and develop a set of operations it makes available to the user as the possible building blocks for the overall algorithm he/she wants to implement.

As a second step in the processing, the *Phylanx* pipeline will analyze the expression tree in order to find an optimal data layout (tiling) and data distribution over a set of compute nodes for the input, intermediate, and output data, ensuring an optimized execution time (aiming at minimizing communication overheads, utilizing cache hierarchies of the target architecture, and best possible utilization of vector pipelines, etc.). This process should also identify the number of compute nodes necessary to perform the required operation on a cluster of a given architecture. We propose to perform research with the goal to develop a better optimization algorithm than implemented by Spartan [7], as Spartan's tiled array optimizer provides no optimality guarantees. For this we plan to employ submodularity based techniques which provide a mathematically provable guarantee of optimality approaching $(1 - 1/e)$ in machine learning problem domains and in sensor placement problems [49, 48, 50]. This processing step will be implemented as a C++ module using the HPX runtime system which enables wide parallelization capabilities necessary for running such algorithms in a distributed setting. We expect for this processing step to generate an optimized and annotated expression tree similar to the one produced by step one. The optimization process itself may turn out to be very time consuming, thus having the ability to store and archive a preprocessed (and optimized) representation of the optimized set of numeric expressions will be very useful.

The third and central step in the processing interprets (and executes) the annotated expression tree. This step involves loading the (input) data in accordance with the data layout and distribution as determined from optimizing the expression tree during the second step. For the scheduling and execution, we propose to utilize the distributed dataflow-style scheduling and execution facilities as provided by HPX. This will automatically make sure that tasks (nodes on the expression tree) will be scheduled and run only if all dependent data has been produced (as defined by the edges of the expression tree). Each node of the expression tree represents a task which has to be performed on one or more input arrays and produces output arrays as mandated by the initial expression provided by the user. The task is executed next to where the data is located in the system as soon as all the input data arrays have been made available by their producing tasks. No global barrier-synchronization is performed as the computation is fully driven by the availability of the data required for a particular execution step. Generally, HPX moves the work to the data (e.g. schedules a task at the locality where the data is located, thus by properly tiling and distributing the data across the system we automatically achieve load balancing as the algorithmic nodes will execute code close to where the corresponding data is located.

Special consideration will be given to input/output operations. These tasks will read/write the data based on the required data layout as described by the annotated expression tree. Input/output will be integrated into HPX's dataflow scheduling in a way allowing to make other work depending on it. The data read from external storage will be directly placed in the memory of the node. We will use existing HPX facilities such as the `hpx::partitioned_vector` and the HPX co-array implementation [57] to store the distributed arrays in memory and for higher-dimensional data handling and SPMD style data processing. This simplifies the use of standards conforming algorithms as provided by HPX. We also plan to create new HPX distributed data storage containers as necessary and add support for data layouts based on space filling curves.

One of the central research topics of *Phylanx* will be the design and implementation of the algorithmic

blocks of which the initial expressions can be built. Those blocks have to be sufficiently fine-grained for them to be usable as building blocks of more complex algorithms, but on the other hand should be as coarse-grained as possible to be able to encapsulate sufficient amount of data processing needed for the data processing algorithms to be built (see Section 3.2). During the research for *Phylanx*, a list of building blocks to be discovered and used to implement those benchmarks. Those building blocks will utilize and extend on HPX's existing parallel algorithm and distributed data structures modules.

In all of the design and development of the *Phylanx* software infrastructure, we plan to maximize the re-use of already existing libraries. Notable examples of such libraries are HDF5 [58] we plan to use as the main data representation format and the well known C++ Hypertable library [59] for distributed persistent data access. For highly efficient dense and sparse linear algebra, we will investigate using one (or more) of the related existing modern C++ libraries, such as Eigen [60], Blaze [61], and SuiteSparse [62].

3.2 The Six Benchmarks of *Phylanx*

We propose to focus on developing six benchmarks: Cholesky Decomposition, Singular Value Decomposition, Logistic Regression with Conjugate Gradient, Latent Dirichlet Allocation, Neural Networks, and Jaccard Index and BFS from GraphBLAS. Those will be used as the driving applications and proof of concept applications for the *Phylanx* framework.

We intend to replicate and enhance the functionality of Spartan across three of the twelve Spartan algorithms by making use of (1) submodular algorithms, and (2) cache-oblivious algorithms.

3.2.1 Cholesky Decomposition

Cholesky Decomposition is a well-studied and tested algorithm in distributed computing. The algorithm has an $O(N^3)$ complexity with strong inner data dependencies. Cholesky will motivate the performance study of the underlying distributed storage data structures and their associated communication costs. This algorithm has been studied in [7, 63].

3.2.2 Singular Value Decomposition

Singular Value Decomposition is a fundamental algorithm in scientific and statistical computing. Like Cholesky, the algorithm is well-studied and it's performance characteristics are equally well understood. It provides a mechanism for studying data layout, exercising the memory hierarchy, and BLAS operations. A version of this algorithm implemented in this platform provides a reusable capability for several other statistical algorithms.

3.2.3 Logistic Regression w/Conjugate Gradient

Logistic regression is an introductory classifier representative of a family of linear models used in statistical computing. Conjugate Gradient exercises the functionality of the code in many important ways. For this reason it is being considered as a replacement for the HPL benchmark currently used by Top500.org to classify supercomputers [64, 65]. The proposed benchmark is called HPCG. While the actual HPCG benchmark cannot be made part of this proposal (the code design is fixed), we do intend to make our CG benchmark resemble it as closely as possible. As noted by the advocates of the HPCG benchmark, CG tests sparse matrix-vector multiplication, vector updates, global dot products. local symmetric Gauss-Seidel smoothing, and sparse triangular solves (as part of the Gauss-Seidel smoother). Logistic Regression optimized using Conjugate Gradient, will provide a well-understood benchmark for performance that can be compared against Spartan and other tuned implementations.

3.2.4 Latent Dirichlet Allocation with hyper parameter optimization

Latent Dirichlet Allocation (LDA) is an algorithm popular in the natural language processing community. LDA is described in the literature as a 3-level hierarchical Bayesian model. Training the algorithm can be achieved using Bayesian techniques. Collapsed Gibbs sampling is a preferred technique that has been studied for distributed training. This algorithm was selected to demonstrate system performance on text

analysis problems, Bayesian statistical problems, and the randomized nature (Markov Chain Monte Carlo or simply MCMC) of the Gibbs sampling algorithm. For a description of this algorithm, see [66, 67, 68].

3.2.5 Neural Networks

Training neural networks is the most computational expensive aspect of using this particular family of classification algorithms. The algorithm makes heavy use of dense-matrix, dense-matrix multiply; dense-matrix, dense-vector multiply; dense-vector, and dense-vector multiply operations. Scaling the training process to several machines is an area of significant commercial investment. This algorithm is the starting point for deep learning and provides an opportunity to study how to scale some of the more popular optimization algorithms (Stochastic gradient descent, L-BFGS, or fmin-cg) to an HPC environment (see also: [69, 70]).

3.2.6 Jaccard Index and BFS using GraphBLAS primitives

The Jaccard index is a graph algorithm for measuring similarity between two vertices. Jaccard Index has been presented as a candidate application for benchmarking GraphBLAS operations (Graph500, GAP, HPC Graph Analysis, Kepner and Gilbert). This algorithm provides an opportunity to study sparsity, concurrency/parallelism, communication-avoidance, and data locality trade-offs. Jaccard Index will test *Phylanx*'s ability to operate efficiently on a variety of sparse matrix/vector data sets and operations. Specifically, a Jaccard Index implementation will test the following GraphBLAS kernels [71, 72, 73, 74]: SpGEMM, SpMV, SpEWiseX. Users should be capable of implementing these kernels, and others, at a high-level. The *Phylanx* platform provides a novel environment to study the possibility of supporting GraphBLAS operations given the other algorithms that have been proposed. For the Jaccard Index, we wish to compute the overlap in an unweighted, undirected graph, represented as a symmetric, sparse, $N \times N$ matrix. The Jaccard index for a pair of vertices, x and y, is as follows: $J_{xy} = \frac{|N(x) \cap N(y)|}{|N(x) \cup N(y)|}$ Where $N(x)$ is the set of neighbors of x.

While this project focuses on arrays and not graphs, a sparse $N \times N$ matrix can be used to represent a graph of N nodes. In this context, BFS is a sparse matrix, sparse vector multiply which is important for many algorithms. BFS is difficult to implement efficiently because it is typically communication intensive when performed in a distributed setting. BFS is also important for calculating spanning forests, maximum flows, betweenness centrality, etc.

3.3 Algorithmic Improvements

The *Phylanx* will perform research on how the algorithms listed in 3.2 can be advanced beyond what was implemented in Spartan. We propose to improve those in two main directions: (1) by applying techniques relying on submodularity and (2) by using cache-oblivious algorithms.

3.3.1 Submodular Algorithms

A *submodular function* maps subsets to real numbers such that adding an element has diminishing returns as the base set grows. In particular, if E is a finite set and $f : E \to \mathbb{R}$ is a function, we define the *marginal contribution* of element $e \in E$ to subset $S \subseteq E$ as $\Delta(e|S) := f(S \cup \{e\}) - f(S)$. The function is *submodular* if, for all e and all $A \subseteq B \subseteq E - \{e\}$, we have $\Delta(e|B) \leq \Delta(e|A)$. Such functions arise naturally in many optimization contexts. An integer-valued submodular function with unit increase is known as a *matroid*. See Krause and Golovin [75] for an extensive discussion of submodular function maximization and its applications in big data. Particularly promising for our plans are results on streaming submodular optimization by Badanidiyuru et al. [48], on training data distribution by Wei et al. [76], and on document summarization by Lin and Bilmes [77].

We will exploit the fact that the presence of submodularity opens up algorithmic techniques for otherwise intractable problems. While exact solutions are still infeasible, constant-factor approximation algorithms become available. Notably, the celebrated result of Nemhauser and Wolsey [46] provides an algorithm that produces a set S with $f(S)$ within a $(1 - 1/e)$ factor of the subset maximizing f.

The Spartan paper uses the greedy algorithm to select a data layout from operations users have annotated to the distributed tiled array nodes in the compute graph. The greedy algorithm in the Spartan paper provides

no guarantees of an optimal selection. The data layout automation can be framed into a submodular subset selection problem to provide a guarantee of optimal (subset) selection.

3.3.2 Branch width and tangles

A popular measure for the complexity of the input to an optimization problem involving graphs is *branch width*, a parameter that encodes how "tree-like" the input graph is. For low branch width, dynamic programming techniques can often overcome the NP-hardness of a problem and provide an exact solution efficiently (see e.g. [78]). For high branch width, a dual notion, that of a *tangle*, can be used to cluster the input, thus decreasing communication between nodes. [79, 80]

Both notions are built around *connectivity functions*, which are symmetric, submodular functions. A proof-of-concept paper on the use of tangles in image analysis was written by Diestel and Whittle [81]. Similar ideas could be explored for our data tiling problem.

3.3.3 Cache-Oblivious Algorithms

The idea behind cache-oblivious algorithms is efficient usage of processor caches and reduction of memory bandwidth requirements. Both things are equally important for single-threaded algorithms, but especially crucial for parallel algorithms, because available memory bandwidth is usually shared between hardware threads and frequently becomes a bottleneck for scalability. Those algorithms are oblivious of particular cache hierarchy and cache parameters and make efficient use of whatever cache hierarchy/parameters. Cache-oblivious algorithms perform well on a multilevel memory hierarchy without knowing any parameters of the hierarchy, only knowing the existence of a hierarchy [82]. An algorithm is cache oblivious if no program variables dependent on hardware configuration parameters, such as cache size and cache-line length need to be tuned to minimize the number of cache misses.

We propose to use space-filling curves (Peano, Z-order, Hilbert) as an internal representation because it makes it possible to perform operations such as matrix multiplication without jumps in indexing. Thus, it is able to get a performance benefit from the cache without specifically knowing the details of any machine's cache structure. Other algorithms, like matrix transposition, sorting, or Jacobi pass filters have been investigated and have been shown to be implementable using cache-oblivious algorithms as well [83, 84].

In *Phylanx* we propose to investigate the applicability of those techniques to the algorithms we will implement.

4 Milestones

4.1 Year 1: Software Design, System Analysis, Prototypical implementation

During the first year of the project, a plurality of the effort will be directed towards designing and implementing the *Phylanx* infrastructure with HPX, and providing high-level interfaces to the algorithms in C++ and/or Python.

Concepts Identify algorithmic blocks required to represent the applications to be implemented, as described in Section 3.1.

Python layer Design and implement the first version of a Python layer producing an expression tree corresponding to the provided set of Python expressions.

Optimizer Design and implement first version of optimizer for a subset of algorithmic blocks. This will include a generation of performance assumptions, along with APEX policies to validate the assumptions and/or re-evaluate the solution strategy if the performance assumptions are not met.

Algorithmic blocks Design and implement first algorithmic blocks with provable performance guarantees, as described in 3.3.1.

HPX Adapt existing facilities in HPX to requirements of the rest of the framework, and create new parallel algorithms and distributed data structures as needed. APEX will be fully integrated and policies enforced for dynamic HPX behavior.

Performance Analysis Establish performance regression testing for *Phylanx*, tracking progress as algorithmic blocks are implemented and evaluated. Periodic performance benchmarking of *Phylanx* components and applications using APEX and TAU. Development of APEX policies for feedback and control of HPX resources.

Benchmark Applications At least two benchmark targets will be prototyped during this period, including a distributed linear algebra framework and the Logic Regression with Conjugate Gradient (HPCG); see Section 3.2.3. Unit tests and correctness testing will validate results while performance regression testing will validate implementation performance.

4.2 Year 2: Evaluation, Development, and Implementation of Software

The second year of the project will focus on the design and implementation of the remaining benchmark applications, as well as refinement of the *Phylanx* infrastructure, HPX and APEX components. The submodular optimization of execution graphs will be evaluated and refined for correctness and performance.

Python layer The Python layer will be evaluated and refined as necessary to support the requirements of all of the benchmark applications and the framework in general.

HPX Tune runtime functionality based on first performance evaluations of implemented parts of the system.

Optimizer Evaluate and refine the optimizer for the subset of algorithmic blocks. Expand the evaluation to include the additional benchmark applications developed in year 2.

Algorithmic blocks Evaluate and refine the algorithmic blocks as necessary to support the additional benchmark applications.

Benchmark Applications Implement the remainder from the list of applications including Cholesky Decomposition 3.2.1, Singular Value Decomposition 3.2.2, Jaccard Index 3.2.6, Neural Networks 3.2.5 and Latent Dirichlet Allocation 3.2.4.

Performance Analysis Continued performance regression testing for *Phylanx*. Continued performance benchmarking of existing applications and performance evaluation of new application benchmarks using APEX and TAU. Evaluation and refinement of APEX policies for feedback and control of HPX resources.

5 Broader Impacts

Phylanx has the potential to have an effect well beyond the field of Big Data. By providing scientist and application developers with a sound framework for performing scalable statistical analysis on big data sets, new types of applications will be able to be written and maintained. *Phylanx* is designed in such that user code will be able to perform efficiently on current and future architecture as long as the runtime system is maintained. This greatly reduces the maintenance burden and will increase the *productivity* of programmers in these fields. *Phylanx* will also have direct societal benefit as well. The lead institution resides in an EPSCoR state. Funding this research fosters the growth and development of HPC in Louisiana. This provides undergraduate, graduate, and post graduate opportunities to the citizens of Louisiana which is vital to assist the State in fostering old and creating new industries with HPC technology. *Phylanx* in particular lays a solid foundation for technology transfer from academia to industry. This project will help fill the gap between academic innovation and commercial application, by creating a software layer that industrial partners can feel confident relying on. Finally, *Phylanx* funds will support societal values by consciously identifying and supporting under represented groups in STEM. The lead institution has a long track record of hiring and training underrepresented minorities. Additionally, the LSU team will support the Beowulf Bootcamp, an HPC summer camp for high school students and teachers. This week long course introduces parallel programming, assembles a small cluster, and introduces students to application areas that depend on HPC.

6 Results from Prior NSF Support

PI, Hartmut Kaiser is the PI of the NSF funded projects **APX: Accelerated ParalleX for Enhanced Scaling AMR based Science** (NSF grant 1117470, $550,000, 9/1/11 – 8/31/14), **STAR: a Scalable toolkit for Transformative Astrophysics Research** (NSF grant 1240655, $550,000, 9/1/12 – 8/31/17), and **STORM: a Scalable Toolkit for an Open community supporting near Real-time high resolution coastal Modeling** (NSF grant 1339782, $2,999,894, 10/1/2014 – 9/30/2018). All those projects focus on different aspects of expanding HPX and its use for scientific applications. APX focuses on improving the class of adaptive mesh refinement applications. The STAR project applies HPX to a class of astrophysics simulations of binary white dwarf systems. The STORM project applies HPX to ADCIRC, a storm surge modeling and forecasting tool. *Intellectual Merit:* All projects focus on the interrelationship of parallelism and overhead, ultimately determining the practical range of attainable scalability. They include immediate impacts in advancing the specific science domain of numerical relativity and more broadly in those science and engineering disciplines relying on both AMR and strong scaling. They broadly impact many problems in physics and engineering which require the simultaneous solution of coupled systems of equations arising from different physical processes which are typically governed by equations of various types (hyperbolic, elliptical, and parabolic), and which require different discretization and numeric strategies for their solution, so-called multi-physics modeling. Publications from those projects include [85, 86, 87, 13, 88] *Broader Impact:* The research results and resources have been applied to the distance-learning course distributed live to other national and international campuses to expand its content and extend its advanced topics section, this in the short term, while motivating a new graduate level seminar course next year around its topic areas.

Co-PI Steven R. Brandt is funded by NSF for the project **Using PDE Descriptions To Generate Code Precisely Tailored to Energy-Constrained Systems Including Large GPU Accelerated Clusters** (NSF grant 1265449, $169,999 9/2013-8/2017). The goal of this work is to implement a framework which reads a PDE description of a system written in a domain-specific language (DSL) and from that generates tuned code targeting computing clusters with accelerated nodes. The project looked at a number of issues, including energy, multiple discretization methods, and hybrid execution. Chemora (in prototype form) is being developed under this project. One publication on the performance model is in print, [89] and two are under review. An early overview of Chemora appeared in [90], and more recent work was presented at the Einstein Toolkit Workshop [91]. There is a Chemora project page [92] and a publicly accessible repository at https://bitbucket.org/chemora/chemoracode.

Co-PI Allen Malony received funding through **SI2-SSI:Collaborative Research: A Glass Box Approach to Enabling Open, Deep Interactions in the HPC Toolchain**; Grant No. OCI-1148346, $926,667; K. Schwan (PI, GT), A. Malony (co-PI, UO), B. Chapman (co-PI, UH); 6/1/12 – 5/31/15. *Intellectual merit:* Build integrated HPC development tools by exposing information between layers of the software stack, including compiler, runtime system, and application performance measurement. *Broader impact:* Results are being incorporated in the TAU Performance System and the OpenUH compiler framework and distributed for broad use. The project contributed to the OpenMP Tools (OMPT) interface extensions to the pending OpenMP 5.0 standard and implemented a dynamic OpenMP runtime framework with APEX. *Publications:* [93, 94, 95, 96] *Software artifacts:* http://tau.uoregon.edu, https://github.com/khuck/xpress-apex. **SI2-SSI. Collaborative Research: A Software Infrastructure for MPI Performance Engineering: Integrating MVAPICH and TAU via the MPI Tools Interface**, Grant No: OCI-1450471. D. Panda (PI, OSU), S. Shende (co-PI, UO), A. Malony (co-PI, UO); *Amount: $1,200,000 subaward, 9/1/15 – 8/31/19. Intellectual merit:* Shares performance information between the MVAPICH2 runtime system and the TAU Performance System to create an adaptive runtime using the MPI-T interface. *Broader impact:* Develops MVAPICH and TAU further. Results will be shared through SC and XSEDE conference tutorials and MVAPICH community meetings. *Publications:* none to date. *Software artifacts:* http://tau.uoregon.edu

References

[1] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[2] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, December 2014.

[3] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[4] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pages 265–283. USENIX Association, 2016.

[6] Martín Abadi. Tensorflow: Learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 1–1, New York, NY, USA, 2016. ACM.

[7] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. Spartan: A distributed array framework with smart tiling. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 1–15, 2015.

[8] Alex Gittens. Scientific matrix factorizations in spark at scale. https://amplab.cs.berkeley.edu/scientific-matrix-factorizations-in-spark-at-scale/, 2016.

[9] Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z Ibrahim, and Jay Srinivasan. Scaling spark on hpc systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 97–110. ACM, 2016.

[10] T. Heller, H. Kaiser, and K. Iglberger. Application of the ParalleX Execution Model to Stencil-based Problems. In *Proceedings of the International Supercomputing Conference ISC'12, Hamburg, Germany*, 2012.

[11] Thomas Heller, Hartmut Kaiser, Andreas Schäfer, and Dietmar Fey. Using HPX and LibGeoDecomp for Scaling HPC Applications on Heterogeneous Supercomputers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '13, pages 1:1–1:8, New York, NY, USA, 2013. ACM.

[12] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[13] Hartmut Kaiser, Thomas Heller, Daniel Bourgeois, and Dietmar Fey. Higher-level parallelization for local and distributed asynchronous task-based programming. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, ESPM '15, pages 29–37, New York, NY, USA, 2015. ACM.

[14] Hartmut Kaiser, Bryce Adelstein-Lelbach, Thomas Heller, and Agustin Berge et.al. HPX V0.9.99: A general purpose C++ runtime system for parallel and distributed applications of any scale, 2016. http://dx.doi.org/10.5281/zenodo.58027.

[15] Thomas Heller, Hartmut Kaiser, Patrick Diehl, Dietmar Fey, and Marc Alexander Schweitzer. Closing the Performance Gap with Modern C++. In Michaela Taufer, Bernd Mohr, and Julian M. Kunkel, editors, *High Performance Computing*, volume 9945 of *Lecture Notes in Computer Science*, pages 18–31. Springer International Publishing, 2016.

[16] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications. In *Parallel Processing Workshops*, pages 394–401, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[17] Thomas Sterling. ParalleX Execution Model V3.1. 2013.

[18] The C++ Standards Committee. ISO International Standard ISO/IEC 14882:2011, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO)., 2011. http://www.open-std.org/jtc1/sc22/wg21.

[19] The C++ Standards Committee. ISO International Standard ISO/IEC 14882:2014, Programming Language C++. Technical report, Geneva, Switzerland: International Organization for Standardization (ISO)., 2014. http://www.open-std.org/jtc1/sc22/wg21.

[20] Niklas Gustafsson and Artur Laksberg and Herb Sutter and Sana Mithani. N3857: Improvements to std::future¡T¿ and Related APIs. Technical report, The C++ Standards Committee, 2014. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3857.pdf.

[21] N4501: Working Draft: Technical Specification for C++ Extensions for Concurrency. Technical report, , 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4501.html.

[22] N4505: Working Draft: Technical Specification for C++ Extensions for Parallelism. Technical report, , 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4505.pdf.

[23] N4411: Task Block (formerly Task Region) R4. Technical report, , 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf.

[24] N4406: Parallel Algorithms Need Executors. Technical report, , 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4406.pdf.

[25] Intel. Intel Thread Building Blocks 4.4, 2016. http://www.threadingbuildingblocks.org.

[26] Microsoft. Microsoft Parallel Pattern Library, 2010. http://msdn.microsoft.com/en-us/library/dd492418.aspx.

[27] Kevin Huck, Sameer Shende, Allen Malony, Hartmut Kaiser, Allan Porterfield, Rob Fowler, and Ron Brightwell. An Early Prototype of an Autonomic Performance Environment for Exascale. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, page 8. ACM, 2013.

[28] Kevin A. Huck, Allan Porterfield, Nick Chaimov, Allen D. Malony, Robert Fowler, Hartmut Kaiser, and Thomas Sterling. An Autonomic Performance Environment for Exascale. *Journal on Supercomputing Frontiers and Innovations*, 2015. in press.

[29] S. Shende and A. Malony. The TAU Parallel Performance System. *IJHPCA*, 20(2, Summer):287–311, 2006. ACTS Collection Special Issue.

[30] TCE: Tensor Contraction Engine. http://barista.cse.ohio-state.edu/wiki/index.php/Tensor_Contraction_Engine.

[31] Synthesis of high performance algorithms for electronic structure calculations. NSF awards 0121676, 0121706; 9/01–8/07; PI: Sadayappan; one of the co-PIs: Ramanujam.

[32] G. Baumgartner, D. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In *Proc. Supercomputing 2002*, November 2002.

[33] D. Cociorva, G. Baumgartner, C. Lam, P. Sadayappan, J. Ramanujam, M. Nooijen, D. Bernholdt, and R. Harrison. Space-Time Trade-Off Optimization for a Class of Electronic Structure Calculations. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 177–186, 2002.

[34] G. Baumgartner, A. Auer, D. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, C. Lam, Q. Lu, M. Nooijen, R. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov. Synthesis of High-Performance Parallel Programs for a Class of Ab Initio Quantum Chemistry Models. *Proceedings of the IEEE*, 93(2):276–292, 2005. Special issue on "Program Generation, Optimization, and Adaptation".

[35] Qingda Lu, Xiaoyang Gao, Sriram Krishnamoorthy, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan. Empirical performance model-driven data layout optimization and library call selection for tensor contraction expressions. *J. Parallel Distrib. Comput.*, 72(3):338–352, 2012.

[36] Ajay Panyala, Pamela Bhattacharya, Gerald Baumgartner, and J Ramanujam. Model-driven search-based loop fusion optimization for handwritten code. In *Proceedings of the 17th Workshop on Compilers for Parallel Computers*, CPC '13, July 2013.

[37] Prashant Rawat, Martin Kong, Tom Henretty, Justin Holewinski, Kevin Stock, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. SDSLc: A multi-target domain-specific compiler for stencil computations. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, pages 6:1–6:10, New York, NY, USA, 2015. ACM.

[38] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proc. CC 2008 - International Conference on Compiler Construction*, pages 132–146. LNCS, Springer-Verlag, 2008.

[39] Uday Bondhugula, Albert Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI '08)*, 2008.

[40] Muthu Baskaran, Albert Hartono, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized Tiling Revisited. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[41] Albert Hartono, Muthu Manikandan Baskaran, C. Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric Multi-Level Tiling of Imperfectly Nested Loops. In *ACM International Conference on Supercomputing*, 2009.

[42] Daniel Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 296–305, New York, NY, USA, 2001. ACM.

[43] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[44] Michel X. Goemans and David P. Williamson. Approximation algorithms for np-hard problems. chapter The Primal-dual Method for Approximation Algorithms and Its Application to Network Design Problems, pages 144–191. PWS Publishing Co., Boston, MA, USA, 1997.

[45] Jochen Könemann, Stefano Leonardi, Guido Schäfer, and Stefan H. M. van Zwam. From primal-dual to cost shares and back: A stronger lp relaxation for the steiner forest problem. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, volume 3580 of *Lecture Notes in Computer Science*, pages 930–942. Springer International Publishing, 2005.

[46] G.L. Nemhauser and L.A. Wolsey. Best algorithms for approximating the maximum of a submodular set function. *Math. Oper. Research*, 3(3):177–188, 1978.

[47] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999. 530 pp.

[48] Ashwinkumar Badanidiyuru, Baharan Mirzasoleiman, Amin Karbasi, and Andreas Krause. Streaming submodular maximization: Massive data summarization on the fly. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 671–680, New York, NY, USA, 2014. ACM.

[49] Niv Buchbinder, Moran Feldman, Joseph (Seffi) Naor, and Roy Schwartz. Submodular maximization with cardinality constraints. In *Proceedings of the Twenty-fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 1433–1452, Philadelphia, PA, USA, 2014. Society for Industrial and Applied Mathematics.

[50] Sebastian Tschiatschek, Rishabh Iyer, Haochen Wei, and Jeff Bilmes. Learning mixtures of submodular functions for image collection summarization. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 1413–1421, Cambridge, MA, USA, 2014. MIT Press.

[51] Jeffrey Alan Daily. *Gain: Distributed array computation with python*. PhD thesis, Citeseer, 2009.

[52] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[53] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association.

[54] Frank McSherry, Michael Isard, and Derek G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, pages 14–14, Berkeley, CA, USA, 2015. USENIX Association.

[55] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13(2):22–30, March 2011.

[56] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[57] Antoine Tran Tan and Hartmut Kaiser. Extending c++ with co-array semantics. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pages 63–68, New York, NY, USA, 2016. ACM.

[58] The HDF Group. Hierarchical data format version 5, 2000-2010. http://www.hdfgroup.org/HDF5.

[59] Hypertable Inc. C++ Hypertable Library, 2017. http://www.hypertable.com.

[60] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[61] Klaus Iglberger, Georg Hager, Jan Treibig, and Ulrich Rüde. Expression templates revisited: A performance analysis of the current ET methodology. *CoRR*, abs/1104.1729, 2011.

[62] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[63] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Dague: A generic distributed dag engine for high performance computing. *Parallel Comput.*, 38(1-2):37–51, January 2012.

[64] TOP500 supercomputer site.

[65] Vladimir Marjanovic, Jos Gracia, and Colin W. Glass. Performance Modeling of the HPCG Benchmark. In Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond, editors, *PMBS@SC*, volume 8966 of *Lecture Notes in Computer Science*, pages 172–192. Springer, 2014.

[66] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.

[67] Research software by Arthur Asuncion and Irvine colleagues at the University of California. A Fast And Scalable Topic-Modeling Toolbox, 2017. http://www.ics.uci.edu/ asuncion/software/fast.htm.

[68] Ian Porteous, David Newman, Alexander Ihler, Arthur Asuncion, Padhraic Smyth, and Max Welling. Fast collapsed gibbs sampling for latent dirichlet allocation. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 569–577, New York, NY, USA, 2008. ACM.

[69] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems*, NIPS'12, pages 1223–1231, USA, 2012. Curran Associates Inc.

[70] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[71] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. Graphs, matrices, and the graphblas: Seven good reasons. *Procedia Computer Science*, 51:2453–2462, 2015.

[72] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the graphblas. In *High Performance Extreme Computing Conference (HPEC), 2016 IEEE*, pages 1–9. IEEE, 2016.

[73] Vijay Gadepally, Jake Bolewski, Dan Hook, Dylan Hutchison, Benjamin A. Miller, and Jeremy Kepner. Graphulo: Linear algebra graph kernels for nosql databases. *CoRR*, abs/1508.07372, 2015.

[74] Jeremy Kepner, David A. Bader, Aydin Buluç, John R. Gilbert, Timothy G. Mattson, and Henning Meyerhenke. Graphs, matrices, and the graphblas: Seven good reasons. *CoRR*, abs/1504.01039, 2015.

[75] Andreas Krause and Daniel Golovin. Submodular function maximization. To appear in Tractability: Practical Approaches to Hard Problems.

[76] Kai Zhang, Wei Wu, Fang Wang, Ming Zhou, and Zhoujun Li. Learning distributed representations of data in community question answering for question retrieval. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, WSDM '16, pages 533–542, New York, NY, USA, 2016. ACM.

[77] Hui Lin and Jeff Bilmes. Multi-document summarization via budgeted maximization of submodular functions. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, HLT '10, pages 912–920, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics.

[78] Petr Hliněný. Branch-width, parse trees, and monadic second-order logic for matroids. *Journal of Combinatorial Theory, Series B*, 96(3):325 – 351, 2006.

[79] Jim Geelen, Bert Gerards, and Geoff Whittle. Tangles, tree-decompositions and grids in matroids. *Journal of Combinatorial Theory, Series B*, 99(4):657 – 667, 2009.

[80] J. Carmesin, R. Diestel, M. Hamann, and F. Hundertmark. Canonical tree-decompositions of finite graphs i. existence and algorithms. *Journal of Combinatorial Theory, Series B*, 116:1 – 24, 2016.

[81] R. Diestel and G. Whittle. Tangles and the Mona Lisa. *ArXiv e-prints*, March 2016.

[82] Gerth Stølting Brodal. *Cache-Oblivious Algorithms and Data Structures*, pages 3–13. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[83] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[84] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4:1–4:22, January 2012.

[85] Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, Bryce Adelstein-Lelbach, and Thomas L. Sterling. Adaptive mesh refinement for astrophysics applications with ParalleX. *CoRR*, abs/1110.1131, 2011. http://arxiv.org/abs/1110.1131.

[86] Matthew Anderson, Maciej Brodowicz, Hartmut Kaiser, Bryce Adelstein-Lelbach, and Thomas L. Sterling. Neutron star evolutions using tabulated equations of state with a new execution model. *CoRR*, abs/1205.5055, 2012.

[87] Matthew Anderson, M. Brodowicz, Hartmut Kaiser, Bryce Adelstein-Lelbach, and Thomas Sterling. Tabulated equations of state with a many-tasking execution model. *Workshop on Large-Scale Parallel Processing*, 2013.

[88] Zachary D. Byerly, Hartmut Kaiser, Steven Brus, and Andreas Schäfer. A non-intrusive technique for interfacing legacy fortran codes with modern C++ runtime systems. In *Third International Symposium on Computing and Networking, CANDAR 2015, Sapporo, Hokkaido, Japan, December 8-11, 2015*, pages 503–507. IEEE Computer Society, 2015.

[89] Yue Hu, David M Koppelman, Steven R Brandt, and Frank Löffler. Model-driven auto-tuning of stencil computations on GPUs. In *Proceedings of the 2nd International Workshop on High-Performance Stencil Computations*, 2015.

[90] E. Schnetter, M. Blazewicz, S. R. Brandt, D. M. Koppelman, and F. Lffler. Chemora: A PDE-solving framework for modern high-performance computing architectures. *Computing in Science Engineering*, 17(2):53–64, Mar 2015.

[91] Steven R. Brandt, David M. Koppelman, and Yue Hu. Chemora kernel mapping optimization, August 2015.

[92] Chemora. http://chemoracode.org/.

[93] Allen D Malony and Kevin A Huck. General Hybrid Parallel Profiling. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 204–212. IEEE, 2014.

[94] Kevin A Huck, Allen D Malony, Sameer Shende, and Doug W Jacobsen. Integrated Measurement for Cross-Platform OpenMP Performance Analysis. In *IWOMP 2014: Using and Improving OpenMP for Devices, Tasks, and More*, pages 146–160. Springer International Publishing, 2014.

[95] Ahmad Qawasmeh, Abid Malik, Barbara Chapman, Kevin Huck, and Allen Malony. Open Source Task Profiling by Extending the OpenMP Runtime API. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 186–199. Springer Berlin Heidelberg, 2013.

[96] Md Abdullah Shahneous Bari, Nicholas Chaimov, Abid M. Malik, Kevin A. Huck, Barbara Chapman, Allen D. Malony, and Osman Sarood. ARCS: Adaptive Runtime Configuration Selection for Power-Constrained OpenMP Applications. In *2016 IEEE International Conference on Cluster Computing*, pages 461–470, 2016.