SANDIA REPORT

SAND2015-1379
Unlimited Release
Printed December, 2014

Algorithms and Abstractions for Assembly in PDE Codes: Workshop Report

Eric C. Cyr, Eric Phipps, Michael A. Heroux, Jed Brown, Ethan T. Coon, Mark Hoemmen, Robert C. Kirby, Tzanio V. Kolev, James C. Sutherland and Christian R. Trott

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2015-1379
Unlimited Release
Printed December, 2014

Algorithms and Abstractions for Assembly in PDE Codes: Workshop Report

Eric C. Cyr, Eric Phipps,
Michael A. Heroux, Mark Hoemmen,
Christian R. Trott
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-9999

Jed Brown
Argonne National Laboratory
9700 S Cass, Bldg 240
Lemont, IL 60439

Ethan T. Coon
Los Alamos National Laboratory
P.O. Box 1663
Los Alamos, NM 87545

Robert C. Kirby
Baylor University
One Bear Place
Waco, TX 76798-7328

Tzanio V. Kolev
Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA 94550

James C. Sutherland
The University of Utah
3290 MEB
50 S Central Campus Dr
Salt Lake City, Ut 84112

Abstract

The emergence of high-concurrency architectures offering unprecedented performance has brought many high-performance partial differential equation (PDE) discretization codes to the precipice of a major refactor. To help address this challenge a workshop titled "Algorithms and Abstractions for Assembly in PDE Codes" was held in the Computer Science Research Institute at Sandia National Laboratories on May 12th-14th, 2014. This document summarizes the goals of the workshop and the results of the presentations and subsequent discussions.

Contents

Appendix

Overview and Themes

The workshop for "Algorithms and Abstractions for Assembly in PDE Codes" was held in the Computer Science Research Institute at Sandia National Laboratories on May 12th-14th, 2014. There were 27 scheduled speakers (one of those scheduled canceled due to weather interrupting travel). Three speakers, Martin Berzins (University of Utah), Michael Heroux (Sandia National Laboratories), and Paul Fischer (Argonne National Laboratory), gave hour-long "keynote" talks. The remaining speakers gave a brief 15-minute talk introducing a poster that was presented during an extended poster session at the end of each day. For a list of poster presenters, talk titles and abstracts see the program included in Appendix A. The format for the workshop was chosen to facilitate in depth conversation and discussion.

This document summarizes the goals of the workshop and the results of the presentations and subsequent discussions. The content included is intended to represent what was presented at the workshop. As such it should not be viewed as a comprehensive survey, but a representative sampling of the state-of-the-art. The first section briefly describes the motivation and reasons for the workshop. The next section assimilates themes and conclusions from the workshop while the remaining sections are dedicated to topic areas that were the focus of workshop speakers. While an effort has been made to decompose the subjects into independent areas, there is substantial overlap between them and this has been noted in the text where relevant. References are included so an interested reader can follow up on topics. Finally, a full list of workshop participants and the program is included in Appendix A.

Goals of the Workshop

The emergence of high-concurrency architectures offering unprecedented performance has brought many high-performance partial differential equation (PDE) discretization codes to the precipice of a major refactor. Yet uncertainty in the specifics of next-generation architectures and programming models makes any rewrite a high-risk, high-cost undertaking. A gross oversimplification of many (implicit) PDE solvers segregates the code into two compute intensive parts to be refactored:

1. Computation of matrix/vector coefficients and assembly into linear algebra data structures.

2. Numerical linear algebra including linear solvers.

For explicit solvers, the matrix assembly is not required, only assembly of the a right-hand-side is necessary.

Much work has been, and continues to be, dedicated to the problem of numerical linear algebra. This can be neatly abstracted within the PDE code. On the other hand, the first part–defining the "assembly" algorithm for this workshop and report–encapsulates much of

the physics of the code. As a result, it is often application specific, and thus may require substantial effort to refactor. Moreover, while the performance of the linear algebra is clearly critical to the overall performance of the code, the performance of the assembly is often dismissed as inconsequential compared to the expense of linear solves. However, a poorly designed abstraction or a careless implementation can result in severe performance penalties, especially in explicit and operator split codes where linear solves are not as costly. Even for implicit codes, where assembly has not typically been a major cost when compared to other phases, the inability of assembly to be thread-scalable is a critical concern on new architectures. Furthermore, different algorithms offer different trade-offs in floating-point performance and memory requirements between matrix assembly, evaluation and linear solve. These trade-offs are worth re-examining on current and future hardware. A goal of this workshop was to provide a forum to exchange ideas about successful abstractions and approaches for assembly on heterogeneous multi-core architectures.

While the need to refactor PDE codes to run on a myriad of proposed next-generation architectures is certainly the inciting event, this workshop took the opportunity to review abstractions that address the broader needs of the assembly process. Topics included handling of the complexity associated with multiphysics assembly, interfaces for linear algebra, and PDE discretization approaches. A final desirable outcome of this workshop was to grow and strengthen an interactive community surrounding the needs of assembly. While success in this regard is a question best left to the future, the energy and interest from the lead up to the meeting and throughout the workshop was clear.

Major Themes and Conclusions

Before diving into the techniques and ideas discussed by the workshop speakers, we first provide a short synopsis of the major themes and conclusions arising from the workshop presentations and discussions. First and foremost, the workshop highlighted substantial challenges that must be overcome for achieving high performance of PDE assembly calculations on emerging multicore and manycore architectures. Indeed, these architectures are expected to present hierarchical parallel execution and memory spaces that must be leveraged and managed for optimal execution efficiency. Thus PDE assembly codes must expose multiple levels of parallelism operating on appropriately sized datasets that can be mapped to architecture-specific execution and memory spaces. Uncertainty in the evolution of these architectures, as well as the large space of potential parallelization strategies for assembly, makes designing future PDE codes extremely challenging. Additionally, several other conclusions arose from the workshop presentations and discussions:

1. Race conditions must be handled effectively for thread-scalable performance, and the use of hardware atomic instructions is a compelling approach that preserves the simplicity of traditional serial assembly procedures.

2. The use of directed-acyclic graphs, whether explicitly formed or implicitly used via programming concepts such as futures or inherent data dependencies, is valuable as a

natural means for exposing multiple levels of parallelism, handling resiliency, and is an intriguing way of managing complexity in multiscale and multiphysics simulations.

3. A wide variety of powerful compute node-level parallel programming models are already available, which emphasize exposing fine-grained parallelism by the application that is then mapped to hardware resources. However significant uncertainty as to how to expose memory hierarchies remains.

4. Linear algebra interfaces for PDE assembly must expose fine-grained parallelism in order to be adapted to a variety of parallel programming models and maintain thread scalability. The count-allocate-fill-compute paradigm is a powerful means for designing these interfaces.

5. There is significant uncertainty in how to best expose multiple levels of parallelism in PDE assembly calculations and how to design discretization schemes that are more amenable to fine-grained parallelism. In particular, the workshop discussed substantial challenges in exploiting higher-order discretizations and block arithmetic approaches.

Topic Areas

This section provides an overview of the major topics included in the workshop.Where appropriate the relevant presenters have been cited using the annotation [*presenter name*].

Assembly in PDE Solvers

The goal of assembly in PDE solvers is to construct an algebraic system that is satisfied by a simple update (as in explicit methods) or requires the solution to possibly multiple linear systems (as in implicit or direct-to-steady state methods). There is a range of other time discretization approaches, for instance semi-implicit and operator split; however, the fundamentals of assembly for these methods are shared with explicit and implicit time integration methods.

The choice of spatial discretization also affects the structure of the assembly. For instance, in finite difference approaches, loops are typically over the nodes of the mesh, in finite volumes they may be over the nodes or faces/edges, and in finite elements, typically loops are over mesh cells. The choice of loop iterate (nodes, edges, cells) may have a large impact on the degree of parallelism in the assembly. For instance in a straightforward finite difference code each row of the algebraic system can be assembled simultaneously. However, in finite elements, as we will see, there is a race condition that must be resolved.

In finite element methods, the physical domain is tiled by cells defining the topology of the mesh. On each cell a set of basis functions is associated with every field in the PDE solution. The goal of finite element methods then is to determine the coefficients, called degrees of freedom in this document, for each basis function. These degrees of freedom satisfy an algebraic system of equations. For implicit time stepping schemes or direct to steady-state methods, the solution to this system is computed by solving a single (for linear problems) or a sequence of (for nonlinear problems) linear systems. Assembly of the linear systems requires computing local integrals over each cell of a mesh. If the matrix is to be explicitly formed, these contributions are summed into a global data structure (like compressed sparse row). For matrix free operations these contributions may be saved; however, it may be more efficient to compute them on the fly as needed to compute the action of the matrix.

These independent local integral calculations offer a prime opportunity for parallelism, as they are embarrassingly parallel. However, when summing into global data structures, degrees of freedom shared between adjacent cells, say associated to a vertex or face, create the potential for a race condition. This is because threads associated with the different cells may write to the same memory location. See Figure 1 for a pseudo-code description of this style of assembly procedure.

```
Import off-processor degrees-of-freedom
Loop over mesh cells (thread-parallel)
    Load degrees-of-freedom corresponding to mesh cell into local data structures
    Evaluate local residual/Jacobian contributions for mesh cell
    Add local contributions to global vector/matrix data structures (race condition)
Export vector/matrix contributions across processors (if necessary)
```

Figure 1. Pseudo code finite element assembly loop.

Resolving the Race Condition

Several possibilities exist to resolve the race condition, and four received considerable attention at the workshop. These are based on mesh coloring ([Turcksin] in Deal.II), a gather-sum technique and atomic operations, both presented by [Edwards] and [Sunderland], and a final technique based on the use of parallel sort and segmented scan presented by [Kirby].

Coloring works by creating a graph among the mesh cells such that two cells are adjacent if they share a degree of freedom (As a brief comment, this graph depends not only on the mesh connectivity but also on the particular finite element discretization since not all finite element spaces have vertex or edge degrees of freedom.). Each cell is assigned a color so that no two adjacent cells receive the same color. Then, provided that a suitable data structure for the global matrix has been allocated, all elementwise contributions may be computed and assembled concurrently. The graph coloring, performed as a pre-processing step, should produce as few colors as possible, and have a relatively balanced number of cells per color to ensure best concurrency. Finding the optimal coloring is an NP complete problem, therefore a fast polynomial time algorithm is not known. Fortunately, having the optimal coloring is not necessary to enable good shared-memory performance, thus fast approximate coloring algorithms can be used.

[Edwards] and [Sunderland] presented another approach to resolving contention. Rather than requiring the coloring analysis step, they use atomic operations to sum contributions into the global structure. Atomic operations ensure that a memory operation (like writing) must complete before another operation can occur at that same location in memory. This guarantees the system is in some well determined state. This leaves the standard flow of code unchanged (build a matrix, sum it) over a serial code, except for the addition of the atomics. However, the extra synchronization incurs some amount of overhead. This overhead depends on whether atomic operations exist in hardware or software. A possibly more insidious issue is the loss of bitwise floating point repeatability because of variation in thread execution order.

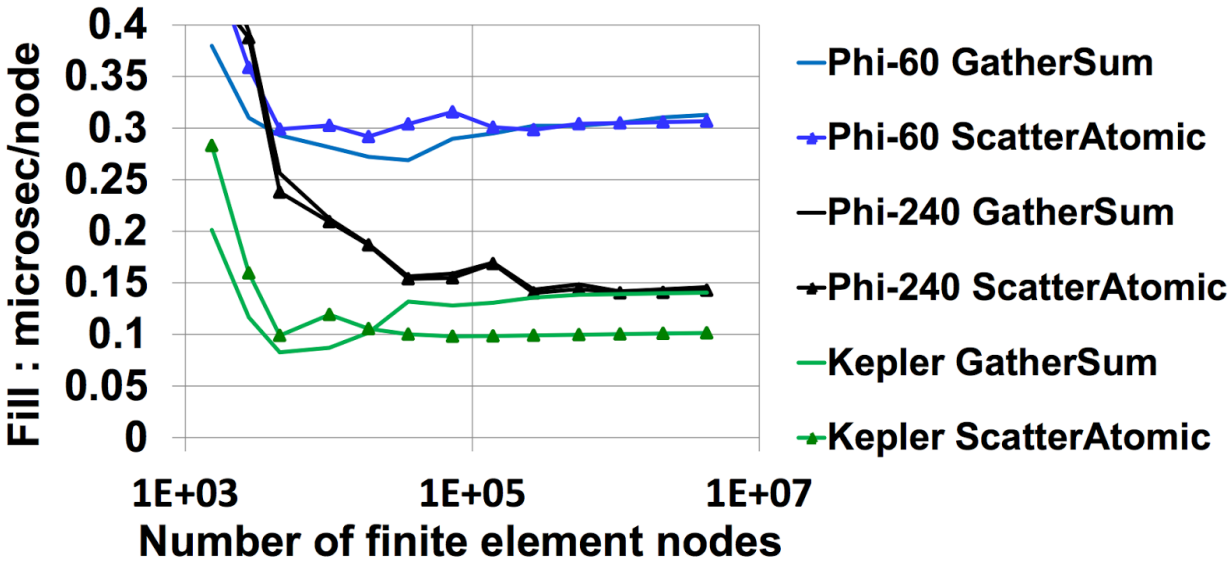

Figure 2. The performance of matrix fill using the gather-sum and atomic approaches for different architectures. Here Phi-60 refers to the Intel Xeon Phi accelerator using 60 cores and one thread per core, Phi-240 refers to the same architecture using four threads per core, and Kepler refers to the Nvidia K20X GPU.

To circumvent the overhead of atomics, [Edwards] and [Sunderland] also considered an approach based on "gather-sum." The typical local-to-global mapping assigns to each cell a list of global degrees of freedom. This data structure can be transposed to find, for each global degree of freedom, a list of all local contributions required to form it. Then, a first computational phase forms the local matrices or vectors for each cell. A second phase, parallel over global degrees of freedom, uses the transposed data structure to gather local contributions. Since the gather phase requires access to all element matrices, this approach uses a relatively large amount of local memory. This local storage could perhaps be reduced by partitioning the domain into chunks still large enough to enable full concurrency but small enough to limit temporary storage.

[Kirby] presented a similar approach, based on standard parallel primitives, that resolves write conflicts without atomics or coloring. The algorithm begins by forming a batch of local matrices or vectors in parallel. Across this batch, the element value is paired with the global index that it contributes to. This list of pairs is sorted (via a parallel radix sort) by the global index. Entries contributing to the same global value are adjacent in the resulting list. A segmented scan, where change in global index indicates a segment boundary, then gathers element contributions together. When performed for matrices as opposed to vectors, this produces a coordinate-format matrix that must then be converted into the matrix-format required by the solver, for instance compressed sparse row (CSR). This approach shares with the gather-sum technique a possibly large temporary storage requirement. A further caveat is that not all shared memory environments allow for user-defined segmented scans.

Of the four presented approaches, the coloring and atomic approaches both maintain the original serial assembly loop. This is achieved by avoiding the race condition in the coloring case and resolving conflicts for atomics. The gather-sum and segmented scan approaches avoid the race condition by changing the domain of parallelization (from mesh cells to matrix entries, for instance). This requires additional data structures and a modification of the serial code. The approach that performs the best depends on the hardware/language/software environment. Results presented by [Edwards] shows the performance of the matrix fill using the gather-sum and atomics approaches (see Figure 2). With the exception of the Kepler GPU architecture the gather-sum and atomic cases are nearly equivalent.

Local Assembly Abstractions

Another important aspect of PDE assembly discussed in the workshop was the efficient evaluation of derivative matrices needed for instance in Newton solvers, sensitivity analysis methods, and optimization methods. Often the code required to evaluate these derivatives is hand-coded based on symbolic differentiation of the PDE discretization. [Pawlowski] and [Stogner] presented two similar but alternative approaches based on automatic differentiation that allows these matrices to be efficiently evaluated using only code written for the discrete PDE residual evaluation. The methodologies, using template-based generic programming [24], involves templating the PDE assembly code on the scalar type and then instantiating this code on derivative scalar types. After suitable initialization of the deriva-

tive scalars associated with PDE degrees-of-freedom, evaluation of the derivative matrix occurs as a side-effect of the residual code evaluation. This approach can be extended to non-derivative-based evaluations, such as the ensemble evaluations for forward uncertainty propagation discussed later.

An alternative to template-based generic programming called Nebo was presented by [Sutherland]. Nebo is a domain-specific language embedded in C++ that allows high-level vectorized operations (MATLAB-style). Nebo provides a portable abstraction for compile-time generation of binary code (through C++) targeting various architectures including serial CPU, multicore CPU and GPU. As a simple example, evaluation of a diffusion operator, $-\frac{\partial}{\partial x}\Gamma\frac{\partial}{\partial x}\phi - \frac{\partial}{\partial y}\Gamma\frac{\partial}{\partial y}\phi - \frac{\partial}{\partial z}\Gamma\frac{\partial}{\partial z}\phi$ can be written in Nebo using standard C++:

```
diffTerm <<= DivX( - InterpX(gamma) * GradX(phi) )
           + DivY( - InterpY(gamma) * GradY(phi) )
           + DivZ( - InterpZ(gamma) * GradZ(phi) );
```

where the `<<=` operator triggers Nebo's expression template engine. All fields and operators in Nebo are strongly typed, providing compile-time robustness, and type traits on fields and operators allow sophisticated type inference to enable highly generic programming that works across a range of discretizations. Nebo deploys assignment operations on the device where the destination field (`diffTerm` in the above example) is active, and fields can simultaneously have multiple field locations (CPU, GPU 1, GPU 2, ...) available. Nebo internally keeps track of which locations are valid, and provides APIs for explicitly managing memory transfers (synchronous or asynchronous) between devices. Nebo also provides the notion of a mask, where a subset of points in a field can be subject to an operation. This abstraction is useful for application of boundary conditions. Note that both the template-based generic programming and DSL approaches can be readily combined with the directed acyclic graph approaches described below.

Directed Acyclic Graphs

The previous section assumed that the elementwise stiffness matrices and load vectors were precomputed (the "evaluate local residual/Jacobian" steps in Figure 1). This simple presentation hides much of what implements a particular PDE. This can in itself be quite complex and offer new opportunities for parallelism. To address the complexity and simultaneously exploit new levels of parallelism software architectures based on directed acyclic graphs (DAGs) have been developed exposing the structure of the calculation [16, 23, 26]. The basic abstraction involves a task that expresses its dependencies (requirements) and the quantities that it produces/computes. This topic received considerable focus during the workshop with the [Berzins] keynote dedicated to the topic, as well as posters by [Pawlowski], [Sutherland], [Demeshko], [Bennett], [Andrs] and [Moulton].

The DAG approach provides a number of noteworthy opportunities to abstract much of the tedious aspects of traditional programming approaches. Specifically,

- Task-level parallelization can be easily accomplished because the data dependencies are explicitly known by the graph.

- Communication and computation can be naturally overlapped. This applies to intra-node (e.g., host-device or device-device transfers) as well as inter-node (e.g. MPI) transfers. Importantly for multiphysics applications, as complexity increases, there is more opportunity for DAG schedulers to overlap communication and computation.

- Memory can be dynamically reclaimed or reused to minimize memory footprint. This can be automated by the graph scheduling algorithm that can determine when a field is no longer required and can be released.

- Complex dependencies that typically result in a difficult to follow "cascading if" can be handled automatically. Moreover each task declares its dependencies and products locally thus enhancing modularity.

- Lines of code written by domain experts who are implementing the discretizations are simpler to write because task parallelism is primarily managed outside the scope of execution of this code.

As we look toward emerging architectures, DAG-based approaches provide key flexibility to overlap communication and computation as well as naturally expose task-level parallelism. This can be directly exploited on existing multicore architectures as well as recent GPU architectures. Additionally, data-level parallelism can be incorporated both within a task (via multithreaded loops or CUDA-type kernel calls) or above tasks via domain decomposition. Indeed, the DAG frameworks Uintah and Charm++ exploit coarse task-level parallelism as well as data parallelism within a DAG representation.

Successful use of task-based parallelism requires over-decomposition. If each processor is dedicated to a single task there will not be an opportunity to backfill those operations. The granularity of each task is still an ongoing point of discussion. Some efforts focus on over-decomposition in the extreme, while others take a more measured approach. The notion of over-decomposition is limited, however; at the strong scaling extreme where the full degree of parallelism is exposed and allocated, over-decomposition is of no use. In this context the overhead of using a runtime task-based system may be prohibitively expensive.

Finally, an interesting aspect of DAG-centric designs is that they have a natural resilience to faults. [Bennett] argued that a task-based approach characterized both by transaction semantics and assumed resilient collectives can lead to a runtime that is robust to failures. If a fault is detected, the DAG scheduler can determine what task(s) must be re-executed to recover from the fault. This can be done at the task-granularity level rather than the level of a time integrator, for example.

Basic Elements of a DAG Abstraction

A task-based software architecture implementation requires a number of components. Two of particular interest are the scheduler and an abstraction for user introduced tasks. The scheduler is fundamental to any successful DAG-based implementation. Schedulers analyze the DAG structure to determine when communication, computation, memory (de)allocation, etc. occur. They are key to providing efficient, scalable software [2].

DAG abstractions fundamentally rely on the description of a task, which support the following:

- Provide a description of the quantities that it computes
- Provide a description of the quantities it requires
- Perform the calculation

Given the computes/requires information, together with a "root node" (e.g., the quantity/quantities desired), a DAG can be automatically constructed by recursing through the dependencies until terminal nodes (with no requirements) are encountered.

Hierarchical DAGs

The Wasatch component within the Uintah computational framework is a multiphysics PDE solver that employs a hierarchical DAG. The Uintah framework is built on DAG technology to describe "coarse" tasks and handles MPI, I/O, etc. Each vertex in the Uintah-level DAG is represented as a full DAG in Wasatch which can handle on-node computations and memory management. This abstraction allows four levels of parallelism (two data parallel and two task parallel levels) at different granularity.

Resources are typically managed by the coarsest level of parallelism and then "pushed down" as necessary. For example, if the coarse level task and data parallelism is insufficient to saturate the available resources, then resources can be pushed down to the fine-level DAG scheduler that can use them there. In the context of CUDA, for example, each task has a stream associated with it to allow both task- and data-parallel execution models simultaneously on GPUs.

Node-Level Abstractions

With the proliferation of novel node-level architectures, developing programming models and abstractions for exploiting the new features was an important aspect of the assembly workshop. The keynote by [Heroux] discussed how fine-grain functionals can be used as a mechanism for exposing parallelism. However, the talk also cautions that the ideal for

domain scientists to "write no parallel code" is ultimately unachievable. Thus, hiding architectural details from applications programmers presents an important challenge to developing performant assembly code. With rapidly evolving architectures and device languages, it is important to decrease the time it takes for applications developers to put algorithms on new machines and also increase the shelf life of these implementations.

The two main requirements for parallelization interfaces identified at the workshop are parallel dispatch and an abstraction layer to handle more complex memory systems. In particular it is necessary to be able to submit work to heavy CPU threads that use vectorization as well as to lightweight GPU threads. A common technique across different models to facilitate this is to provide state-less loop bodies in separate functions or constructs. Constructs include functors (C++ classes that behave like functions with state), lambdas (C++11 anonymous functions), or code provided as the argument to a C preprocessor macro. These loop bodies are then handed to a mechanism that loops over the index range in a fashion appropriate for the targeted device. These mechanisms have the side effect of requiring the algorithm developer to identify the finest degree of parallelism available to achieve portable performance.

Abstraction layers for the memory system can provide a number of key capabilities:

1. Handle multiple memory spaces (e.g., GPU memory, host memory, or nonvolatile storage),
2. Provide data layout abstractions,
3. Give access to atomics,
4. Expose special hardware capabilities (e.g., texture fetches, nontemporal loads, or huge pages).

The ability to manage multiple memory spaces is expected to be necessary for many future HPC platforms. Similar to today's GPU-based machines, future HPC platforms are expected to have at least two memory spaces:

- A smaller but faster space, using stacked memory technologies such as High-Bandwidth Memory (HBM) [12] or Micro's Hybrid Memory Cube (HMC) [20]
- A larger but slower memory space, with much larger capacity

We expect multiple memory spaces, even if the hardware has only one execution space. For example, systems based on the next-generation Intel Xeon Phi (codename KNL) can be deployed as "self hosted," meaning that the application runs on the Phi directly, not treating it as an accelerator. Nevertheless, these systems will have access both to faster stacked memory (16 GB [22]) and to slower but larger DDR memory. Further complicating the situation, byte addressable non-volatile memory might be added to future machines,

which can serve to hold huge databases or allow fast check-pointing for resilience purposes. While in theory this can be handled by a hardware caching mechanism, a cache mode could have large performance and/or energy penalties.

Data layout abstractions can help with enabling optimal data access patterns on different devices. For example, consecutive threads in a GPU prefer contiguously stored array entries for coalesced loads, while threads in a CPU want to work on far away parts of an array in order not to share cache lines.

Atomics are important to handle write conflicts in massively threaded algorithms. An abstraction layer needs to map generic atomics to the available hardware capabilities. For example some architectures provide only certain integer type atomics, while others have selected atomics for floating point types. Other architectures may have transactional memory (e.g. IBM BG/Q and Intel Haswell) available. Data types that cannot be handled natively (e.g. complex numbers, automatic differentiation types, etc.) must be addressed in software.

Special hardware capabilities can provide significant performance improvements when used appropriately. Texture fetches on GPUs give up to 6 times higher bandwidth for localized random access, while using huge pages for certain allocations on Xeon Phi can reduce the number of extremely costly page faults significantly [27]. Some hardware also exposes special load paths for non temporal access such as streaming loads and stores. An abstraction layer needs to provide a generic method to map certain type of accesses to those capabilities. While it might be feasible for future compilers to automatically insert such special memory access operations and allocators, it is unlikely that it would work in complex situations.

At the workshop a number of language-level tools were presented including:

- Kokkos (see [7], [Edwards], [Sunderland], [Trott]): C++ interface/library abstracted over the shared-memory parallel programming model. One programmer provides an architectures specific backend, and templates allow users to program in terms of parallel recipes over this. Kokkos also provides an extensive data and memory system abstraction layer.
- RAJA: C++ interface/library which provides an C++11 (Lambda) based interface for writing parallel kernels. RAJA uses execution policies to map different loop structures optimally to underlying hardware parallelism such as threads and vectors.
- PyOpenCL/Loopy: Tied to a particular, but portable, device language, wraps into a higher-level environment. Loopy exposes transformations intended to streamline device-specific operations.
- OCCA (see [19], [Warburton]): a lightweight abstraction of common features of different devices/languages, allows users to write loop nests and transforms these into device code by means of C macros.
- SpatialOps/Nebo (see [25], [Sutherland]): a domain-specific language embedded in

C++ that supports vectorized operations on fields including stencil operations. This supports deployment on single- and multi-core CPU as well as GPU (via CUDA).

All discussed abstraction layers are able to map work to shared memory parallel architectures through parallel-for/reduce/scan type algorithms. Kokkos, Occa and Loopy in particular are able to target both GPUs and CPUs. RAJA's reliance on C++11 means that it currently cannot run on any GPU platforms. We expect C++11 support to start appearing on GPUs within the next year or so. Furthermore Kokkos, Occa and Loopy allow and require some kind of explicit memory space management. RAJA's strategy is to rely completely on hardware caching mechanisms. An atomic access abstraction is provided by Kokkos and PyOpenCL/Loopy. Further data abstraction concepts recognizing hardware memory hierarchies are only included in Kokkos at this time. The following illustrates a simple nested parallelism use cases for Kokkos, RAJA and OCCA:

Plain C:

```
void matrix_add (int n, int m, double* C, double a,
                            double* A, double b, double* B) {
  for ( int i = 0; i < n; ++i )  {
    for ( int j = 0; j < m; ++j )  {
      C[i*m + j] = a * A[i*m + j] + b * B[i*m + j] ;
    }
  }
}
```

RAJA:

```
void matrix_add (int n, int m, Real_ptr C, double a,
                            Real_ptr A, double b, Real_ptr B) {
  forall< exec_policy >( n,m, [&] (Index_type k)  {
     C[k.i*m + k.j] = a * A[k.i*m + k.j] + b * B[k.i*m + k.j] ;
  });
}
```

Kokkos:

```
void matrix_add (View<double**> C, double a,
                  View<double**> A, double b, View<double**> B) {
  parallel_for(TeamPolicy<>(C.dimension_0(),4),
    [=] (TeamPolicy<>::member_type team_member) {
      const int i = team_member.league_rank();
      team_member.team_par_for( C.dimension_1(), [&] (const int& j)
      { C(i,j) = a * A(i,j) + b * B(i,j) ; }
      );
  });
}
```

OCCA:

```
occaKernel void matrix_add(occaKernelInfoArg, occaPointer double* C,
                           double a, occaPointer double* A, double b,
                           occaPointer double* B) {
  occaOuterFor0 {
    occaInnerFor0 {
      const int i = occaGlobalId0;
      const int j = occaLocalId0;
      C[i*m + j] = a * A[i*m + j] + b * B[i*m + j] ;
    }
  }
}
```

RAJA and Kokkos use C++11 capabilities for this example, while OCCA requires a special preprocessor. RAJA's "Real_ptr" is used to encapsulate attributes such as `__restrict`. The Kokkos example uses default execution and memory spaces which also imply a default data layout which could be different for different hardware architectures (i.e. row major vs. column major storage). Note that in order to run on GPUs the Kokkos example would need to be implemented using a functor which replaces the Lambda. This will change starting with CUDA 6.5 where the use of C++11 lambdas is allowed for nested parallelism. The OCCA example requires a special preprocessor, which would replace the `occaOuterFor0` and `occaInnerFor0` with the respective parallel dispatch using information encapsulated in `occaKernelInfoArg`.

Linear Algebra Data Structures and Interfaces

This section discusses the data structures and interfaces required for assembly of the linear system (both matrices and vectors) into a globally distributed data structure. The applications of interest to most workshop participants use two kinds of matrices: large and sparse, and small and dense. In this context large means "large enough that that a sparse representation should be used," which includes a size range anywhere from "fits in a big cache" to "requires 64-bit integers to express the dimensions." Small and dense matrices tend to express local discretizations of partial differential equations, like finite elements. Small here means"fits in registers or in the smallest, fastest cache." Block sparse matrices combine both size ranges: they are large and sparse, but with small dense entries.

Small dense matrices

Small dense matrices occur often in discretizations of partial differential equations. They even show up in discretizations that never assemble into a large sparse matrix. This is either because they work with the unassembled representation, as in an unassembled multigrid or multifrontal factorization, or because they do not need to solve linear systems, as in

explicit codes (see [Franko]). Furthermore, reworking algorithms to exploit small dense structure promises increased locality, by increasing computational work per data movement, as well as better exploitation of fine-grained parallel hardware, like short vector units. This motivation is clearly represented in block sparse matrix formats, like block compressed sparse row ("block CSR"). These approaches exploit the topological regularity of a discretization with co-located degrees of freedom at each mesh node (for instance). Assuming there are n degrees of freedom per node, then a block CSR matrix will contain dense matrices of size $n \times n$. If this size block fits into cache, then the locality will improve performance. However, there may be an "unfortunate middle range" of block sizes that are too small for thread parallelization within a single block, but large enough that vectorizing across blocks takes too much local memory. This may call for explicit vectorization within a single block, for example by using compiler directives or intrinsics.

Performant block entry code is challenging to write. [Pierson] gave a poster on this theme. Standard libraries like the BLAS have too much overhead and are not optimized for the small matrix case. Even specialized libraries, like NVIDIA's cuBLAS, do not expose efficient dense operations on very small matrices. There is still a lot of uncertainty about optimal implementation decisions for small dense matrix operations. Furthermore, the workshop showed no knowledge transfer from BLAS implementation lessons learned, even though the basic building blocks of an efficient large dense matrix-matrix multiply are small block operations. This uncertainty and lack of a standard implementation contrasts the conventional wisdom that more blocks will improve performance.

Large sparse matrices

PDE discretizations on unstructured grids naturally produce large sparse graphs and matrices. These data structures show up in many other applications as well. Users create sparse graphs and matrices, modify their sparsity structure or values, and apply computational kernels, like a sparse matrix-vector multiply or triangular solve. Operations that modify a sparse graph or matrix are referred to as fill, in contrast to computational kernels that tend to be part of a linear system solve.

A popular data structure that has been used for unstructured assembly is the compressed sparse row (CSR) storage scheme. However, changes in computer architecture have make the scalability of this data structure (and of matrix assembly in general) a point of discussion (see [Brown]). [Yang] presented a linear algebra interface from the HYPRE [1, 8, 15] package that had multiple interfaces that specialized in both structured and unstructured matrix storage. Storage of structured data (generated from a Cartesian grid for instance) may offer substantial performance improvements based on the regularity of the data access.

Coarse- and fine-grained operations

User operations on sparse graphs or matrices can be separated into two categories based on the amount of work that a single operation does. Coarse-grained operations do enough work inside that it pays for them to exploit all available levels of parallelism that make sense. Examples include sparse matrix-vector multiply over the whole matrix, global data redistribution, making a deep copy of a sparse data structure, or assembling a very large collection of element stiffness matrices into a sparse matrix. Users call coarse-grained operations sequentially or with the semantics of MPI collectives and expect them to be parallel inside. Fine-grained operations do very little work inside with the expectation that users will call them in a parallel way to achieve scalability. For example, reading or writing a few entries of a sparse matrix is a fine-grained operation.

The talk by [Hoemmen] explained how the interface differences between fine- and coarse-grained operations impose different performance requirements. Fine-grained operations have much tighter performance requirements than coarse-grained operations, since they have less opportunity to amortize overheads over large collections of data. This also constrains how they access shared data structures. For example, fine-grained operations cannot just be thread-safe; they must also be thread-scalable. This is not so hard for modifying a single entry in a sparse matrix. However, solving this for dynamic data structures like hash tables and sparse graphs requires a concerted effort to construct lock-free thread-scalable data structures. An additional advantage of fine-grained interfaces is they may help porting codes from current single-thread software architectures to future multiple-thread architectures. A fine-grained thread-safe/scalable interface would fit naturally within the OpenMP threading paradigm for instance.

A software interface could support coarse grained matrix fill operations by requiring users to "batch" updates together and submit them as a single collection. Many finite-element codes do a sequential analog of this already, by breaking up assembly into subsets of elements, called worksets. This increases locality and saves memory. The batched fill approach naturally fits into a task-parallel programming model. For example, each workset might map to a task. Parallelizing within a workset would still require fine-grained operations. The sparse matrix itself would also need to do fine-grained operations inside of its coarse-grained fill interface.

Compatible with task-based parallelism, but not tied to it

A recurrent theme of this workshop was the use of task parallelism. Construction of sparse linear systems should be compatible with the task-parallel programming model. At a minimum this implies linear algebra interfaces should not impose thread safety or scalability issues. The pertinent question is does the linear algebra interface need to reflect the task-parallel programming model explicitly? For instance, some scientific codes today represent sparse fill with a blocking interface: functions that modify a graph or matrix block until they complete their work. An alternative would be a nonblocking interface instead. This interface

might follow a dataflow model compatible with a task-parallel framework. Methods might return requests, comparable to those returned by MPI's nonblocking two-sided communication, or they might require a fence to ensure completion. Current codes do not generally express fill interfaces in this way, because our physical models have locality that lends itself to avoiding remote accesses. Models with less locality, and computer architectures that favor fine-grained latency hiding, might perform better with a nonblocking fill interface. However, it's not clear that we need to complicate our fill interfaces in this way. If fill is thread-safe and thread-scalable, and fill operations have transactional semantics (they either succeed, or fail with no externally visible side effects), then parallel tasks could safely and performantly execute fill operations.

Task parallelism still has value in a linear algebra library, though. Operations internal to a linear algebra library, like message buffer packing and unpacking for sparse matrix-vector multiply and data redistribution, could benefit from task parallelism. Breaking up large messages into modestly sized ones and overlapping communication with packing or unpacking operations would be a natural way to exploit the higher message injection rates of modern network hardware.

For users of the linear algebra library, it could be natural to interact with nonblocking coarse-grained operations as tasks with dependencies. Nonblocking collectives (such as those available in MPI 3) have enabled the development of new iterative linear solvers, like the pipelined versions of GMRES [11] and CG [10], which can overlap global inner products or norms with sparse matrix-vector multiplies or preconditioner applications. Nonblocking communication can help reduce the effects of dynamic load imbalance and system noise [13]. The latter may arise due to local recovery from hardware faults. Expressing nonblocking operations as tasks with dependencies can avoid common user errors, like giving the result of a sparse matrix-vector multiply to an inner product before it is ready. It could even discover optimizations like overlap or kernel fusion automatically, even in existing solver algorithms [14].

"Count, allocate, fill, compute"

In MPI-only, codes dynamic memory allocation is permitted as long as load was balanced evenly over MPI processes. However, with thread parallelism, operations like allocation of memory shared between threads may require expensive synchronization. Some programming models, like NVIDIA's CUDA, forbid or discourage dynamic allocation. Others just make the synchronization implicit (and costly). High-performance allocators like TCMalloc [9, 17] may eliminate much of this cost, at least for entirely thread-local memory. However, the problem is algorithmic. If threads want to share access to memory, they must synchronize. We prefer instead to treat shared memory allocation as a "thread collective," like `MPI_Allreduce` or `MPI_Barrier`.

This suggests that users should structure their codes to treat dynamic allocation of shared state as an expensive collective operation. We propose the following model:

1. Count (or estimate) the required allocation size, in parallel
2. Allocate space, as a thread-collective operation
3. Fill that space with data, computed in parallel
4. Compute with the filled data structure

[Edwards] and [Sunderland] presented an example of this pattern, namely determining the structure of a sparse matrix resulting from an unstructured mesh PDE assembly[1].

1. In parallel, for each row of the sparse matrix, count or estimate the number of entries in the row.
2. Allocate space for the graph.
3. Fill the graph with entries, in parallel.
4. Use the graph for finite-element assembly and iterative linear solves.

In this application, it's possible to get an accurate count of the required number of entries. This is not always true, but often one can estimate in advance. In that case, fill may fail by running out of space, but users may try again with a better estimate. For example, each fine-grained fill operation may return the number of successfully inserted entries. If users treat fill as a thread-parallel sum-reduction over those return values, then the reduction result will give them the right amount of space to allocate. Even if fill succeeded, that result tightens the original estimate.

A similar approach is used in the Linear-Algebraic System Interface (IJ) of the HYPRE library [1, 8, 15]. The use of assumed partition algorithms for determining the global distribution in $O(1)$ storage and $O(\log P)$ computations, has been essential for the parallel scalability of HYPRE's user interfaces (see [Yang]). In general, avoiding fine-grained dynamic allocation is not new. However, historical trends in memory growth and ease of use in popular programming languages have made the practice less expensive. It is expected that in next-generation architectures this will no longer be the case.

Algorithms and Discretizations for Next Generation Architectures

This report has previously focused on issues associated with the current practices and discretizations in computational simulation. However, the change in architectures may change the types of algorithms and discretizations that run efficiently. For instance [Brown] and [Kolev] both discussed the use of high-order discretizations in a matrix-free context to try to

[1]There are different ways to do this in parallel, and they compare to the different assembly approaches (scatter-atomic-add, gather-sum, and parallel prefix sum) discussed previously.

take advantage of the larger FLOP-to-byte ratio. [Phipps] presented an approach where the assembly was performed over an ensemble of sample points to accelerate collocation based forward uncertainty propagation. The choice of discretization and algorithm for accelerating assembly must be balanced with the need to use conservative and physics compatible discretizations, and to choose methods that have good parallel scalability at a range of scales. The section discusses these approaches and explores the complex tradeoff space associated with these choices.

High-order methods

For smooth problems and those that must limit numerical diffusion and dispersion near the grid scale, high order methods can significantly reduce the number of degrees of freedom needed to reach a desired accuracy. High-order methods tend to achieve higher utilization of floating-point hardware than low-order methods, as evidenced by higher floating-point throughput. This is epitomized by numerous Gordon Bell awards to spectral element packages such as Nek5000, SPECFEM, and HOMME. This high intensity comes with increased pressure on caches since the working set may no longer fit within L1 cache. [Brown]'s presentation discussed vectorization strategies and cache/threading issues impacting achievable performance for moderate-order FEM. To better reuse registers and the fastest levels of cache, it becomes necessary to have many threads operate on the same element (or other smallest natural unit of computation). This results in more complicated code generation (usually at least partially manual) and more inter-thread communication. For the case that the cost of each element lies near the cost of the lightest weight thread synchronization primitives, coordinating hardware threads to work together on an element may not pay off. The challenges related to effective reuse of registers and cache across multiple hardware threads may compromise the efficiency of such methods through repeated spills and/or synchronization overhead. In other words, if vendors continue to raise the number of hardware threads per core without commensurate improvements in caches and low-latency synchronization, floating point hardware will be underutilized, ultimately leading to applications running at ever-decreasing fractions of peak.

A second issue associated with high-order methods is that assembling matrices is much less desirable than for low-order methods because the number of nonzeros per row typically grows as the cube of the order of the method (in 3D). Indeed, assembly of matrices is not intrinsically necessary for solving PDEs; it is an artifact of algebraic preconditioners, like domain-decomposition with incomplete factorization, requiring an explicit representation of the matrix entries. Assembled sparse matrices can suffer from memory bandwidth bottlenecks on modern architectures, possibly limiting performance to a few percent of arithmetic peak. This is especially true for high-order methods, where the sparsity of the matrix decreases with the order. Additionally, coordinating threads for matrix assembly is generally considered to be more challenging than for residual assembly. It is thus worth reconsidering the solver algorithms and interfaces to determine what is necessary to assemble. The potential performance of these methods must be weighed against the convenience and flexibility of reusing more standard components based on assembled matrices.

Some of the trade-offs between the various levels of assembly (ranging from full to matrix-free) were examined by [Kolev] in the context of high-order finite elements for shock hydrodynamic applications (the BLAST [4, 6] code and MFEM [21] finite element library at LLNL). With full or element-level assembly, the memory access per degree of freedom for a matrix-vector product grows with the element order, while it remains bounded in the matrix-free and quadrature-point based storage approaches. Furthermore, the latter approaches are also more efficient with respect to the number of computations per degree of freedom if the polynomial degree is high enough, e.g. at least 3 in 2D and 2 in 3D. The overall effect of reduced-storage assembly in shock hydrodynamic simulations with the BLAST code are increased strong parallel scalability and orders of magnitude reduction in the runtime for 4th and higher-order elements in 2D.

[Brown]'s presentation demonstrated integer-factor speedups and memory reduction for a lithospheric dynamics package (pTatin) using Q2-P1 Stokes elements in 3D. For the heterogeneous viscoplastic Stokes solves, a combination of matrix-free geometric multigrid on the finest levels paired with algebraic multigrid on coarser levels was found to be especially effective [18].

Solution methods

When considering any new discretization approach, be-it high-order matrix-free or low-order assembled, its important to consider the efficiency of the solve phase. While this workshop was primarily about assembly, choosing a discretization approach that is difficult in the solve phase but easy to assemble may not be beneficial.

For instance, [Brown] discussed how the size of vertex separators can have a large effect on the solution methods used and required amount of communication. Direct solves for multi-dimensional problems result in fill that scales superlinearly with problem size. For large problems, this fill becomes the leading memory cost and factorization of the large associated (dense) supernodes becomes the leading time cost. The size of the largest supernode is equal to the minimal "vertex separator", the set of vertices in the matrix graph that split the graph into two separate parts. Vertex separators scale as $n^{\frac{1}{2}}$ and $n^{\frac{2}{3}}$ for isotropic domains in 2D and 3D respectively, but they also depend on the discretization. High-order finite-difference methods result in a separator proportional to the "stencil width." Since memory use is quadratic in separator size and factorization time is cubic, a small increase in stencil width results in ballooning factorization cost. Some methods, such as standard p-version FEM, have small vertex separators independent of approximation order. Other methods such as high-order FD, FV, and Discontinuous Galerkin have vertex separators that grow with approximation order. Some of those have lower rank coupling in a transformed space and can be reformulated to expose that compactness (e.g., hybridizable DG). In addition to direct solvers, small vertex separators are important to control costs for nonlinear Dirichlet domain decomposition, algebraic multigrid, and other methods.

High-order methods are also more difficult to use with multigrid because poor h-ellipticity

requires more powerful smoothers or the use of more sophisticated techniques such as dual-order defect correction schemes. Multilevel domain decomposition can in some cases do a better job of controlling complexity, but often requires direct subdomain solves, which in turn require assembled matrices and generally high cost. There is some reprieve, however, because discretizations such as p-FEM possess small vertex separators, resulting in no more fill than a low-order method with the same number of degrees of freedom.

Multigrid and related techniques are an essential ingredient for solving stiff and steady-state PDEs. As discretizations are changed to improve performance and alternative representations are used to represent sparse matrices, we must evaluate the efficiency, robustness, and implementation cost of suitable multigrid methods. First, increasing the approximation order degrades h-ellipticity (a necessary and sufficient measure for applicability of a pointwise smoother), thus requiring more expensive smoothers. For separable PDEs on near-affine meshes, the technique of "sum factorization" that is used by Nek5000 (see [Fischer]) and other SEM packages is an inexpensive element-wise near-exact solve that is sufficient for rapid p-coarsening to a low-order discretization that is amenable to standard AMG. Sum factorization is not available for general operators, often leading to much higher smoothing costs. A popular and effective technique is to use a defect correction based on a low order discretization embedded in the high order discretization. This technique is more generally applicable and is effective for a broad range of problems, but is less efficient when sum factorization is applicable.

If one eschews assembled matrices, it may also be attractive to avoid global linearization in favor of nonlinear multigrid methods (usually FAS). Discretizations like FEM have high overhead for pointwise residual or Jacobian evaluation relative to global evaluations; for example, an element must be visited once for each adjacent vertex in a pointwise multiplicative smoother, but only once for a global residual. It is desirable to either find discretizations that can evaluate pointwise residuals efficiently or to find smoothers such as multi-stage/polynomial that require only global residual evaluation. Such smoothers are typically easier to parallelize and vectorize than multiplicative smoothers, but suffer from inferior robustness.

Reuse across other dimensions

Many problem of interest involve not just spatial dimensions, but also temporal and stochastic/uncertain dimensions. In some cases, spatial discretizations and associated data structures may be reused to reduce memory motion and improve efficiency. This is obvious for a linear problem where the reuse appears as multiple right hand sides or a Kronecker product system, but nonlinear problems may also be amenable to local linearizations that enable reuse. For example, implicit Runge-Kutta systems are often solved by modified Newton (a shared linearization) and a decomposition of the resulting Kronecker product system [3, 5, 28].

Often forward uncertainty propagation for problems with uncertain input data involve

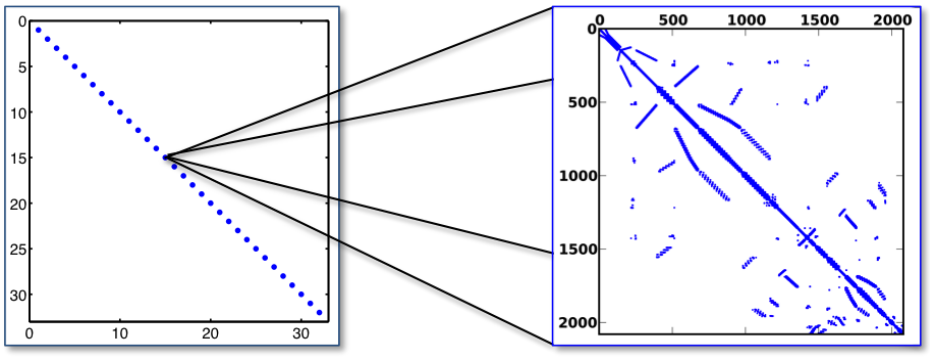

Figure 3. Kronecker-product matrix generated by sampling-based uncertainty quantification methods.

samples of the PDE solution evaluated at numerous realizations of that uncertain data. This too can be formulated as a block-diagonal Kronecker product system such as the one shown in Figure 3, where the diagonal blocks represent FEM matrices for each sample. For many problems, significant reuse of data accessed and generated through the PDE assembly and solution processes is possible from realization to realization. Furthermore, [Phipps] presented a sampling-based uncertainty propagation method where samples are grouped into ensembles of some fixed size determined by the architecture's native vector width and the Kronecker product system above was formed for each ensemble. The Kronecker system was commuted to a block spatial system where each block is a diagonal matrix given by the ensemble size such as the one shown in Figure 4, and then applied to PDE assembly of low-order discretizations on unstructured meshes. The template-based generic programming approach presented by [Pawlowski] was used to effect this reordering without requiring explicit management of the ensemble dimension in the assembly code. It was found that this approach enabled not only reuse of the unstructured mesh between samples within the ensemble (and therefore amortized the latency costs of those data structures), but also mapping of fine-grained SIMD/SIMT parallelism across the ensemble resulting in significantly improved performance. Thus we see that when the system is organized suitably, other dimensions of the full problem discretization can be exploited for better memory access patterns and fine-grained parallelism, potentially mitigating the need for the spatial discretization to achieve these goals.

Application Needs and Capabilities

Applications require a diverse set of capabilities from both discretizations and assembly, which historically have resulted in limited adoption of abstraction, especially compared to the success of abstractions in linear and nonlinear solvers. This diversity is ever growing, especially as applications shift toward higher order and application specific discretizations.

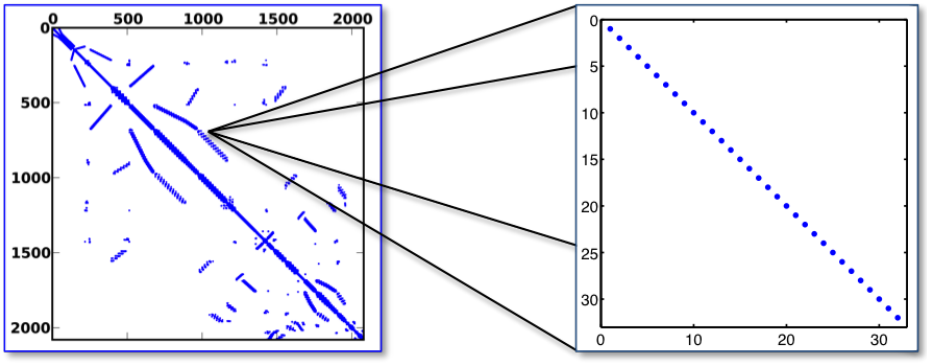

Figure 4. Commuted Kronecker-product matrix generated by sampling-based uncertainty quantification methods.

Physics-motivated strategies at the nexus of physics and algorithms, such as upwinding nonlinear coefficients, discrete outflow boundary conditions, or variational crimes, further make abstractions difficult to adopt. Historically, most applications have chosen to implement their own discretizations and assembly.

However, extreme-scale machines have provided new energy and motivation for the adoption of libraries, frameworks, DSLs, and other tools. As applications see their existing codes run slower on newer machines, and uncertainty in discretization selection, data layout, and best practices for emerging architectures increases, application software developers are increasingly interested in software for managing abstractions in both discretization and assembly. Successes in abstractions for solvers such as PETSc and Trilinos have resulted in new generations of application software authors willing to adopt libraries that require the application to cede some control.

However, such abstractions must work closely with applications, and may require differing concepts for differing fields. However, several common themes were identified in this workshop, and potential abstraction designers should be aware of these themes.

The first theme is the ubiquity of block systems. Few applications assemble and solve a single operator. Block systems arise in applications in several ways. Mixed finite element methods such as Raviart-Thomas and higher-order discretizations such as mimetic finite differences or higher order finite elements naturally introduce a block system for entities in different spaces or on different mesh entities. Tightly coupled physical equations, such as coupled energy and fluid flow equations, result in block systems. And multi-domain simulations often result in constraints that are expressed as a block system with the equations for the primal variable. However, block systems provide unique difficulties to abstractions for assembly. This concern arose in several applications including coupled thermal-hydrological flows in the subsurface (Coon), ice sheet evolution ([Demeshko]), Navier-Stokes/Stokes equations ([Fischer], [Cyr]), and shock hydrodynamics and MHD ([Drake]), amongst others. Nonlinear

coefficients and boundary conditions have always been important to the application community, and are only becoming more common. Abstractions must be aware of difficulties presented by these nonlinearities. For instance, even simple applications may require nonlinear Robin boundary conditions, and multi-domain simulations may use boundary conditions or constraints on boundary unknowns to couple domains. Boundary conditions such as outflow boundaries where a normal component is fixed while a tangential component is not may result in systems where the discrete system is well-posed while the continuous one is not. Inflexibility in ways of allowing physics-specific coefficients and boundary conditions is a frequent reason for applications to implement their own assembly process.

Much as DAGs provide opportunities for assembly, they also are a powerful idea within multiphysics applications. By enabling and enforcing modularity, they enable testing, automate model evaluation within complex codes, and partially automate the often buggy process of assembling Jacobians. DAGs have been adopted extensively in several application code efforts presented, including Amanzi and the Arctic Terrestrial Simulator via Arcos (Coon and [Moulton]), Uintah ([Berzins]), Moose and its family of codes ([Andrs]), and Albany through Phalanx ([Demeshko], [Pawlowski]). While DAG abstractions may be very application specific, the underlying data structures storing the graph and task schedulers can be very general. Increasing adoption of the DAG concept within application codes provides opportunities for abstraction for both discretization and assembly. DAGs should be viewed as a very promising tool for exposing concurrency and providing a way for physics to communicate problem structure to algorithms.

Another commonality of the applications codes represented was the role of coupling in dictating assembly and operator layout. For instance, in the Arctic Terrestrial Simulator, which models phase change in ice-rich tundra, tightly coupled energy and hydrology equations have been viewed in two ways - as a single block operator with interleaved unknowns (equation is fastest varying, followed by mesh entity), and as two, independent operators in a block system (mesh entity is fastest varying, followed by equation). Each view of the system of equations has its advantages: preconditioners for the coupled system are more diagonally dominant and best suited to, for instance, multigrid solvers, if organized using the former data layout, while code modularity encourages the adoption of the latter layout. Additionally, block preconditioning strategies such as Schur complements for Stokes problems may make it less clear as to which strategy is best for which application. [Cyr] explored how simple mappings can be used to map between these layouts, providing extremely useful functionality for enabling modular physics code. However, this dichotomy of two views of the same operators provides unique challenges to assembly abstractions; ideally both are available and neither is assembled until it is actually required.

And finally, several applications demonstrated the need for care in assembly as it relates to other computational components, including the mesh framework, discretization, and solver strategy. Several presenters discussed how changes in mesh implementations resulted in performance changes in assembly, including [Drake], with respect to remeshing, remapping, and iteration over mesh objects, and [Sahni], with respect to unstructured meshes near boundary layers. High order discretization methods are becoming increasingly common in application

codes due to their improved efficiency, as discussed above. These can be leveraged using libraries such as MFEM and Blast, as demonstrated by [Kolev], placing new requirements on assembly. Differing solution methodologies on different physical approximations may lead to different approaches for when to assemble and when not to. [Pautz] showed an example of this on two forms of the Boltzmann equations within SCEPTRE.

Conclusions

In principle, PDE discretization and assembly are some of the simplest computations to design and implement for efficient execution on any modern computing system. For any specific discretization scheme, application code and computer system, an optimal implementation is possible and fairly straightforward to implement. The challenge is that we do not want to, nor can afford to, provide such a custom implementation for each important case. There are too many. Instead we need to define meaningful abstractions that enable performance, reusability and expressibility for broad collections of discretizations. This is the overarching goal of our efforts.

This report described a number of challenges and approaches to achieving high performance assembly abstractions and implementations on next generation multicore and many core architectures. Principle among these issues was how to expose and exploit the parallelism available in assembly required by the current generation of codes. A number of approaches for resolving race conditions in assembly were proposed. Additionally, new parallel programming models based on a highly threaded SIMD architecture were also discussed. To expose further parallelism novel software architectures based on directed acyclic graphs were considered. The intriguing possibility of achieving high performance by combining those approaches was alluded to but not considered in detail. Critical to any assembly algorithm is the interface between high density computation and scatter and gather to and from sparse linear algebra data structures. This report considered both fine and coarse grained interfaces for this activity. In addition, interfaces and data structures that supported exposure of additional structure where possible within a calculation were considered. Finally, the notion of the "count, allocate, fill, compute" process proved to be a useful abstraction when forming matrices. While the focus of the report was on assembly, the possibility of new discretizations and mathematical approaches that may lead to algorithms that better map to new hardware was also discussed. One approach is to consider higher-order discretizations that, in a simplistic explanation, increase the FLOP rate to memory access ratio. Additional approaches focusing on taking advantage of structure in time integration and uncertainty quantification were also proposed. Finally, the use of these techniques as abstractions within a PDE application was explored. In particular, the growing importance of abstraction in those codes and the need for a software design cycle between the application and the component abstraction developers was emphasized.

This report has laid out a number of innovative approaches for improving assembly, both in performance but also in flexibility. However, it has also exposed a number of areas for future improvement and study. A few that are particularly critical:

- the effectivity of task parallelism for implicit solution methods and assembly, and the importance of computationally expensive equations of state to achieve good scalability,
- the utility, need for, and abstraction of the memory hierarchy,
- the evolution of static and dynamic linear algebra data structures for assembly and

solve,

- the use of advanced discretization and the connected effect on the solver stack,
- exploiting more parallelism in beyond forward simulation techniques like optimization and uncertainty quantification when mapping onto advanced architectures, and
- new software abstractions meeting the needs of applications beyond the mathematical abstraction.

Finally, improvement in the area of assembly performance and utilization of next generation hardware will require a substantial community-wide commitment to abstraction. These issues are often derisively viewed as "software" problems in the computational sciences. The question is, can this attitude be overcome so that a vibrant research community is sustained?

References

[1] Allison H. Baker et al. "Scaling Hypre's Multigrid Solvers to 100,000 Cores". In: *High-Performance Scientific Computing*. Springer London, 2012, pp. 261–279.

[2] M. Berzins et al. "Past, present and future scalability of the Uintah software". In: *Proceedings of the Extreme Scaling Workshop*. University of Illinois at Urbana-Champaign. 2012, p. 6.

[3] T. A. Bickart. "An efficient solution process for implicit Runge-Kutta methods". In: *SIAM Journal on Numerical Analysis* 14.6 (1977), pp. 1022–1027.

[4] *BLAST: High-Order Curvilinear Finite Elements for Shock Hydrodynamics*. `http://www.llnl.gov/CASC/blast`.

[5] J. C. Butcher. "On the implementation of implicit Runge-Kutta methods". In: *BIT Numerical Mathematics* 16.3 (1976), pp. 237–240.

[6] V. Dobrev, Tz. Kolev, and R. Rieben. "High-Order Curvilinear Finite Element Methods for Lagrangian Hydrodynamics". In: *SIAM J. Sci. Comp.* 34.5 (2012), pp. 606–641.

[7] H. C. Edwards, C. R. Trott, and D. Sunderland. "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns". In: *Journal of Parallel and Distributed Computing* (2014).

[8] R. D. Falgout, J. E. Jones, and U. M. Yang. "Pursuing Scalability for *hypre*'s Conceptual Interfaces". In: *ACM Trans. Math. Softw.* 31.3 (2005), pp. 326–350.

[9] S. Ghemawat and P. Menage. "Tcmalloc: Thread-caching malloc". In: *goog-perftools. sourceforge. net/doc/tcmalloc. html* (2009).

[10] P. Ghysels and W. Vanroose. "Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm". In: *Parallel Computing* (2013).

[11] P. Ghysels et al. "Hiding global communication latency in the GMRES algorithm on massively parallel machines". In: *SIAM Journal on Scientific Computing* 35.1 (2013), pp. C48–C71.

[12] *HIGH BANDWIDTH MEMORY (HBM) DRAM*. `http://www.jedec.org/standards-documents/docs/jesd235`. [Accessed: Sept. 15, 2014].

[13] T. Hoefler, T. Schneider, and A. Lumsdaine. "Characterizing the influence of system noise on large-scale applications by simulation". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society. 2010, pp. 1–11.

[14] M. Hoemmen and K. Nusbaum. *Asynchronous, performance-portable Krylov methods on accelerators*. Tech. rep. Sandia National Laboratories, 2012.

[15] hypre*: High Performance Preconditioners*. `http://www.llnl.gov/CASC/hypre`.

[16] L. V. Kale and S. Krishnan. *CHARM++: a portable concurrent object oriented system based on C++*. ACM, 1993.

[17] S. Lee, T. Johnson, and E. Raman. "Feedback directed optimization of TCMalloc". In: *Proceedings of the workshop on Memory Systems Performance and Correctness*. ACM. 2014, p. 3.

[18] D. A. May, J. Brown, and L. Le Pourhiet. "pTatin3D: High-performance Methods for Long-term Lithospheric Dynamics". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '14. New Orleans, Louisana: IEEE Press, 2014, pp. 274–284. ISBN: 978-1-4799-5500-8. DOI: 10.1109/SC.2014.28. URL: http://dx.doi.org/10.1109/SC.2014.28.

[19] D. S. Medina, A. St-Cyr, and T. Warburton. "OCCA: A unified approach to multi-threading languages". In: *arXiv preprint arXiv:1403.0968* (2014).

[20] C. Mellor. *Micron: Our STACKED SILICON BEAUTY solves the DRAM problem.* The Register. 27 Nov. 2013. http://www.theregister.co.uk/2013/11/27/micron_engaged_in_consenting_dram_stackery/. [Accessed Sept. 15, 2014].

[21] *MFEM: Modular parallel finite element methods library*. http://mfem.googlecode.com.

[22] *Micron HMC Memory Technology to Enhance Knights Landing*. http://insidehpc.com/2014/06/micron-hmc-memory-technology-enhance-knights-landing/. [Accessed Sept. 21, 2014].

[23] P. K. Notz, R. P. Pawlowski, and J. C. Sutherland. "Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software". In: *ACM Transactions on Mathematical Software (TOMS)* 39.1 (2012), p. 1.

[24] R. P. Pawlowski, E. T. Phipps, and A. G. Salinger. "Automating embedded analysis capabilities and managing software complexity in multiphysics simulation, Part I: Template-based generic programming". In: *Scientific Programming* 20.2 (2012), pp. 197–219.

[25] *SpatialOps: Main Page*. http://minimac.crsim.utah.edu:8080/job/SpatialOps/doxygen. [Accessed Nov. 28, 2014].

[26] J. Davison de St Germain et al. "Uintah: A massively parallel problem solving environment". In: *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*. IEEE. 2000, pp. 33–41.

[27] S. Valat, M. Pérache, and W. Jalby. "Introducing kernel-level page reuse for high performance computing". In: *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. ACM. 2013, p. 3.

[28] G. Wanner and E. Hairer. *Solving ordinary differential equations II: Stiff and Differential-Algebraic Problems*. Berlin: Springer-Verlag, 1991.

Presentations

[Andrs] D. Andrs. "Massive Hybrid Parallelism for Fully Implicit Multiphysics". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Bennett] J. Bennett. "Fault-tolerant programming at the extreme-scale". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Berzins] M. Berzins. "Software Abstractions for Extreme-Scale Scalability of Computational Frameworks". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Brown] J. Brown. "High-performance matrix-free operator application and preconditioning". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Cyr] E. C. Cyr, B. Seefeldt, and R. P. Pawlowski. "Global Unknown Numbering for Fully-Coupled Mixed Finite Element Methods". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Demeshko] I. Demeshko. "A performance-portable implementation of the Finite Element Assembly: preliminary results of using Kokkos in the Albany code". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Drake] R. Drake. "The ALEGRA Production Application: Strategy, Challenges and Progress Toward Next Generation Platforms". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Edwards] H. C. Edwards. "MiniFENL: Fully Hybrid Parallel and Performance Portable Nonlinear Finite Element Miniapp using MPI+Kokkos". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Fischer] P. Fischer. "Scaling PDE solvers beyond a million cores". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Franko] K. Franko. "MiniAero". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Heroux] M. Heroux. "Challenges and Opportunities for Scalable Finite Element Assembly". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Hoemmen] M. Hoemmen. "Tpetra interface changes to support thread-parallel fill". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Kirby] R. Kirby. "Fine-grained finite element parallelism". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Kolev] T. Kolev. "Scalable high-order finite elements with MFEM, hypre and BLAST". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Moulton] D. Moulton. "Amanzi and the Arctic Terrestrial Simulator: Flexible Multiphysics Simulators for Environment and Ecosystem Applications". In: Presented at "Algorithms and Abstractions for Assembly in PDE Codes", 2014.

[Pautz] S. Pautz. “Matrix Assembly Tasks in the Sceptre Deterministic Radiation Transport Code”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Pawlowski] R. P. Pawlowski. “Template-based Generic Programming Techniques for Finite Element Assembly”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Phipps] E. Phipps. “Improving PDE Assembly Performance Through Embedded Uncertainty Quantification”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Pierson] K. Pierson. “Efficient Block Sparse Assembly with SIMD”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Sahni] O. Sahni. “Abstractions and algorithms for adaptive methods on boundary layer meshes”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Stogner] R. Stogner. “C++14 Generic Programming as a Domain-Specific Language for PDEs”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Sunderland] D. Sunderland. “Thread Scalable CRS Graph Construction”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Sutherland] J. Sutherland. “Flexible, Efficient Abstractions for High Performance Computation on Current and Emerging Architectures”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Trott] C. Trott. “A migration strategy for utilizing the Kokkos many-core programming model”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Turcksin] B. Turcksin. “Multithreaded matrix assembly for finite elements”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Warburton] T. Warburton. “OCCA: A Unified Approach to Multi-Threading Languages”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

[Yang] Ulrike Yang. “Matrix and Vector Assembly in hypres Conceptual Interfaces”. In: Presented at “Algorithms and Abstractions for Assembly in PDE Codes”, 2014.

A Workshop Program

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY

Program for "Algorithms and Abstractions for Assembly in PDE Codes" Workshop at Sandia National Laboratories*

May 12-14, 2014

FORMAT: All talks will be in CSRI/90. There will be three one hour long (with ten minutes for questions) keynote talks, and 24 "poster blitz" talks. Each blitz speaker is given 15 minutes to summarize the main points of their poster that will be presented. The idea is to stimulate deeper conversation on these topics. The poster may only be displayed during the allocated poster session following the poster blitz at the end of the day. Presenters are responsible for making sure their own poster is displayed.

SCHEDULE:

Monday - May 12, 2014

Start Time	Speaker and Title
9:00a-9:15	Opening Remarks
9:15a-10:15	Keynote: **Martin Berzins** - University of Utah *Software Abstractions for Extreme-Scale Scalability of Computational Frameworks*
10:15a-10:30	Break
10:30a-12:00	Poster Blitz **Andreas Kloeckner** - University of Illinois *Operator transformation and code generation for FEM* **Roy Stogner** - University of Texas *C++14 Generic Programming as a Domain-Specific Language for PDEs* **Irina Demeshko** - Sandia National Laboratories *A performance-portable implementation of the Finite Element Assembly: preliminary results of using Kokkos in the Albany code* **James Sutherland** - University of Utah *Flexible, Efficient Abstractions for High Performance Computation on Current and Emerging Architectures* **Roger Pawlowski** - Sandia National Laboratories *Template-based Generic Programming Techniques for Finite Element Assembly* **Janine Bennett** - Sandia National Laboratories *Fault-tolerant programming at the extreme-scale*
12:00-1:30p	Lunch
1:30p-2:30	Poster Blitz **David Andrs** - Idaho National Laboratory *Massive Hybrid Parallelism for Fully Implicit Multiphysics* **David Moulton** - Los Alamos National Laboratory *Amanzi and the Arctic Terrestrial Simulator: Flexible Multiphysics Simulators for Environment and Ecosystem Applications* **Ken Franko** - Sandia National Laboratories *MiniAero* **Eric Phipps** - Sandia National Laboratories *Improving PDE Assembly Performance Through Embedded Uncertainty Quantification*
2:30p-5:00	Poster Session

*Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Tuesday - May 13, 2014

Time	Speaker and Title
9:00a-10:00	Keynote: **Mike Heroux** - Sandia National Laboratories
	Challenges and Opportunities for Scalable Finite Element Assembly
10:00a-10:30	Break
10:30a-12:00	Poster Blitz
	Rob Kirby - Baylor University
	Fine-grained finite element parallelism
	Shawn Pautz - Sandia National Laboratories
	Matrix Assembly Tasks in the Sceptre Deterministic Radiation Transport Code
	Tim Warburton - Rice University
	OCCA: A Unified Approach to Multi-Threading Languages
	Tzanio Kolev - Lawrence Livermore National Laboratory
	Scalable high-order finite elements with MFEM, hypre and BLAST
	Onkar Sahni - RPI
	Abstractions and algorithms for adaptive methods on boundary layer meshes
	Rich Drake - Sandia National Laboratories
	The ALEGRA Production Application: Strategy, Challenges and Progress Toward Next Generation Platforms
12:00-1:30p	Lunch
1:30p-2:30	Poster Blitz
	Dan Sunderland - Sandia National Laboratories
	Thread Scalable CRS Graph Construction
	Ulrike Yang - Lawrence Livermore National Laboratory
	Matrix and Vector Assembly in hypres Conceptual Interfaces
	Mark Hoemmen - Sandia National Laboratories
	Tpetra interface changes to support thread-parallel fill
	Bruno Turcksin - Texas A&M
	Multithreaded matrix assembly for finite elements
2:30p-5:00	Poster Session

Wednesday - May 14, 2014

Start Time	Speaker and Title
9:00a-10:00	Keynote: **Paul Fischer** - Argonne National Laboratory
	Scaling PDE solvers beyond a million cores
10:00a-10:15	Break
10:15a-11:30	Poster Blitz
	Jed Brown - Argonne National Laboratory
	High-performance matrix-free operator application and preconditioning
	Carter Edwards - Sandia National Laboratories
	MiniFENL: Fully Hybrid Parallel and Performance Portable Nonlinear Finite Element Miniapp using MPI+Kokkos
	Christian Trott - Sandia National Laboratories
	A migration strategy for utilizing the Kokkos many-core programming model
	Kendall Pierson - Sandia National Laboratories
	Efficient Block Sparse Assembly with SIMD
	Eric C. Cyr - Sandia National Laboratories
	Global Unknown Numbering for Fully-Coupled Mixed Finite Element Methods
11:30p-12:30	Poster Session

ABSTRACTS: Speakers in bold

David Andrs, Derek Gaston, Cody Permann, John Peterson, Andrew Slaughter, Richard Martineau
Title: *Massive Hybrid Parallelism for Fully Implicit Multiphysics*
Abstract: As hardware advances continue to modify the supercomputing landscape, traditional scientific software development practices will become more outdated, ineffective, and inefficient. The process of rewriting/retooling existing software for new architectures is a Sisyphean task, and results in substantial hours of development time, effort, and money. Software libraries which provide an abstraction of the resources provided by such architectures are therefore essential if the computational engineering and science communities are to continue to flourish in this modern computing environment. The Multiphysics Object Oriented Simulation Environment (MOOSE) framework enables complex multiphysics analysis tools to be built rapidly by scientists, engineers, and domain specialists, while also allowing them to both take advantage of current HPC architectures, and efficiently prepare for future supercomputer designs. MOOSE employs a hybrid shared-memory and distributed-memory parallel model and provides a complete and consistent interface for creating multiphysics analysis tools. A brief discussion of the mathematical algorithms underlying the framework and the internal object-oriented hybrid parallel design are given. Representative massively parallel results from several applications and a brief discussion of future areas of research for the framework will be presented.

Janine Bennett, John Floren, Hemanth Kolla, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke
Title: *Fault-tolerant programming at the extreme-scale*
Abstract: It is widely acknowledged that performant software at exasacle will require significant increases in fine-grained parallelism and resiliency. Asynchronous, many-task programming models are acknowledged to provide the desired levels task- and data-level parallelism and, furthermore, show promise at sustaining performance despite node degradation and failures. Asynchronous task models introduce a challenging distributed consistency problem both within and amongst several interacting components (e.g. scheduler, global address server, transport layer), which demands a large set of programming model and runtime tools to address process failures. Existing many-task solutions often have nascent resilience support, addressing a subset of the resilience problem. In this poster we outline a holistic resilience approach based on a deferred consistency model. As much as possible, we isolate the resilience problem to a distributed hash table and library of resilient collective communications, transforming the massive challenge of resilient many-task scheduling and execution into related, but better understood resilience problems.

Martin Berzins
Title: *Software Abstractions for Extreme-Scale Scalability of Computational Frameworks*
Abstract: Abstractions play a key role in the development of both computer and computational science. Often the choice of the abstraction is of key importance in enabling performance at the required level. At the same time the choice of abstraction alone may not be enough to guarantee that performance. A key abstraction in the move to extreme-scale computing is sometime stated to be that of basing execution around the concept of multiple directed acyclic graphs of tasks. We will show that using such an approach within the Utah Uintah makes it possible to separate the user specification and the runtime that executes the resulting tasks. This separation then makes it possible to scale the same (unchanged) applications code from 700 to 700K cores. The mechanism for making such an abstraction work is the constant re-engineering of the runtime system, based on a careful analysis of its performance. The techniques that make it possible for the Uintah software framework to run complex engineering applications at such scales will be described and their use illustrated in the context of problems such as modeling energetic materials, clean-coal turbulent combustion and multiscale materials by design. Finally the challenge of extending such an abstraction to present and future heterogeneous machines will be considered.

This work is joint with Alan Humphrey, Qingyu Meng and John Schmidt from the runtime system and Jacqueline Beckvermit, Todd Harman and Jeremy Thornock from the applications side.

Jed Brown, Dave May, Matt Knepley
Title: *High-performance matrix-free operator application and preconditioning*
Abstract: Assembled sparse matrices lead to algorithms with extremely low arithmetic intensity, thus using hardware inefficiently. The same linear systems can often be represented using less memory by storing information at quadrature points or flux points. In this form, operator application looks more like residual assembly. Preconditioning techniques need to be adapted to these representations. Techniques will be compared on the basis of generality and performance (up to 30% of FPU peak for some variants).

Eric C. Cyr, Ben Seefeldt, Roger Pawlowski
Title: *Global Unknown Numbering for Fully-Coupled Mixed Finite Element Methods*
Abstract: TBD

Irina Demeshko, H. Carter Edwards, Michael A. Heroux, Eric T. Phipps, Andrew G. Salinger
Title: *A performance-portable implementation of the Finite Element Assembly: preliminary results of using Kokkos in the Albany code*
Abstract: The diversity of modern HPC architectures and programming models introduces a performance portability issue: parallel code needs to be executed correctly and performant despite variation in the architecture, operating system and software libraries. In this poster we present our progress towards a performance portable implementation of Finite Element Assembly in the Albany code, based on using the Kokkos programming model from Trilinos.

Rich Drake
Title: *The ALEGRA Production Application: Strategy, Challenges and Progress Toward Next Generation Platforms*
Abstract: ALEGRA is a large, highly capable, option rich, production application solving coupled multi-physics PDEs modeling magnetohydrodynamics, electromechanics, stochastic damage modeling and detailed interface mechanics in high strain rate regimes on unstructured meshes in an ALE framework. Nearly all the algorithms must accept dynamic, mixed-material elements, which are modified by remeshing, interface reconstruction, and advection components. Recent trends in computing hardware have forced application developers to think about how to address and improve performance on traditional CPUs and to look forward to next generation platforms. Core to the ALEGRA performance strategy is to improve and rewrite loop bodies to be conformant with the requirements of high performance kernels, such as accessing data in array form, no pointer dereferencing, no function calls, and thread safety. Necessary to achieve this, however, are changes to the underlying infrastructure. We report on recent progress in the infrastructure to support array-based data access and on iteration of mesh objects. The effects on performance on traditional platforms will be shown. We also discuss the practical realities and cost estimates for attempting to move an existing full featured production application like ALEGRA toward running effectively on future platforms and being maintainable at the same time.

H. Carter Edwards
Title: *MiniFENL: Fully Hybrid Parallel and Performance Portable Nonlinear Finite Element Miniapp using MPI+Kokkos*
Abstract: MiniFENL is a miniapplication which solves a nonlinear system of equations generated from a finite element discretization. MiniFENL is implemented with MPI+Kokkos for performance-portability to heterogeneous platforms with manycore CPUs and accelerators such as Intel Xeon Phi and Nvidia GPUs. Every phase of miniFENL is hybrid parallel: internal generation of the finite element mesh, construction of the sparse linear system graph from the finite element mesh connectivity, computation of per-element nonlinear residuals and Jacobians, assembly of these per-element

contributions into the global sparse linear system, and two level Newton / conjugate gradient iterative solution of the nonlinear problem. We use miniFENL to explore hybrid parallel algorithms and performance tradeoffs across the entire solution process. For example, we recently demonstrated that a global linear system assembly scatter-add approach has better performance than a gather-sum approach on both Xeon Phi and Nvidia Kepler accelerators. The scatter-add approach uses atomic-fetch-and-add operations for thread safety whereas the gather-sum approach saves per-element contributions into a temporary array and then mines this array for a thread-safe one-thread-per-row assembly.

Paul Fischer
Title: *Scaling PDE solvers beyond a million cores*
Abstract: We discuss design and performance of communication kernels in the context of PDE based simulation at petascale and beyond. In the first part of the talk, we present a gather-scatter (GS) framework that has a particularly simple interface and has demonstrated scaling to beyond 6 million MPI ranks. The interface requires a "setup" phase in which each participating rank supplies a vector of 64-bit integers that map local degrees of freedom to their global index. In subsequent "execute" phases, ranks supply a vector, an operand type (32- or 64-bit real/int), and an associative/commutative operator (+,*,min,max) that is applied across sets of scalar or vector operands sharing the same global index. Depending on the density of the underlying graph, GS will choose one of three exchange strategies: pairwise, crystal-router (CR), or all-reduce. The latter two nominally have log P complexity, save for all-reduce on BG/L-P-Q, where all-reduce is essentially P-independent out to a million ranks. The CR is a scalable generalized all-to-many that is also used in GS-setup. We discuss the performance of these kernels in the context of billion-point simulations on over 100,000 cores.

In the second half of the presentation we examine fundamental issues that will be critical for strong scaling at exascale given current trends in compute/communication ratios. We propose hardware supported reduce-scan stategies for essential kernels (e.g., algebraic multigrid) that could mitigate the internode latency that ultimately limits strong scalability and, thus, utility of exascale platforms.

Michael A. Heroux
Title: *Challenges and Opportunities for Scalable Finite Element Assembly*
Abstract: Emerging computer architectures are forcing the finite element community to consider disruptive algorithmic and software changes in order to exploit new commodity performance curves, and address expected resilience issues at extreme scales.

In this presentation we characterize the architectural trends that pose the most significant algorithmic challenges and opportunities for the design and implementation of the next generation of finite element computations. In particular, we discuss strategies for exploiting new performance trends, issues of latency and bandwidth, and abstract models for resilient algorithm development. Finally we discuss practical consideration for developing the next generation of library software in this area, including reproducibility, data structures and mixed threading model concerns.

Mark Hoemmen
Title: *Tpetra interface changes to support thread-parallel fill*
Abstract: Tpetra is Trilinos' next-generation sparse linear algebra package. It provides sparse graphs and matrices and dense vectors, and has a parallel data redistribution facility which applications can use. Tpetra lets users choose the type of values in its matrices and vectors, has been demonstrated to solve problems with well over two billion unknowns, and supports "hybrid" MPI + X parallelism for several different shared-memory parallel programming models X. This poster will show our work in progress to improve Tpetra's support for thread-parallel fill. By "fill," we mean constructing and modifying Tpetra data structures, like sparse matrices and dense vectors, as for example in finite element assembly. This work builds on the new Kokkos thread-parallel programming model, but does not require that applications use Kokkos. Our interface changes

will help applications gradually adopt threads, and guide application developers with performant idioms that support different data structure fill patterns.

Ken Franko
Title: *MiniAero*
Abstract: Kokkos was used to develop a mini-application for gas dynamics applications, miniAero. miniAero is an explicit cell-centered finite-volume code that is MPI enabled and uses Kokkos for thread and GPU execution of kernels. Performance numbers for MPI+X for a variety of platforms will be presented along with lessons learned.

Rob Kirby
Title: *Fine-grained finite element parallelism*
Abstract: This poster will demonstrate available concurrency in elementwise finite element kernels, as well as using expressing certain global operations in terms of shared-memory primitives. Preliminary numerical results will be given using PyOpenCL.

Andreas Kloeckner
Title: *Operator transformation and code generation for FEM*
Abstract: The present talk and poster discuss three software components designed to ease and automate tasks encountered in FEM assembly. The first, named 'pymbolic', is an expression tree with extensive traversal and rewriting capabilities. Both its mathematical vocabulary and its traversal operations are easily extended. This functionality is demonstrated in action in the context of operator description and transformation for discontinuous Galerkin (dG) FEM and high-order integral equation codes, with special attention paid to the transformation pipeline implemented and the design constraints imposed by each environment. A brief mention is made of 'PyOpenCL' that, in addition to providing a friendly interface to heterogeneous, shared-memory parallel computing hardware, incorporates an array container and implementations of a variety of parallel primitives, including scan, sort, and reduction. Making use of these foundations, a generic code generator is shown. 'Loo.py' targets shared-memory, massively parallel machines, and based upon a mathematical description of a computation along with a sequence of transformations, generates efficient, low-level code. Its use is shown in the context of FEM assembly and dG operator evaluation. All tools are hosted in the Python programming language, which, by its design, enables and encourages reuse, abstraction, and modularization. The tools are available under the MIT license and straightforwardly incorporated into user code.

Tzanio Kolev, Veselin Dobrev, Michael Kumbera, Robert Rieben
Title: *Scalable high-order finite elements with MFEM, hypre and BLAST*
Abstract: The finite element method (FEM) is a powerful discretization technique that can utilize general unstructured grids to approximate the solutions of many PDEs. High-order finite elements, in particular, are ideally suited to take advantage of the changing computational landscape, because their order can be used to tune the performance, by increasing the FLOPs/bytes ratio, or to adjust the algorithm for different hardware. In this poster we present our work on scalable high-order finite element software that combines the modular finite element library MFEM [1], the hypre library of scalable linear solvers [2], and the high-order shock hydrodynamics research code BLAST [3]. We will first discuss the finite element abstractions provided by MFEM, which include arbitrary high-order H1-conforming, discontinuous (L2), H(div)-conforming, H(curl)-conforming and NURBS elements, defined on general high-order meshes. We will then explain how the MPI-based version of MFEM uses data structures and kernels from the hypre library to enable scalable finite element assembly in parallel. Finally, we will describe the efficient implementation of high-order force matrices in the MFEM-based BLAST application, where we will also demonstrate the benefits of our approach with respect to strong scaling and GPU acceleration.
[1] https://mfem.googlecode.com
[2] https://www.llnl.gov/casc/hypre

[3] https://www.llnl.gov/casc/blast This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, LLNL-ABS-652336.

David Moulton, Ethan Coon, Markus Berndt, Rao Garimella
Title: *Amanzi and the Arctic Terrestrial Simulator: Flexible Multiphysics Simulators for Environment and Ecosystem Applications.*
Abstract: Climate and environmental simulations present a rich set of challenges for multiphysics and multiscale tools. The Advanced Simulation Capability for Environmental Management (ASCEM) program is tasked with addressing these challenges for the effective and defensible cleanup and closure of legacy nuclear waste sites. ASCEM initiated development of a flexible and extensible multiprocess simulator, dubbed Amanzi, as part of its open-source suite of tools. This simulator provides a flow and reactive transport capability on general unstructured polyhedral meshes using Mimetic Finite Difference (MFD) discretizations, services from Trilinos, and multigrid solvers from HYPRE. Its capabilities include variably saturated flow and reactive transport, including a wide range of chemical reactions. Recently, it's modular and flexible design was leveraged and extended in a project to model the climate impacts of a warming arctic through changes in microtopography and its coupling to the hydrology. This extension, the Arctic Terrestrial Simulator (ATS), is significant because the sheer number of processes that are coupled (potentially tightly coupled) defies hand coding and manual testing. Current capabilities include coupled surface and subsurface thermally dependent flows, along with a surface energy balance model including snow. In this poster we highlight the design and implementation approach we used to represent this complex system. Specifically, we refer to the mathematical description of a process as a process model, and its discrete representation as a process kernel (PK). We use a hierarchical representation of the complete system as a graph of PKs with a hierarchy of couplers (the Multi-Process couplers or MPCs). This PK/MPC graph provides a natural structure for the system, and hence we create a discrete distributed vector, a tree vector, that mimics this structure for use by solvers and time integrators. We developed a dynamic data manager, represented as a directed acyclic graph (DAG), to create complex models at runtime and manage the dependencies of their variables. To support an accurate representation of the physical model including polygonal ground, troughs, pinch outs, and ice wedges, we use the MFD method with polyhedral meshes using the MSTK mesh infrastructure. The challenge created by MFD methods is the need for scalar degrees of freedom on the faces of mesh elements. This leads to a block system of cell-based and face-based unknowns, even in scalar models such as thermal energy. To mimic this structure we create composite vectors, which are used naturally as leaves of the tree vector. These abstractions and structures provide flexible building blocks, and are collected in a package named Arcos. We are now beginning to explore performance and optimization. This includes investigating both the local assembly of element matrices as well as the assembly of the complete global system in more general matrices. The lack of a block interface to the HYPRE AMG solver leads to the explicit creation of Schur complements or the copying of block matrices into a unblocked form. Moreover, we have been focused on MPI based parallelism to this point and are now beginning to investigate threading options as well. Here we will demonstrate existing capabilities of Arcos and its use in Amanzi and ATS, and highlight the challenges and potential for future development of these codes.

Shawn Pautz, Clif Drumm, Wesley Fann, Bill Bohnhoff
Title: *Matrix Assembly Tasks in the Sceptre Deterministic Radiation Transport Code*
Abstract: The Sceptre radiation transport code implements discretizations of two different forms of the linear Boltzmann transport equation. Solvers for these discretizations are divided into two different classes. In one class of solvers the full matrix is formed, which is then solved with either CG or GMRES. In the other class we use a wavefront sweep algorithm to solve a block-lower triangular system, which allows assembly of numerous small on-node systems when needed. We describe the various operations that we perform in order to create either type of linear system.

Roger P. Pawlowski, Eric C. Cyr, Eric T. Phipps, and Andrew G. Salinger
Title: *Template-based Generic Programming Techniques for Finite Element Assembly*
Abstract: Modeling and simulation are used to understand, analyze, predict, and design increasingly complex physical, biological, and engineered systems. Because of this complexity, significant investments must be made, both in terms of manpower and programming environments, to develop simulation capabilities capable of accurately representing the system at hand. At the same time, modern analysis approaches such as stability analysis, sensitivity analysis, optimization, and uncertainty quantification require increasingly sophisticated capabilities of those complex simulation tools. Often simulation frameworks are not designed with these kinds of analysis requirements in mind, which limits the efficiency, robustness, and accuracy of the resulting analysis.

In this work, we describe an approach for building simulation code capabilities that natively support the requirements of many types of analysis algorithms. This approach leverages compile-time polymorphism and generic programming through C++ templates to insulate the code developer from the need to worry about the requirements of advanced analysis, yet provides hooks within the simulation code so that these analysis techniques can be added later. The ideas presented here build on operator overloading-based automatic differentiation techniques to transform a simulation code into one that is capable of providing analytic derivatives. However we extend these ideas to compute quantities that aren't derivatives such as polynomial chaos expansions, floating point counts, and extended precision calculations. The capabilities in this work have been released in the open-source Trilinos packages Sacado, Stokhos and Phalanx.

Eric Phipps, H. Carter Edwards
Title: *Improving PDE Assembly Performance Through Embedded Uncertainty Quantification*
Abstract: Achieving high performance for PDE assembly on emerging multicore architectures (such as GPUs, multi-core CPUs, and many-core accelerators) is often difficult due to memory access and code design patterns that are not commensurate with architectural capabilities. These architectures require accesses of wide regions of contiguous memory to achieve good performance, which is often challenging for PDE assembly on unstructured meshes. Furthermore, Intel-based architectures require consistent vectorization to achieve good performance, which is difficult for complex PDE codes. To address these issues, we explore opportunities for improving memory access patterns and vectorization by simultaneously propagating ensembles of PDE samples relevant to uncertainty quantification. Here we leverage the fact that memory access patterns and instructions are often very similar for PDE evaluations across samples in an uncertainty quantification calculation. We use template-based generic programming techniques to replace each scalar in the PDE assembly with a small array tuned to the natural vector length of the architecture, and organize data structure layouts so that data corresponding to each sample instance are stored contiguously in memory. The performance and scalability of this approach will be investigated on a variety of contemporary multicore architectures.

Kendall Pierson, Micah Howard, Michael Tupek
Title: *Efficient Block Sparse Assembly with SIMD*
Abstract: High Mach fluid regimes are critical environments to simulate, understand, and predict for the NW mission. Conchas is our high Mach application code built upon the Sierra toolkit, a custom block compressed sparse row data structure and a native point-implicit solver. This work describes the transformation of the data structures to take advantage of SIMD instructions to improve current performance through vectorization which is a necessary step towards multi-core, GPU, and next-generation platforms.

Onkar Sahni
Title: *Abstractions and algorithms for adaptive methods on boundary layer meshes*
Abstract: A set of tools and techniques are presented for general unstructured meshes with a focus on adaptive methods for boundary layer meshes. Such meshes are useful, for example, in wall-bounded turbulent flows that require tightly controlled mesh spacing and structure near the walls.

An adaptive approach for such meshes must maintain highly anisotropic, graded, and layered elements near the walls while error estimators or indicators must incorporate the structure of the flow boundary layer and associated physics. Similarly, parallel procedures must account for mixed element types, i.e., in mesh modifications and dynamic load balancing. We present abstractions and algorithms that address these needs. We also present high-order discretization techniques for boundary layer meshes including use of higher interelement continuity in the wall-normal direction.

Roy Stogner
Title: *C++14 Generic Programming as a Domain-Specific Language for PDEs*
Abstract: Abstractions and techniques are shown for employing expression template class hierarchies in C++ to provide users with a natural way to express physics kernels and solve Initial Boundary Value Problems. Basic compile-time metaprogramming is used to construct an API which recasts PDE expressions in a syntax which is valid C++ yet also natural to write. Topics include the use of expression templates to generate GPGPU code and automatically differentiated Jacobian matrices, the use of C++14 return type deduction to enable kernel fusion within a modular code, and the use of generic programming to maintain flexibility of design and ease of debugging. Challenges relating to optimization, hybrid meshes, and mesh adaptivity will be discussed.

Daniel Sunderland, H. Carter Edwards
Title: *Thread Scalable CRS Graph Construction*
Abstract: Our portable thread scalable pattern for CRS graph construction consists of four simple steps: 1) parallel counting the non-zeros, 2) allocating storage, 3) parallel filling, and 4) parallel post-procssing each row. Counting the non-zeros can be one of the more difficult algorithms to correctly implement in a scalable way. We demonstrate a simple solution for parallel counting which uses a Kokkos UnorderedMap to achieve good scalability. We also show how the Kokkos UnorderedMap implements a portable, scalable, and lock-free insert.

James Sutherland, Matthew Might, Tony Saad, Christopher Earl, Abhishek Bagusetty
Title: *Flexible, Efficient Abstractions for High Performance Computation on Current and Emerging Architectures*
Abstract: Complexity for large-scale simulation software stems from two primary sources: the physics being simulated and the language abstractions for various hardware targets. Multiplicity of physical modeling options, each of which may introduce unique nonlinear coupling and dependencies, can create rigid, fragile software that isn't easily maintained or modified. Changes in hardware (e.g., multicore or GPU architectures) can require different computational kernels to be maintained for each hardware target. Handling these two general challenges together to produce efficient, scalable software can be a daunting challenge. This poster discusses two abstractions that work in tandem to address the aforementioned challenges. First, the software is written to represent nodes that can be self-assembled into a directed, acyclic graph (DAG) which exposes the structure of the calculation. This facilitates automated scheduling of nodes in the DAG, and reasoning about efficient management of CPU and GPU. Second, a domain-specific language, embedded in C++, is under development to allow the application programmer to specify high-level intent while allowing highly efficient back-ends targeting various hardware (CPU, GPU) to be generated at compile time. These two abstractions combine to create a powerful environment where application developers can increase productivity and deploy complex software across a variety of hardware environments.

Christian Trott
Title: *A migration strategy for utilizing the Kokkos many-core programming model*
Abstract: In order to support many-core architectures in Trilinos many packages have started to explore the utilization of Kokkos. Here a migration strategy will be presented for an incremental transition to using Kokkos, starting with simple thread-parallelism, continuing with GPU support and finishing with two and three-level parallelism employing thread teams and vectorization. A particular focus is put on software which already uses Tpetra, Trilinos' next-generation sparse linear

algebra package.

Bruno Turcksin, Martin Kronbichler, Wolfgang Bangerth
Title: *Multithreaded matrix assembly for finite elements*
Abstract: We present a design pattern that can be applied to any operation requiring to be done independently on every cell and which is followed by a reduction of the local result into a global data structure. This design pattern can be directly applied to multithreaded matrix assembly and implemented using the parallel pipeline design pattern. When assembling a global matrix for finite elements, a local matrix is assembled on each cell; this step is embarrassingly parallel. However, when the local matrices are incorporated into the global matrix, it is necessary to ensure that two processors do not attempt to write simultaneously in the same global matrix element. To prevent this, a colorization algorithm is used before the reduction operation; all the elements of a given color can be simultaneously written into. It is important for the colorization algorithm to produce few colors, but it is more so that the size of these colors are similar; small colors would degrade the scalability of the algorithm. This design pattern was implemented in the deal.II library and was shown to significantly speed up matrix assembly

Tim Warburton, David Medina, Amik St-Cyr
Title: *OCCA: A Unified Approach to Multi-Threading Languages*
Abstract: Currently there are a number of relatively popular APIs for multi-threading programming including but not limited to CUDA, OpenCL, OpenACC, and OpenMP. Initially it might appear that many-core programming forces programmers to lock into a specific API. Additionally simulation codes, frameworks, and libraries have lifetimes measured in decades that might outlive a specific API. To address these issues we developed the lightweight OCCA API in a way that allows a programmer to write single source kernel implementations that are portable and can be dynamically compiled and executed at run-time as CUDA, OpenCL, or OpenMP. Example performance results from our OCCA based finite difference, discontinuous Galerkin, and spectral element method based PDE solvers will show that it is possible to develop efficient and portable many-core code for CPUs and GPUs.

Ulrike Yang, Rob Falgout, Tzanio Kolev, Jacob Schroder
Title: *Matrix and Vector Assembly in hypres Conceptual Interfaces*
Abstract: The hypre software library provides high performance preconditioners and solvers for the solution of large sparse linear systems on massively parallel computers. One of its attractive features is the provision of conceptual interfaces, which include a structured, a semi-structured, and a traditional linear-algebra based interface. These interfaces give application users a more natural means for describing their linear systems, and provide access to methods such as structured multigrid solvers, which require additional information beyond just the matrix. We discuss the assembly of matrices and vectors within the various interfaces in hypre as well as current efforts to increase the use of OpenMP threads in the interfaces.


Sandia National Laboratories