The Journal of Systems and Software 125 (2017) 389–400

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Nebo: An efficient, parallel, and portable domain-specific language for numerically solving partial differential equations

Christopher Earl[a,*], Matthew Might[b], Abhishek Bagusetty[c], James C. Sutherland[b]

[a] Lawrence Livermore National Laboratory, United States
[b] University of Utah, United States
[c] University of Pittsburgh, United States

ARTICLE INFO

Article history:
Received 15 May 2015
Revised 1 January 2016
Accepted 12 January 2016
Available online 26 January 2016

Keywords:
Domain-specific language embedded in C++
GPGPU

ABSTRACT

This paper presents Nebo, a declarative domain-specific language embedded in C++ for discretizing partial differential equations for transport phenomena on multiple architectures. Application programmers use Nebo to write code that appears sequential but can be run in parallel, without editing the code. Currently Nebo supports single-thread execution, multi-thread execution, and many-core (GPU-based) execution. With single-thread execution, Nebo performs on par with code written by domain experts. With multi-thread execution, Nebo can linearly scale (with roughly 90% efficiency) up to 12 cores, compared to its single-thread execution. Moreover, Nebo's many-core execution can be over 140x faster than its single-thread execution.

1. Introduction

To avoid inefficiencies, most high-performance computing (HPC) code is written at a very low level. However, with the rise of new architectures, such as multi-core CPUs and GPUs existing code must be rewritten for each new architecture, which is a labor-intensive and error-prone process that also creates a maintenance challenge.

This paper describes Nebo, an efficient domain-specific language (DSL) embedded in C++[1], the purpose of which is to enable domain experts to create code that is efficient, scalable, and portable across multiple architectures. Nebo is a declarative DSL for numerically solving partial differential equations for transport phenomena such as computational fluid dynamics on structured meshes. The fundamental unit of variable abstraction in Nebo is a **field**, which represents the value of a variable at all points on the mesh.

Nebo was designed for use in high-performance simulation projects such as Wasatch, which is a component within the Uintah Berzins et al. (2012); Parker (2002); Berzins et al. (2010) framework and has demonstrated scalability to 262,000 cores Schmidt et al. (2013a). Wasatch is a code for convection–diffusion–reaction problems, and focuses on turbulent reacting flow simulations using large eddy simulation. Uintah is a set of libraries and applications for simulating and analyzing complex chemical and physical reactions. While this paper discusses Nebo's use in Wasatch, Nebo is a stand-alone library and is used in other projects (see, e.g., Punati et al. (2011)). Nebo handles data parallelism but leaves memory management and data transfers between CPU and GPU either to a framework or to the end user. Both Wasatch and Uintah provide users with easy to use options to manage memory and data transfers. If Nebo is used outside of Wasatch and Uintah, either users can manage these tasks themselves or use other software support for them.

Because many current HPC codes are written in C++, Nebo is embedded within C++ to allow incremental adoption; when Nebo lacks needed functionality, domain experts are able to prototype the code natively in C++. Then, when new Nebo functionality becomes available, domain experts rewrite code in Nebo that is more flexible and easier to maintain than the original. Our experience is that code that uses Nebo is frequently more efficient than the code hand-written by the domain experts, and can be deployed on both CPU and GPU. Furthermore, since Nebo and the existing application code are both written in C++, refactoring existing C++ code into Nebo syntax is relatively straightforward.

To simultaneously achieve expressiveness, efficiency and portability, Nebo separates *what* computation should be performed from *how* that computation should be done. Nebo has a restrictive declarative syntax so that the computation can be represented as an abstract syntax tree (AST) within the C++ template system. From the AST representing the Nebo calculation, Nebo generates

* Corresponding author. Tel.: +19254237697.
E-mail addresses: earl2@llnl.gov (C. Earl), might@cs.utah.edu (M. Might), abb58@pitt.edu (A. Bagusetty), James.Sutherland@utah.edu (J.C. Sutherland).

[1] Nebo targets the 1998 standard of C++.

http://dx.doi.org/10.1016/j.jss.2016.01.023
0164-1212/

efficient code for a variety of backend implementations. Nebo supports three major backends: A single-thread (sequential) backend, a multi-thread backend, and a GPU-based backend. Moreover, Nebo's semantics are intensionally restricted, which limits what can be computed within Nebo. For example, all Nebo code will terminate because Nebo does not recurse and all loops in Nebo iterate a fixed number of times (based upon runtime parameters). Finally, Nebo calculations write results to a finite amount of mutable memory a fixed number of times (once per Nebo assignment). By avoiding Turing-Completeness, Nebo is focused and optimized for its domain.

Additionally, Nebo is intentionally limited in its capabilities for its domain: Nebo does not provide memory management, task parallelism, or inter-node communication (such as MPI). For these capabilities, Nebo is intended to be used with other libraries/frameworks for HPC applications, such as the Uintah Berzins et al. (2012); Parker (2002); Berzins et al. (2010) framework.

Nebo's single-thread backend performs at least as well as the hand-written code it replaces. With computationally intensive calculations, Nebo's multi-thread backend can scale linearly to the number of cores available, and Nebo's many-core (GPU) backend can perform 140 × faster than Nebo's single-thread backend. With less computationally intensive calculations, Nebo's parallel backends do not scale as well, mainly because of memory latency. That said, practical uses of Nebo are computationally intensive enough that these limits of Nebo rarely arise.

Nebo is available for download, as part of the SpatialOps project, using git from: https://software.crsim.utah.edu:8443/James_Research_Group/SpatialOps.git

Nebo's most recent documentation can be built from the source code using doxygen or viewed from: https://software.crsim.utah.edu/jenkins/job/SpatialOps/doxygen/

After discussing Nebo's syntax and semantics in Section 2, Section 3 discusses the technical details of Nebo's implementation. Section 4 contains case studies of real uses of Nebo, which are taken directly from Wasatch. This section also contains performance results from these uses of Nebo for all of Nebo's backends as well as performance comparisons with other components of Uintah.

2. Syntax and semantics of Nebo

This section explains Nebo's syntax, semantics, and some information about how specific features of Nebo are implemented. The next section focuses on Nebo's overall implementation and details about how the backends work.

Because Nebo is a domain-specific (rather than general-purpose) language for numerically solving PDEs in high-performance simulations, Nebo's syntax and semantics are limited and it is not Turing complete. Each assignment statement in Nebo is roughly analogous to a mathematical operation over fields. A field is a one-, two-, or three-dimensional array, which is explained in more detail in Section 2.1.

Because Nebo is embedded within C++ Stoustrup (1997), standard C++ compilers parse Nebo code without modification. We view this as an advantage since C++ is ubiquitously supported on high performance computing architectures and we can leverage existing compilers rather than developing and maintaining a separate one. Because Nebo is limited to C++'s basic syntax, it uses C++-style function call syntax, C++-style operator syntax, and only the operators that can be overloaded within C++. Nebo also inherits its syntax, operators, and operator precedence that is well-defined, well-documented, and well-understood by C++ programmers. Furthermore, Nebo maintains the semantic meaning of the operators it overloads, lifted over fields. For example, addition of two fields represents the pointwise addition of the elements with those two fields. The sole exception to Nebo maintaining the semantic meaning of its overloaded operators is its assignment operator, which is discussed in Section 2.3.

The Nebo semantics define what calculation a Nebo Expression denotes, but not the order of evaluation over all elements in the mesh. The calculation of a given Nebo Expression is the same for all valid elements in the fields involved. Because the semantics do not define an ordering, each backend may choose a different order of execution that is specific to the targeted architecture. Nebo is tuned to choose an order of execution that benefits the most from architecture-specific capabilities and that avoids synchronization and communication. The details of the backends are discussed in Section 3.2.

Readers familiar with MATLAB or Fortran90 array syntax will recognize similar patterns in Nebo's syntax. Nebo uses fields as MATLAB uses matrices and Fortran uses arrays: Expressions and assignments in Nebo can be calculated over fields, without needing to provide indices or explicit loop structures.

The rest of this section is laid out as follows: Section 2.1 discusses the definition and declaration of fields, Nebo's array type. Section 2.2 discusses basic Nebo Expressions and operations. Section 2.3 discusses Nebo assignment. Section 2.4 discusses conditional expressions within Nebo. Section 2.5 discusses stencil operations in Nebo, which are the only way to perform nonpointwise calculations in Nebo. Finally, Section 2.6 discusses Nebo reductions.

2.1. Field definition and usage

A field is a structured three-dimensional array. One- and two-dimensional arrays can defined by setting the extents of the unused dimensions to 1. To build a new field, the most basic information required is a memory window, boundary cell information, and ghost cell information. Optionally, if the field is to be built using existing memory (allocated and initialized in non-Nebo code), a pointer to the memory can be provided. If a pointer to existing memory is provided, a flag for external storage needs to be set, so the field does not free the memory. Also, optionally, a device flag (defaults to CPU) can be set to specify where the memory should be allocated (for internal allocation) or where the memory already was allocated (for external allocation).

Each field represents a single quantity over an array of elements that can represent volumes or faces. Moreover, an single field may represent the entire simulation space, or it may only represent some partition of the simulation space.

Section 2.1.1 describes what a memory window is, what one contains, and how to create one. Section 2.1.2 describes what boundary information is needed to create a field. Section 2.1.3 describes what boundary information is needed to create a field. Section 2.1.4 describes how users can manage the memory of fields across CPUs and GPUs. Additional information on fields and their components can be found in the documentation for Nebo, which can be found either in the source code or at the online documentation at: https://software.crsim.utah.edu/jenkins/job/SpatialOps/doxygen/

2.1.1. Memory windows

A memory window is a logical subset, or window, of array elements. The elements within a memory window are logically adjacent but are not necessarily a single contiguous subarray of the larger array. A memory window is made up of three components: The global extents of the entire array, the offset to the first element of the window, and the extents of the window itself. Each of these components is represented as an array of three integers.

The global extents represent number of elements in each dimension (x, y, and z) of the entire underlying contiguous allocated

block of memory. The global extents are used by Nebo to determine where and how many elements to skip when iterating over the elements of a memory window.

The offset represents the index (in *x*-, *y*-, and *z*-coordinates) of the first element in the memory window. With the global extents and the offset together, Nebo can determine the flat (global) index of the first element of the window.

The local extents represent the number of elements in each direction that the window contains. With the global extents, offset, and local extents together Nebo can determine when to skip elements and how many to skip, when iterating over the elements of the window.

2.1.2. Boundary cells

Fields can represent different types of spatial values: Some represent quantities of volumes (temperature, mass, etc.); others represent quantities of the faces between volumes (flux, difference in temperature, etc.). Fields can also represent staggered meshes, that is structured meshes partially offset from one another. (More information about the different types of fields can be found in Nebo's documentation.)

Boundary cells are extra cells on some meshes to finish out needed faces or volumes. For example, consider a field of faces between volumes, where the faces are normal to the *x*-direction (ignoring the faces normal to the *y*- and *z*-directions). By convention in Nebo, the negative face (the face on shared with the lower index neighbor element) is associated with a volume and shares its index. The positive face (the face on shared with the higher index neighbor element) is associated with its neighbor volume and shares its index. However, the volume with highest x-index by definition has no neighbor on its positive side; however, that volume still needs a positive face. To provide this extra face, boundary cells are used.

Three boolean values are used to indicate whether or not a mesh has an extra boundary cell. (True indicates that an extra cell is present.) Not all types of fields may need extra boundary cells at the positive end of the simulation space. (More information about the different types of fields and their various boundary cell possibilities can be found in Nebo's documentation.)

2.1.3. Ghost cells

Multiple fields can combine to represent a single quantity for the entire simulation space. Usually, this happens when the simulation is distributed over a cluster using MPI. Ghost cells are extra cells added along faces of the field shared with a neighboring field. Data from the edge of a field can be copied into the ghost cells of the field that logically neighbors it and vice versa, over MPI. Nebo allows for fields to have an arbitrary number of ghost cells along all six faces of the field. However, most current users of Nebo are limited to a single ghost cell because of design constraints predating the use of Nebo.

The ghost cell information needed to construct a field is the number of ghost cells on each face, which is six nonnegative integers. The current implementation of fields allows for shortcuts when all faces have the same number of ghost cells.

Additionally, fields keep track of which ghost contain valid values and which have been invalidated. Section 2.5 discusses when and how ghost cells can be invalidated. Ghost cell validation counts can also be reset manually.

2.1.4. Multiple device locations of a field

While a field is initially created on a single device (CPU or GPU), it can be expanded to multiple devices. For example, consider the field, *f*, initialized on the CPU. To copy this field to the GPU, call the following function:

```
f.add_device(GPU_INDEX);
```

This command checks to see if memory has already been allocated on that device, and if not, allocates the proper amount of space. After checking that memory is allocated on the correct device, this command checks to see if the data on that device has been marked as invalid/out of date. If the data has been marked as invalid, it copies data from the primary/active device (see below) to the specified device. Once the copy has completed, the device memory is marked as valid/current. There is also an asynchronous version of this command:

```
f.add_device_async(GPU_INDEX);
```

The data in a field can with memory allocated on multiple devices can be read from all devices whose data is marked as valid. (Attempting to read a field on a device without valid data throws a runtime exception.) The data in a field can with memory allocated on multiple devices can be overwritten only from the primary/active device. (Attempting to write to a field on a device that is not the primary/active device throws a runtime exception.) When data is overwritten, all non-active devices have their data marked as invalid, and require revalidating the data on those devices (recopying).

Only one device can be primary/active at a time. The active device can be changed at any time with either of the following commands:

```
f.set_device_as_active(GPU_INDEX);
f.set_device_as_active_async(GPU_INDEX);
```

This system of one active device and (potentially) multiple valid devices simplifies memory management. Invalidation happens automatically when data is accessed by a non-constant method. Data transfers between CPU and GPU either direction happen through a single function with a single parameter (target device). While Nebo users (developers) are required to explicitly handle data transfers, the current implementation of fields provides a simple API for doing so. (The complete API is in Nebo's documentation.)

To handle the asynchronous function calls correctly, every field contains a unique CUDA stream. Data transfers and Nebo statements both use the CUDA streams in the internal calls to CUDA functions. For example, consider fields, *f* and *g*, both initially created on the CPU. The following code will transfer both to the GPU, and execute the assignment statement only after both transfers have completed:

```
f.set_device_as_active_async(GPU_INDEX);
g.add_device_async(GPU_INDEX);

f <<= g * 2.0;
```

2.2. Basic Nebo expressions

Expressions are the basic abstraction of Nebo. Nebo Expressions represent calculations, not values, as is generally expected of expressions. Nebo Expressions can be used in field assignment and reductions (see Sections 2.3 and 2.6, respectively).

A Nebo Expression can be: a scalar value; a field; the valid use of supported operators and functions (below), whose arguments themselves are Nebo Expressions; a conditional expression, which is discussed in Section 2.4; or a stencil operator applied to a Nebo Expression, which is discussed in Section 2.5. Nebo Expressions support the following operations and functions: algebraic

operators (addition [$\bullet + \bullet$], subtraction [$\bullet - \bullet$], multiplication [$\bullet * \bullet$], division [$\bullet / \bullet$], and negation [$-\bullet$]); trigonometric functions (sine [`sin(•)`], cosine [`cos(•)`], tangent [`tan(•)`], and hyperbolic tangent [`tanh(•)`]); extremum functions (minimum [`min(•, •)`] and maximum [`max(•, •)`]), and other mathematical functions (exponentiation with base *e* [`exp(•)`], exponentiation with given base [`pow(•, •)`], absolute value [`abs(•)`], square root [`sqrt(•)`], and natural logarithm [`log(•)`]). Nebo provides support for these operators and functions through operator (and function) overloading and template meta-programming, and the set of supported functions is easily extensible. Examples of Nebo expressions appear throughout this paper, beginning with the next section.

2.3. Assignment

Because Nebo calculates the discretized results of partial differential equations, Nebo assignments keep syntax very close to the mathematical expressions. Field assignment is the primary use of Nebo Expressions, which is comparable to a foreach operation or Lisp's map operation. With field assignment, Nebo Expressions produce a field (array) of values, which are the values used in the assignment. Nebo uses `operator <<=` for assignment instead of `operator =` because using `operator <<=` makes it explicit where Nebo overloads assignment. This assignment operator is the only operator that Nebo has changed the semantic meaning from the semantics of C++, other than lifting the operations over fields.

As a concrete but simple example of Nebo assignment, consider the following equation, where *a, b*, and *c* are fields:

$$c = a + sin(b)$$

Without Nebo, this equation could be calculated with the following code, which is similar to what Wasatch developers would write before they started using Nebo:

```
Field a, b, c;
//...
Field::iterator ic = c.begin();
Field::iterator const ec = c.end();
Field::iterator ia = a.begin();
Field::iterator ib = b.begin();
while(ic != ec) {
    *ic = *ia + sin(*ib);
    ++ic;
    ++ia;
    ++ib;
}
```

With Nebo, this same equation can be calculated by:

```
Field a, b, c;
//...
c <<= a + sin(b);
```

which deploys on single- or multi-thread CPU as well as GPU.

2.4. Conditional expressions

The conditional statement `if` and the ternary operator ($\bullet$? $\bullet$: $\bullet$) cannot be overloaded in C++. Thus, to have pointwise conditional evaluation over fields, Nebo introduces `cond`, as used in many functional languages (introduced by LISP). Fortunately, `cond` fits into Nebo's syntax through C++ operator overloading and template meta-programming, which Nebo already exploits. Additionally, through the use of inlined templated functions, `cond` compiles down to nested ternary operators ($\bullet$? $\bullet$: $\bullet$). Thus, while the templates for `cond` are not simple, the executed code is simple and efficient.

For conditional expressions to be useful, expressions must express boolean values, which are provided by Nebo Boolean Expressions. Nebo Boolean Expressions are similar to Nebo Expressions in that they represent calculations, not values. Unlike Nebo Expressions, which produce scalar values when evaluated, Nebo Boolean Expressions produce boolean values when evaluated. A Nebo Boolean Expression can be: a boolean value; the numeric comparison of two Nebo Expressions, using any of the C++ numeric comparison operators ($\bullet == \bullet$, $\bullet\ != \bullet$, $\bullet < \bullet$, $\bullet > \bullet$, $\bullet <= \bullet$, and $\bullet >= \bullet$); or a logical connective of Nebo Boolean Expressions, using any of the C++ logical connective operators ($\bullet$ && $\bullet$, $\bullet$ || $\bullet$, and $!\bullet$).

The requirements for `cond` are strict: Every non-final clause must contain exactly two arguments, and the last clause must contain exactly one argument. The second argument of each non-final clause and the single argument of the final clause must be a valid Nebo Expression. The first argument of each non-final clause must be a Nebo Boolean Expression.

The semantics of `cond` mimic nested ternary operators ($\bullet$? $\bullet$: $\bullet$), lifted pointwise over fields. For each point, the conditional expression returns the value associated with the first true Nebo Boolean Expression. If none of the Nebo Boolean Expressions evaluate to true the conditional expression returns the value of the final clause.

For a concrete example, consider the following code:

```
bool d; Field a, b;
//...
Field::iterator ib = b.begin();
Field::iterator const eb = b.end();
Field::iterator ia = a.begin();
while(ib != eb) {
  if(*ia > 0.0) *ib *= *ia;
  else if(*ia < 0.0) *ib *= -(*ia);
  else if(d) *ib *= *ib;
  ++ib; ++ia;
}
```

With Nebo, this code can be rewritten as:

```
bool d; Field a, b;
//...
b <<= b * cond(a > 0.0, a)
               (a < 0.0, -a)
               (d, b)
               (1.0);
```

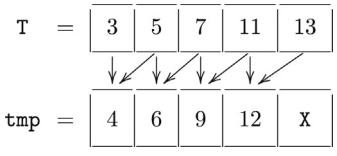
2.5. Stencil operations

Stencil calculations arise from interpolation as well as discrete calculus operations in the solution of partial differential equations. In Nebo, stencils shapes are fixed at compile time while coefficients can be determined at runtime.

Consider the following 1-dimensional stencil example, which calculates an approximation of a derivative of field `T` and uses traditional C array access (`array[•]`) for clarity:

```
int total;  Field1 T;  Field2 tmp;
//...
for(int cur=0; cur<total; cur++)
  tmp[cur] = 0.5 * (T[cur] + T[cur+1]);
```

Graphically, with arbitrary values in field `T`, this gradient calculation looks like:



The values in field `tmp` come directly from field `T`: $0.5 * (3 + 5) = 4$; $0.5 * (5 + 7) = 6$; $0.5 * (7 + 11) = 9$; and $0.5 * (11 + 13) = 12$.

If `T` and `tmp` are the same size, the last cell of field `tmp` does not contain a valid value because there is no further cell in field

T. With any stencil there are edge cases that cannot be computed. There are various ways to handle such edge conditions. Wasatch uses "ghost cells," a layer of cells surrounding the fields on all sides to handle these boundary cases. Nebo supports usage of ghost cells, and uses them as necessary. From type introspection on operators, Nebo can determine how many ghost cells cannot be filled by the application of an operator. This ghost cell information is retained on a field through subsequent Nebo operations to ensure that fields do not exhaust their supply of valid ghost cells at runtime. Once ghost cells are invalidated (left unfilled), application-specific methods are used to revalidate them.

As another example, consider $\phi = \nabla \cdot \nabla T$. To remain as close as possible to the natural mathematical syntax and to allow loop fusion, Nebo uses function application,

```
Field1 T, phi;
//...
phi <<= Div(Grad(T));
```

Nebo also supports two- and three-dimensional stencils with the same syntax.

2.6. Reductions

Reductions, such as calculating the sum of all elements and finding the max value in a field, use Nebo Expressions as their input. Reductions, along with field assignment, are currently the only uses of Nebo Expressions. These reductions act much like MapReduce Dean and Ghemawat (2008), where the calculation of the Nebo Expression is the map step, and the reduction operation (sum, max, etc.) is the reduce step. Thus, Nebo reductions produce a single scalar value. Currently, Nebo supports the following reductions directly: min, max, sum, and L_2 norm.

For example, consider the following expression involving fields a and b:

$\mathtt{sum}(a + sin(b))$

Without Nebo, this equation could be calculated with the following code:

```
Field a, b;  Scalar sum;
//...
sum = 0;
Field::iterator ia = a.begin();
Field::iterator const ea = a.end();
Field::iterator ib = b.begin();
while(ia != ea) {
    sum += *ia + sin(*ib);
    ++ia; ++ib;
}
```

With Nebo, this same expression can be calculated by:

```
Field a, b;  Scalar sum;
//...
sum = nebo_sum(a + sin(b));
```

In cases where reductions over subsets of fields and/or calculations are needed, conditional expressions (see Section 2.4) can be used.

3. Implementation of Nebo

This section presents the implementation of Nebo in two parts: Parsing (Section 3.1) and the backends (Section 3.2). Section 3.1 discusses how Nebo is parsed into abstract syntax trees through template meta-programming. Section 3.2 discusses how an abstract syntax tree is converted into runnable code for each of Nebo's backends.

3.1. Template meta-programming

The C++ template system is a completely-pure, Turing-Complete functional language. There are several implementations of the lambda calculus in the C++ template system that serve as proofs of concept Unruh (1994); Veldhuizen (1995).

Even though the Turing-Completeness of C++ templates is interesting, meta-programming tools like Nebo rarely use its full capabilities. Unless the compile-time computation affects the generated code, there are more straightforward tools that can compute the same results. Since this system is part of the type system of C++, compile-time computation can remove runtime overhead, by informing the compiler about constant values, inlineable functions, and control flow paths. Thus, using the C++ template system, we can inform the compiler exactly what the subexpressions in a given Nebo Expression are and avoid using virtual lookup tables for functions, which would be used with traditional C++ class inheritance.

Nebo Expressions are template objects, whose template parameters are the types of the expression's subexpressions. Thus, when analyzing the function calls to subexpressions, the compiler knows the specific type and therefore the specific function that will be called to evaluate a given element at runtime. Therefore, the type of a Nebo Expression is an abstract syntax tree (AST) of the calculation that the Nebo Expression is to perform. Consider the following Nebo code:

```
Field a, b;
//...
b <<= 3.14159 + sin(a);
```

Building this AST framework is rather straightforward. C++ operator overloading allows functions and operators to return any type. The addition operator generates an object of type `SumOp<Arg1, Arg2>`, and the `sin` function generates an object of type `SinOp<Arg>`. A simplified version of the right-hand side's return type in the above example is:

```
NeboExpression<SumOp<NeboScalar,
                     SinOp<NeboField> > >
```

We also use this template/type AST approach to generate different backends. Depending on available resources and run-time conditions, Nebo is able to run on a single thread, on multiple threads, or on a GPU. Each backend requires different yet related functionality to run on its target architecture. To keep each backend separate and distinct, Nebo uses another template parameter. Each Nebo Expression has a template parameter for mode. A mode is either a backend or an intermediate step towards a backend. When a use of a Nebo Expression, such as in a Nebo assignment, is executed, an instance of the Nebo Expression AST is constructed in the `Initial` mode. At compile time, all enabled backends are compiled for the Nebo Expression AST with the mode matching that backend. During execution, based upon the current location of memory and runtime options, a single enabled backend is selected and instantiated with the appropriate mode.

Consider again the example from earlier in this section. The type of the Nebo Expression in this assignment starts out as:

```
SumOp<Initial,
      NeboScalar<Initial>,
      SinOp<Initial,
            NeboField<Initial> > >
```

At compile-time, each specific use of Nebo assignment builds an AST for each enabled backend. Each AST for each specific use of Nebo assignment for a specific backend represents a different type. At runtime, Nebo will chose a specific backend to run each time a Nebo assignment statement is executed. (Section 3.2 below describes how a backend is chosen.) When a backend has been

chosen, a new instance of the AST (a specific type) is created. For example, if the single-thread backend is chosen, which uses the `SeqWalk` mode (short for sequential walk), the AST becomes:

```
SumOp<SeqWalk,
      NeboScalar<SeqWalk>,
      SinOp<SeqWalk,
            NeboField<SeqWalk> > >
```

There are a total of five modes: `Initial`, `SeqWalk`, `Resize`, `GPUWalk`, and `Reduction`. The use of each of these modes is discussed in Section 3.2.

3.2. Backends

This section discusses Nebo's single-thread, multi-thread, and GPU backends and how Nebo decides which backend to use during execution. As discussed in Section 3.1, Nebo uses different modes (types) to implement different backends to Nebo. The mode of a Nebo Expression is represented by a template argument. The implementation of a mode for a Nebo Expression is a partial template specialization. Each partial template specialization is a different type and therefore has no restrictions on what it can and cannot contain, beyond the limits of a C++ class. However, by convention in Nebo's implementation, each mode provides a uniform interface. For example, every term's `Initial` mode implements an `init()` method, which returns the same Nebo Expression but in the SeqWalk mode. Thus, Nebo's partial template specializations behave much like C++ classes that have inherited some basic interface. This convention creates a uniform way for Nebo Expression terms to interact with their subexpressions.

Nebo uses a mix of compile-time and runtime parameters to determine which backends to use. For the compile-time flags, Nebo uses the C preprocessor macro `#ifdef` to add or to ignore Nebo's various backends. By default, Nebo only compiles the single-thread CPU backend, and regardless of how flags are set this backend is always available. The thread-parallel and GPU backends are compiled by defining `ENABLE_THREADS` and `ENABLE_CUDA`, respectively. Furthermore, for the GPU backend to be used, the code must be compiled by NVidia's CUDA compiler, nvcc.

At runtime, the `Initial` mode of a Nebo Expression is constructed first. If the Nebo Expression is used in a reduction, the expression switches to the reduction mode, and continues as explained in Section 3.2.4. If the Nebo Expression is used in an assignment, Nebo then chooses which particular backend to use based on choices implicitly made by the user/developer. Assuming all backends are compiled, Nebo first checks the location of the memory for the result field. (The field class contains information about where the memory was allocated through the use of a device index, so Nebo simply checks that device index. By convention, the CPU is -1 and GPU indices match NVidia's device index, which are 0 or greater.) If the memory is located on a GPU, then Nebo uses its GPU backend. If the memory is located on a CPU, then Nebo checks the number of active threads in the thread-pool it uses. If there is more than one active thread, Nebo uses its thread-parallel backend. Otherwise, Nebo uses its single-thread CPU backend. Of course, if a particular backend of Nebo is not compiled, Nebo will skip the check for that backend.

In the case where the memory of different fields is located in different memory spaces, Nebo throws a runtime exception. In theory, Nebo could handle the data transfer between CPU and GPU - and in fact does in a specialized GPU debug build mode. However, one of Nebo's guiding design principles is to make expensive operations explicit to developers using Nebo. In a production environment, an unexpected and silent data transfer is a performance bug, and so Nebo makes that an explicit runtime error with an a detailed and explicit message as to why an exception was thrown.

The reason behind this decision process is to simplify control-flow and data movement: Nebo runs its calculation on the processing unit (CPU or GPU) where the data already is. Because of its scope, Nebo leaves the decisions of how many threads to use and where to allocate memory up to end users. Nebo only considers how to efficiently compute the result of a single Nebo Expression with a given backend. Broader issues, such as available resources, effectiveness of those resources, and how heavily those resources are being used are beyond Nebo's capabilities. Thus, Nebo's design, syntax, and different backends make it easy and simple to change scheduling and memory locality for users and tools that can reason about the above issues.

3.2.1. Single-thread implementation

The `SeqWalk` mode implements Nebo's default single-thread execution backend. The `SeqWalk` mode uses affine loop indices to calculate individual points. The interface has a single function for the right-hand side (Nebo Expression) of an assignment: An `eval()` method which evaluates the Nebo Expression at the current index. The interface has a single function for the left-hand side of an assignment (a `NeboField` object): A `ref()` method which returns a reference to the current element of the underlying field. To execute the assignment, Nebo loops over all valid indices:

```
for(int z = zLow; z < zHigh; z++)
 for(int y = yLow; y < yHigh; y++)
 for(int x = xLow; x < xHigh; x++)
 ref(x, y, z) = rhs.eval(x, y, z);
}
```

For example, consider the single-thread execution of the example from Section 3.1:

```
Field a, b;
//...
b <<= 3.14159 + sin(a);
```

Ignoring the `SeqWalk` mode and the field type template arguments, the type of the left-hand side of this assignment becomes `NeboField`. The type of the right-hand side of this assignment becomes

```
SumOp<NeboScalar,SinOp<NeboField> >
```

Each call to `rhs.eval()`, `SumOp`'s evaluate method, calls the evaluate method on both of its arguments, adds the values from these evaluate calls together, and returns the result. Each call to `NeboScalar`'s evaluate method simply returns its scalar value, which is in this case 3.14159. Each call to `SinOp`'s evaluate method calls the evaluate method on its argument, applies the sine function to the value from this evaluate call, and returns the result. Each call to `NeboConstField`'s evaluate method dereferences the value from the current index and returns that value.

While there are a lot of nested `eval` function calls for each iteration of the above while loop, all of these functions are able to be inlined. Fortunately, these function calls are textbook examples of functions to inline: First, they are short and simple functions. Second, each function is used in exactly one location. Thus, when compiling Nebo with standard optimizations, such as gcc's `-O3` optimizations, gcc inlines most `eval` function calls. (Compilers, such as gcc, use heuristics to determine when it is useful to inline functions such as Nebo's internal `eval` functions. In some rare cases it is not beneficial to inline functions, such as when inlining a long function would increase the distance of short jumps, such as conditional jumps, in the surrounding code beyond their range, forcing the surrounding code to use less efficient, yet longer range, jump instructions. Regardless of the efficiency of these heuristics, Nebo does not handle any of these optimizations and leaves them solely to the host C++ compiler.)

3.2.2. Multi-thread implementation

Nebo's strategy for multi-thread execution is to divide the fields underlying the current Nebo Expression into subfields. Each subfield is then assigned to a thread and executed sequentially on that thread. Nebo's semantics define that elements in Nebo assignment can be evaluated and assigned in any order. Thus, there is no need for inter-thread communication other than to signal that a subfield has finished execution.

When Nebo has decided to use the multi-thread backend, Nebo uses information from the field on the left-hand side of the assignment to determine a partitioning scheme. Users or frameworks can set that partition scheme. Once Nebo has determined its partitioning scheme, Nebo creates an instance of the Nebo Expression in `Resize` mode. The original Nebo Expression schedules each partition in a FIFO work queue with the Nebo Expression in `Resize` mode. A thread pool pulls jobs off of the work queue. A semaphore is used so that the original instance of the Nebo Expression in `Initial` mode is informed when the job is done.

3.2.3. Many-core (GPU) implementation

Because GPUs use a Single-Instruction Multiple-Data (SIMD) model of execution, Nebo's GPU backend is very different from Nebo's other backends. Nebo's GPU backend sets up a 'plane' of threads, such that each thread has a unique pair of *X*-axis and *Y*-axis indices. Then all the threads together iterate through all the *Z* indices. At each Z index, each thread calculates the result for its unique combination of *X*, *Y*, and *Z* indices. For example, consider a field whose dimensions are 3 by 4 by 5 (*X*, *Y*, *Z*, respectively). In this case, Nebo's GPU backend would use 12 threads (3 times 4), and each thread would calculate 5 different elements.

The code for each thread is somewhat similar to the code for sequential execution:

```
int x = xLow + threadIdx.x +
  blockIdx.x * blockDim.x;
int y = yLow + threadIdx.y +
  blockIdx.y * blockDim.y;

start(x, y, xHigh, yHigh);

for(int z = zLow; z < zHigh; z++)
 if(valid())
  ref(x, y, z) = rhs.eval(x, y, z);
```

Despite the differences between execution model, the code to execute a Nebo assignment on a GPU looks very similar: The first obvious difference between the sequential CPU code and the GPU code is that the `x` and `y` indices are fixed for each thread. Because the CUDA programming model constructs a Nebo Expression for each thread exactly the same for all threads, except for a few indexing variables (`blockIdx`, `blockDim`, and `threadIdx`), each thread must determine what its assigned *X*-axis and *Y*-axis indices are. The next obvious difference are the initialization method `start` and the guard method, `valid()`. For the sake of execution speed and regularity, sometimes threads are assigned *X*-axis and *Y*-axis indices that are outside the bounds of the fields. Thus, the `start` method determines for each thread if the *X*-axis and *Y*-axis indices of the current thread point to a valid element of the fields. The call to the `valid` method returns true, if and only if the `start` method determined that the indices are valid.

Because Nebo uses asynchronous kernel invocations, some synchronization must be handled by the end user. Nebo uses CUDA streams to synchronize kernel calls. Each field contains a CUDA stream, which are set outside of Nebo and which Nebo passes to the kernel calls. When the end user is using Nebo inside of Wasatch, Wasatch handles initializing CUDA streams and assigning them to the proper fields. Also, Wasatch handles synchronizing the CUDA streams where necessary.

3.2.4. Reduction implementation

The `Reduction` mode implements Nebo's reduction operations. Currently, Nebo reductions are implemented for single-core and many-core (GPU) execution. For the single-core implementation, the `Reduction` mode uses an interface for Nebo Expressions which is almost identical to the interface for `SeqWalk` mode. The major difference between a reduction and a single-thread assignment is that there is no left-hand-side/assignee in a reduction.

The reduction backend loop is very similar to the single-core backend for assignment:

```
for(int z = zLow; z < zHigh; z++)
 for(int y = yLow; y < yHigh; y++)
 for(int x = xLow; x < xHigh; x++)
 res = proc(res, expr.eval(x,y,z);
```

The GPU backend for reductions is likewise similar to the GPU backend for assignment. The major difference is that each thread computes a partial result, and the results are combined using standard NVidia parallel reduction techniques.

4. Results

This section presents three different performance results: The first in Section 4.1 for a simple scalar right-hand side term; the second in Section 4.2 for tests with different levels of computational intensity run with ExprLib, the task parallelism library used in Wasatch Notz et al. (2012); and the third in Section 4.3 comparing Wasatch with Nebo to other components of Uintah. These first two sections are tested using all of Nebo's backends. The final section is only evaluated using Nebo's CPU single-thread backend, to simplify the comparison to the other components of Uintah.

4.1. Scalar right-hand side term

The scalar right-hand side term, a single Wasatch task, is a great example of how Nebo has improved code in Wasatch:

$$\frac{\partial \phi_i}{\partial t} = -\frac{\partial}{\partial x}\left(C_x + D_x\right) - \frac{\partial}{\partial y}\left(C_y + D_y\right) - \frac{\partial}{\partial z}\left(C_z + D_z\right)$$

where C and D represent the convective and diffusive fluxes of ϕ in each direction. The original version of this calculation used 13 loops, because it predated the development of Nebo and there was no way to combine multiple operations into a single loop:

```
rhs = 0.0;

divOpX->apply_to_field(xConvFlux, tmp);
rhs -= tmp;
divOpX->apply_to_field(xDiffFlux, tmp);
rhs -= tmp;

divOpY->apply_to_field(yConvFlux, tmp);
rhs -= tmp;
divOpY->apply_to_field(yDiffFlux, tmp);
rhs -= tmp;

divOpZ->apply_to_field(zConvFlux, tmp);
rhs -= tmp;
divOpZ->apply_to_field(zDiffFlux, tmp);
rhs -= tmp;
```

The current version makes full use of Nebo, and needs only a single assignment (one loop):

```
rhs <<= - divOpX(xConvFlux + xDiffFlux)
        - divOpY(yConvFlux + yDiffFlux)
        - divOpZ(zConvFlux + zDiffFlux);
```

The Nebo version of this code replaces 13 statements, each of which did a single operation, with one statement that performs the same calculation. The Nebo version is easier to read, understand, and maintain than the original version is.

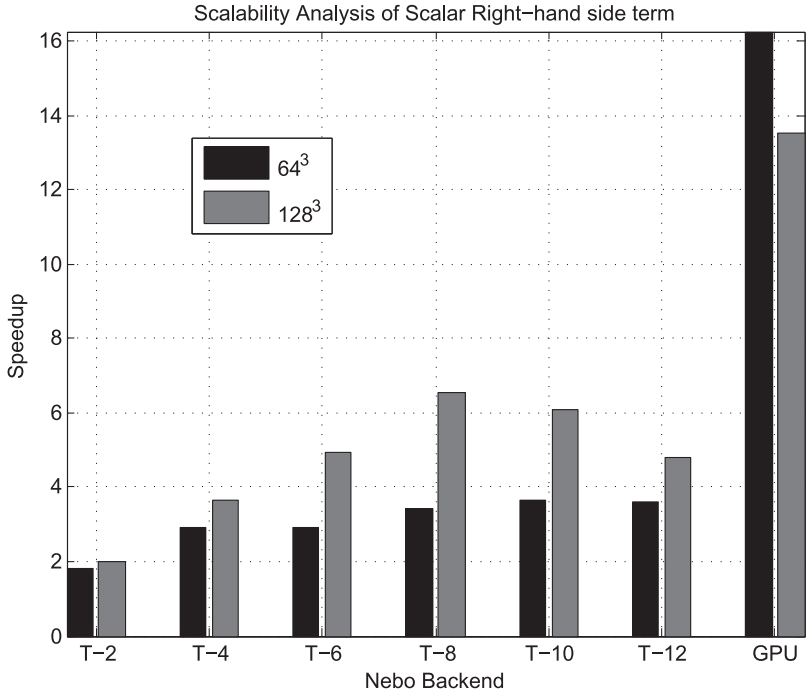


Fig. 1. Speedup of Nebo's parallel backends over Nebo's single-thread backend for the scalar right-hand side term with problem size of 64^3 and 128^3. The specific parallel backends tested here are 2, 4, 6, 8, 10, and 12 threads with the multi-thread backend as well as the GPU backend. (T-X refers to the multi-thread backend with X threads.) These tests were run on a 12-core Intel Xeon E5-2620 (2×6 cores at 2.00 GHz and 15MB cache) with 16 GB RAM and a NVidia GeForce GTX 680. These tests compared the entire execution run. Thus, while Nebo does not automate memory transfer to/from GPUs, the speedups for the GPU execution includes the time needed for memory transfers to/from the GPU.

More important than the simplification of the code, the current version of the scalar right-hand side code performs nearly twice as fast as the original version: For a problem size of 64^3 elements, the current version is 1.88 $\times$ faster than the original version, and for a problem size of 128^3 elements, the current version is 1.91 $\times$ faster than the original version.

The speedup of the Nebo version over the original comes from fewer instructions (the overhead from one loop instead of 13 loops) and from not storing intermediate results to memory (better cache usage). One could write by hand a single loop that performs the same calculation; however, one would then need to calculate the stencil operations by hand. Furthermore, the hand-written single loop would not be portable to the GPU, and would require further modification to run on multiple threads in parallel.

Fig. 1 shows that Nebo's multi-thread backend for the scalar right-hand side term scales to 4 threads for both problem sizes. The GPU backend, however, is up to 16 $\times$ faster than Nebo's single-thread CPU backend. The GPU backend has less loop-related overhead than the CPU backends. (All CPU threads hava a triply-nested loop structure for x-, y-, z-dimensions, whereas all GPU threads have a flat loop structure for the z-dimension.) However, at larger problem sizes, such as the 128^3 case, this loop-related overhead matters less on the CPU. Thus, the 128^3 case scales better than the 64^3 case on the multi-threaded CPU backend. Likewise, the CPU loop-related overhead is less with on the 128^3 case, and so on that case the GPU backend does not perform as well relative to the single-thread CPU backend, compared to its performance on the 64^3 case.

Finally, the calculation for this term is computationally light: Each stencil contains two multiplications and an addition, for a total of six multiplications, six additions, two subtractions, and a negation. For Nebo's multi-thread backend to scale further, the calculations need to be more computationally intensive. Likewise, Nebo's many-core backend can perform better relative to single-thread performance with more computationally intensive calculations. The following performance tests confirm the need for more computationally intensive calculations.

4.2. Task graph results

We set up several tests of differing computational intensity in ExprLib that evaluate diffusion and source term expressions for obtaining solution variables. The source terms involved in these tests are representative of the same type of calculations used in a detailed chemical kinetics simulation. These tests each involve evaluating 30 partial differential equations (PDEs) arranged in the form of a task graph for 100 iterations. Each of the 30 PDEs has a task for the diffusive flux, a task for the source term (when present), and a task to combine the results of the diffusive flux and the source term. Thus, there are 90 tasks in the task graph for the tests with source terms and 60 tasks for the test without a source term.

The mathematical expression calculated for these tests is:

$$\frac{\partial}{\partial x}\left(\Gamma_i \frac{\partial \phi_i}{\partial x}\right) + \frac{\partial}{\partial y}\left(\Gamma_i \frac{\partial \phi_i}{\partial y}\right) + \frac{\partial}{\partial z}\left(\Gamma_i \frac{\partial \phi_i}{\partial z}\right) + s_i$$

where s_i is different for each of the three tests. The first test calculates only diffusive flux, and so the source term is removed. The second test calculates diffusive flux and an independent source term, $s_i = \sum_{j=1}^{n} \exp(\phi_j)$. The third test calculated diffusive flux and a coupled source term, where s_i depends on all of the ϕ_j as $s_i = \sum_{j=1}^{n} \exp(\phi_j)$. These tests are in order of increasing computational intensity.

Fig. 2 shows speedup of Nebo's multi-thread performance over Nebo's single-thread performance for all three tests with problem sizes of 64^3 and 128^3. Likewise, Fig. 3 shows speedup of Nebo's many-core (GPU) performance over single-thread performance for

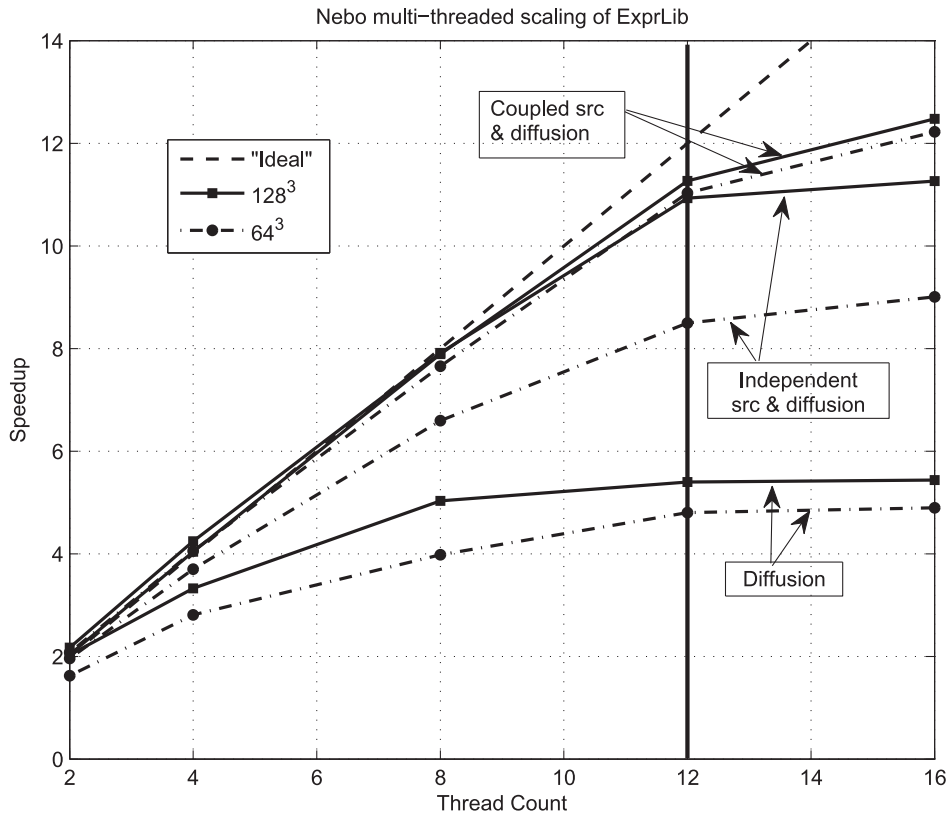


Fig. 2. Speedup of Nebo's multi-thread backend with 2, 4, 8, 12, and 16 threads over Nebo's single-thread backend for the ExprLib tests with problem size of 64^3 and 128^3. These tests were run on a 12-core Intel Xeon E5-2620 (2x6 cores at 2.00 GHz) with 16 GB RAM. The solid vertical line indicates the number of physical cores on the machine. The tests were conducted without using the Uintah's framework and hence there is no overhead related to communication and data-storage offered by Uintah.

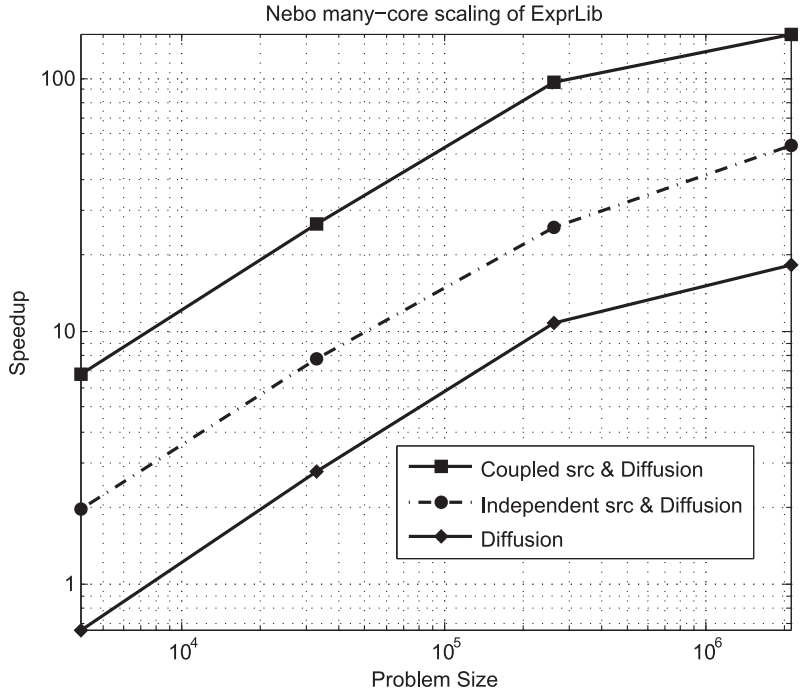


Fig. 3. Speedup of Nebo's many-core (GPU) backend over Nebo's single-thread backend for the ExprLib tests with problem size of 16^3, 32^3, 64^3, and 128^3. The coupled source and diffusion test is 140 $\times$ faster on Nebo's many-core backend than Nebo's single-thread backend with a problem size of 128^3. These task graph tests were run on a 12-core Intel Xeon E5-2620 (2x6 cores at 2.00 GHz) with 16 GB RAM and a NVidia Tesla K20 architecture. The tests were conducted without using the Uintah's framework and hence there is no overhead related to communication and data-storage offered by Uintah. These tests compared the entire execution run. Thus, while Nebo does not automate memory transfer to/from GPUs, the speedups for the GPU execution includes the time needed for memory transfers to/from the GPU.

the same tests and problem sizes. Fig. 2 shows that the diffusion with a coupled source term test scales linearly up to the number of cores on the system in use (12) with Nebo's multi-core backend. The diffusion with independent source term test does not scale as well, especially for the smaller problem size; moreover, the diffusion only test does not scale well, particularly beyond 4 cores. Fig. 3 shows that more than half of the ExprLib tests with Nebo's many-core backend are more than 10 × faster than the single-thread backend, and the fastest of which is just over 140 × faster. Only the diffusion only test on the smallest problem size (16^3) is slower than the single-thread backend. This problem size is far smaller than what is typically run on a single node within the broadly distributed MPI simulation. As with the multi-thread backend tests, the diffusion with a coupled source term test scales the best, and the diffusion only test scales the worst.

The general trend of these tests is that more computationally intensive calculations (coupled source with diffusion) perform better than less computationally intensive (diffusion only) calculations. The reason for this trend is that computational intensity hides memory latency. Finally, it is interesting to note that Nebo's multi-thread backend with 16 threads improves over 12 threads for most tests on a system with 12 cores. This limited improvement comes from hyper-threading.

4.3. Code to code comparisons

The Taylor-Green vortex Taylor and Green (1937); Brachet et al. (1983) is a classic two-dimensional fluid dynamics problem, whose analytic solution makes it a common verification problem for numerical PDE solvers. In this section, we are not using the Taylor-Green vortex for verification but rather as a basis for comparison of performance of several CFD solvers. In particular, Wasatch, Arches Schmidt et al. (2013b), ICE Guilkey et al. (2007), all components of Uintah written by application domain experts, solve the Taylor-Green vortex problem using very similar numerical schemes. Unlike Wasatch, Arches and ICE do not use a domain-specific language for their numeric calculations but use hand-written loops instead.

Fig. 4 presents the single-core speedup for Wasatch relative to Arches and ICE. For small domain sizes (8^3), Wasatch, Arches, and ICE perform roughly the same with Wasatch doing slightly better. As the domain size grows, Wasatch performs increasingly well compared to Arches and ICE. At the largest size (128^3), Wasatch using Nebo runs nearly 6x faster than Arches and nearly 10x faster than ICE. In practical applications, for MPI-scalability considerations, patch sizes are typically in excess of 32^3, which corresponds to $> 4\times$ and $> 6\times$ speedup for Wasatch.

As components of Uintah, Arches and ICE use the same interface and framework for communication and data storage as Wasatch does. This comparison shows that Wasatch's approach and use of Nebo is very competitive inside of Uintah. In particular, it demonstrates that Nebo does involve any intrinsic overhead that prevents Wasatch from performing better than both Arches and ICE. It also bears mentioning that the timings reported in this section exclude the Poisson solver, used to calculate pressure, since it is used in the same manner across all three components. Nebo provides an efficient (and correct) language/library for calculating the numeric solutions to PDEs that separates the concerns of correctness and speed (what versus how). By using Nebo, Wasatch can write code that not only out-performs sibling codes within the Uintah framework, but that is also architecture-portable; deployment of Nebo on GPU or multi-core CPU is done without any intervention on the part of the application developer.

5. Related work

There are many other domain-specific languages that have functionality and domains similar to Nebo, but none contain all of Nebo's features. POOMA Reynders (1997) is probably the most comparable DSL to Nebo. POOMA is in the same domain, uses similar abstractions, is embedded in C++, and supports thread and message-passing parallelism. POOMA has not scaled to the extent that Nebo, Wasatch, and Uintah have (weak scaling upto 262K cores, see Earl (2014)), and POOMA does not support GPU execution.

The Pochoir stencil compiler Tang et al. (2011) supports stencil calculations very similar to the stencils Nebo provides. Pochoir is semi-embedded in C++, since it uses an external compiler for optimization. While Pochoir's optimizations are more advanced that Nebo's, Pochoir does not support GPU execution. Furthermore, Pochoir does more optimizations than Nebo; however, Pochoir must analyze the entire time-step function, which currently limits it to simple time-step functions. By comparison, Wasatch regularly runs time-step functions that use dozens and sometimes hundreds of variables.

Liszt DeVito et al. (2011) represents PDEs by abstracting based on geometry and spatial reasoning rather than mathematical equations as Nebo does. Liszt supports both CPU- and GPU-based parallelism, but does not support incremental adoption. Thus, entire applications must be written in Liszt to use Liszt.

OptiMesh Sujeeth et al. (2013), developed with the Delite compiler Brown et al. (2011), offers CPU- and GPU-based parallel backends within the same runtime environment, like Nebo. OptiMesh uses the same abstractions and much of the same syntax as Liszt for solving PDEs. In general, OptiMesh performs better than Liszt because Delite supports more aggressive optimizations.

POOMA, Pochoir, and OptiMesh support forms of incremental adoption, while Liszt does not. For Pochoir and OptiMesh, partial adoption require adding new compilers to a projects build system. In comparison, Nebo works without adding a new compiler in existing C++ projects.

Algorithmic skeletons, such as SKePU Enmyren and Kessler (2010) and Marrow Marques et al. (2013), provide similar capabilities to Nebo, without the domain-specific abstractions and functionality for numerically solving partial differential equations. Algorithmic skeletons provide basic parallelism abstractions that avoid implementation-specific parallelism details; however, to use algorithmic skeletons, users must write their programs in terms of these basic parallelism abstractions. To use Nebo, users write their programs in mathematical equations and terms, closely matching the partial differential equations that they are ultimately trying to solve numerically.

6. Future work

Nebo and ExprLib are works in progress. We are currently integrating Wasatch's (Nebo) GPU backend with Uintah's GPU support. Some calculations in Wasatch are done through third-party libraries which do not support parallelization. We are working on integrating these libraries into Nebo's multi-thread backend to parallelize these heavy operations. We are also working on adding boundary conditions to Nebo, so that non-periodic boundary conditions can be computed on the GPU, rather than just on the CPU, as is currently done. Since Nebo does not have any method to fuse loops, we are starting a new project that would automate loop fusion between Nebo assignments. We are considering adding new backends to Nebo to support architectures such as Intel's Xeon Phi co-processors.

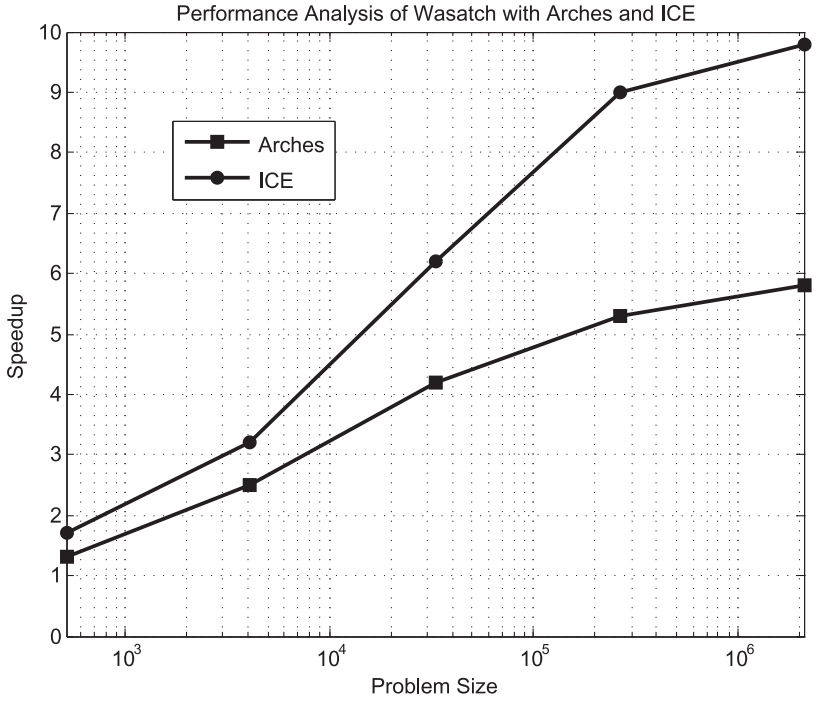


Fig. 4. Taylor-Green vortex test results showing speedup of Wasatch over Arches and ICE on problem sizes 8^3, 16^3, 32^3, 64^3, and 128^3. Each test ran on a single processor using a single thread.

7. Conclusion

Using Nebo, domain experts are able to create code that is efficient, scalable, and portable across multiple architectures. Despite not being feature complete, Nebo and ExprLib have good results: Nebo, on its own, can be better than C++ code hand-written by domain experts, and automates parallelism with threads and GPUs. With a graph of 90 tasks and a computationally intensive simulation, Nebo and ExprLib scale linearly up to the number of cores on the system with Nebo's multi-thread backend, and can perform 140 × faster with Nebo's GPU backend than Nebo's CPU backend. Moreover, Nebo, ExprLib, and Wasatch are significantly faster than Arches and ICE for the Taylor-Green Vortex problem. Finally, Wasatch using ExprLib and Nebo has weakly scaled to 262K cores on Titan Schmidt et al. (2013a).

Acknowledgments

The authors gratefully acknowledge support from NSF PetaApps award 0904631 and DOE Cooperative Agreement DE-NA0000740. This material is based in part upon work supported by the National Science Foundation under Grant Number 1248464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Document number: LLNL-JRNL-665611.

References

Berzins, M., Luitjens, J., Meng, Q., Harman, T., Wight, C.A., Peterson, J.R., 2010. Uintah: a scalable framework for hazard analysis. In: Proceedings of the 2010 TeraGrid Conference. ACM, New York, NY, USA, pp. 3:1–3:8. doi:10.1145/1838574.1838577.

Berzins, M., Meng, Q., Schmidt, J., Sutherland, J.C., 2012. Dag-based software frameworks for PDEs. In: Euro-Par 2011: Parallel Processing Workshops. Springer, pp. 324–333.

Brachet, M.E., Meiron, D.I., Orszag, S.A., Nickel, B., Morf, R.H., Frisch, U., 1983. Small-scale structure of the Taylor-Green vortex. J. Fluid Mech. 130, 411–452.

Brown, K.J., Sujeeth, A.K., Lee, H.J., Rompf, T., Chafi, H., Odersky, M., Olukotun, K., 2011. A heterogeneous parallel framework for domain-specific languages. In: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT),. IEEE, pp. 89–100.

Dean, J., Ghemawat, S., 2008. MapReduce: Simplified data processing on large clusters. Commun. ACM 51 (1), 107–113. doi:10.1145/1327452.1327492.

DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P., 2011. Liszt: a domain specific language for building portable mesh-based PDE solvers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, New York, NY, USA, pp. 9:1–9:12. doi:10.1145/2063384.2063396.

Earl, C., 2014. Introspective pushdown analysis and Nebo Ph.D. thesis. The University of Utah.

Enmyren, J., Kessler, C.W., 2010. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications. ACM, New York, NY, USA, pp. 5–14. doi:10.1145/1863482.1863487.

Guilkey, J.E., Harman, T.B., Banerjee, B., 2007. An Eulerian–Lagrangian approach for simulating explosions of energetic devices. Comput. Struct. 85 (11-14), 660–674. doi:10.1016/j.compstruc.2007.01.031.

Marques, R., Paulino, H., Alexandre, F., Medeiros, P.D., 2013. Algorithmic skeleton framework for the orchestration of GPU computations. In: Wolf, F., Mohr, B., an Mey, D. (Eds.), Euro-Par 2013 Parallel Processing. In: Lecture Notes in Computer Science, 8097. Springer Berlin Heidelberg, pp. 874–885. doi:10.1007/978-3-642-40047-6_86.

Notz, P.K., Pawlowski, R.P., Sutherland, J.C., 2012. Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software. ACM Trans. Math. Softw. 39 (1), 1.

Parker, S.G., 2002. A component-based architecture for parallel multi-physics PDE simulation. In: Computational Science—ICCS 2002. Springer, pp. 719–734.

Punati, N., Sutherland, J.C., Kerstein, A.R., Hawkes, E.R., Chen, J.H., 2011. An evaluation of the one-dimensional turbulence model: comparison with direct numerical simulations of CO/H2 jets with extinction and reignition. Proc. Combust. Inst. 33 (1), 1515–1522. doi:10.1016/j.proci.2010.06.127.

Reynders, J., 1997. The POOMA framework: a templated class library for parallel scientific computing.. In: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing.

Schmidt, J., Berzins, M., Thornock, J., Saad, T., Sutherland, J., 2013. Large scale parallel solution of incompressible flow problems using Uintah and hypre. In: Proceedings of the International Symposium on Cluster, Cloud and Grid Computing. Delft, Netherlands.

Schmidt, J., Berzins, M., Thornock, J., Saad, T., Sutherland, J., 2013. Large scale parallel solution of incompressible flow problems using uintah and hypre. In: Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, pp. 458–465.

Stoustrup, B., 1997. The c++ programming language.
Sujeeth, A.K., Rompf, T., Brown, K.J., Lee, H., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M., et al., 2013. Composition and reuse with compiled domain-specific languages. In: Proceedings of ECOOP.
Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.-K., Leiserson, C.E., 2011. The Pochoir stencil compiler. In: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures. ACM, pp. 117–128.
Taylor, G., Green, A., 1937. Mechanism of the production of small eddies from large ones. Proc. R. Soc. Lond. Ser. A, Math. Phys. Sci. 158 (895), 499–521.
Unruh, E., 1994. Prime number computation. Technical Report. ANSI X3J16-94-0075/ISO WG21-462.
Veldhuizen, T., 1995. Template metaprograms. C++ Report 7 (4), 36–43.

Christopher Earl is a postdoctoral researcher in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. He obtained his Ph.D. in Computer Science at the University of Utah.

Matthew Might is an Associate Professor in the School of Computing and Presidential Scholar at the University of Utah and a Visiting Associate Professor in Biomedical Informatics at the Harvard Medical School.

Abhishek Bagusetty is a Ph.D. student at Center for Simulation and Modeling and affiliated with Department of Chemical & Petroleum Engineering at the University of Pittsburgh. Prior to University of Pittsburgh, he obtained his Master's in Chemical Engineering at the University of Utah.

James Sutherland is an Associate Professor in the Chemical Engineering department at the University of Utah whose research focuses on developing tools, algorithms and models to enable multiscale simulation of turbulent reacting flows on modern computing architectures.