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

Christopher Earl

University of Utah
cwearl@cs.utah.edu

Matthew Might

University of Utah
might@cs.utah.edu

Abhishek Bagusetty

University of Pittsburgh
abb58@pitt.edu

James C. Sutherland

University of Utah
James.Sutherland@utah.edu

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 [1, 2, 3] framework and has demonstrated scalability to 262,000 cores [4]. Wasatch is a code for convection-diffusion-reaction problems, and focuses on turbulent reacting flow simulations using large eddy simulation. While this paper discusses Nebo's use in Wasatch, Nebo is a stand-alone library and is used in other projects (see, e.g., [5]). Nebo handles data parallelism but leaves memory management and data transfers between CPU and GPU either to a

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

framework or to the end user.

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 efficient code for a variety of backend implementations. Moreover, Nebo's semantics are intensionally restricted, which limits what can be computed within Nebo. For example, all Nebo code will terminate: Nebo calculations only read data from memory a fixed number of times (usually once). 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 [1, 2, 3] 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 cores available, and Nebo's many-core backend can perform 140x 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.

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.

Because Nebo is embedded within C++ [6], 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 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 2.2.

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.

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

2.1. 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 2.2 and 2.5, 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 2.3; or a stencil operator applied to a Nebo Expression, which is discussed in Section 2.4. 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.

2.2. 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.

2.3. 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 (• ? • : •). 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 (• == •, • ! = •, • < •, • > •, • <= •, and • >= •); or a logical connective of Nebo Boolean Expressions, using any of the C++ logical connective operators (• && •, • || •, and !•).

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 (• ? • : •), 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; Scalar s; Field a, b;
//...
b <<= b * cond(a > 0.0, a)
               (a < 0.0, -a)
               (d, b)
               (1.0);
```

2.4. 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 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]);
```

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`. 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 are invalidated by 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 available ghost information at runtime. Once ghost cells are exhausted, communication with neighboring processes is required 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));
```

2.5. 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 [7], 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:

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

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));
```

3. Implementation of Nebo

This section presents the implementation of Nebo in two parts: Parsing (Section 3.1) and the backends (Section 3.2). The implementation of Nebo can be downloaded using git from:

`git: //software.crsim.utah.edu/SpatialOps.git`

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 [8, 9].

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 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 Nebo assignment, is executed, an instance of the Nebo Expression AST is constructed in the `Initial` mode. Then, based on compile-time and runtime conditions, a backend is chosen, and

the Nebo Expression AST is constructed 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> > >
```

When a backend has been chosen, a new instance of the AST is created with a different mode. 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 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 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. 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.

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.

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 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 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 constructs the 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 result, and then all thread coordinate after the loop to find the final result.

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 [10]; 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:

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

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);
```

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 1.88x faster than the original version, and for a problem size of 128^3 elements, the current version is 1.91x 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 and would have comparable performance to the Nebo version; 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.

Figure 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 16x faster than Nebo's single-thread CPU backend.

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 be 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,

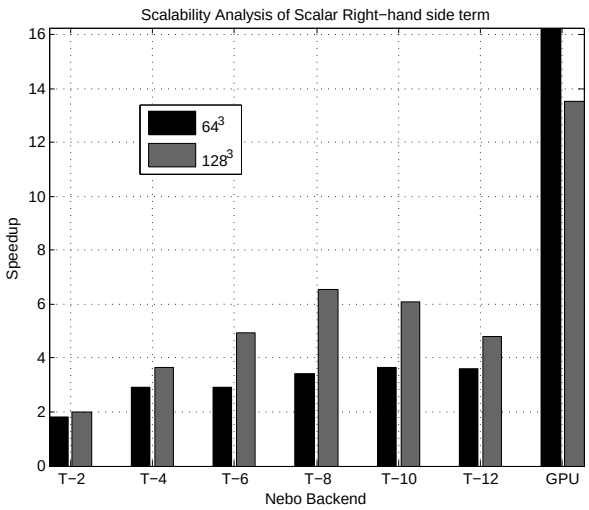

Figure 1: Speedup of Nebo's parallel backends over Nebo's single-thread backend for the scalar right-hand side term with problem size of 64^3 and 128^3. The specific parallel backends tested here are 2, 4, 6, 8, 10, and 12 threads with the multi-thread backend as well as the GPU backend. (T-X refers to the multi-thread backend with X threads.) These tests were run on a 12-core Intel Xeon E5-2620 (2x6 cores at 2.00 GHz) with 16 GB RAM and a NVidia GeForce GTX 680. As Nebo does not handle memory transfer to/from GPUs, the GPU performance results represent the runtime of the kernel call without any data transfers.

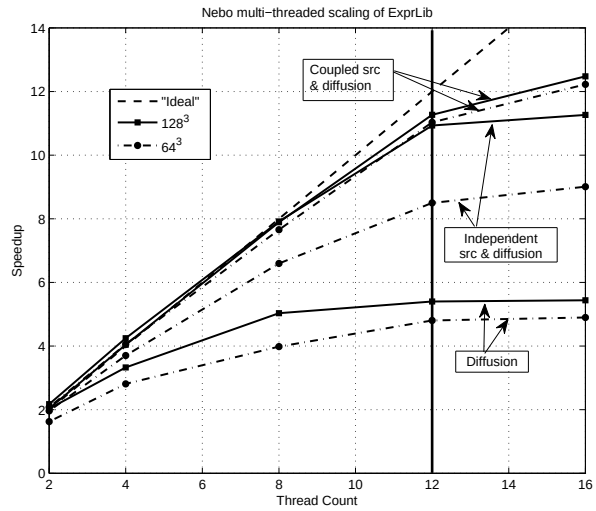

Figure 2: Speedup of Nebo's multi-thread backend with 2, 4, 8, 12, and 16 threads over Nebo's single-thread backend for the ExprLib tests with problem size of 64^3 and 128^3. These tests were run on a 12-core Intel Xeon E5-2620 (2x6 cores at 2.00 GHz) with 16 GB RAM. The solid vertical line indicates 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.

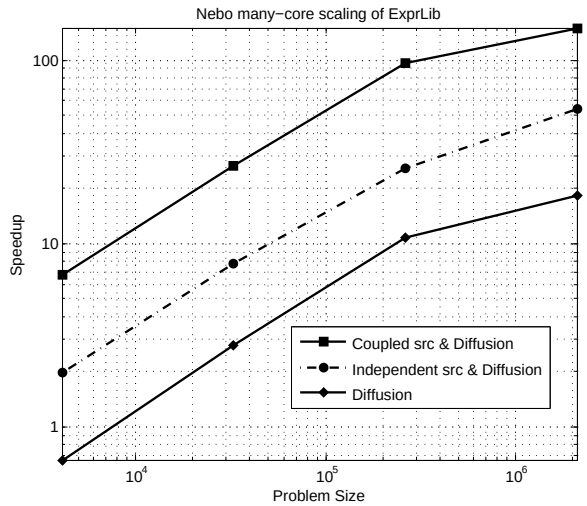

Figure 3: Speedup of Nebo's many-core (GPU) backend over Nebo's single-thread backend for the ExprLib tests with problem size of 16^3, 32^3, 64^3, and 128^3. The coupled source and diffusion test is 140x faster on Nebo's many-core backend than Nebo's single-thread backend with a problem size of 128^3. 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. The tests run on GPUs did not require any intermediate data-transfers between CPU and GPU, and thus data transfer to/from the GPU is not measured in these tests.

$s_i = \sum_{i=1}^{n} \exp(\phi_i)$. 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.

Figure 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, Figure 3 shows speedup of Nebo's many-core (GPU) performance over single-thread performance for the same tests and problem sizes. Figure 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. Figure 3 shows that more than half of the ExprLib tests with Nebo's many-core backend are more than 10x faster than the single-thread backend, and the fastest of which is just over 140x 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 [11, 12] 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 [13], ICE [14], 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.

Figure 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 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 >4x and >6x 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

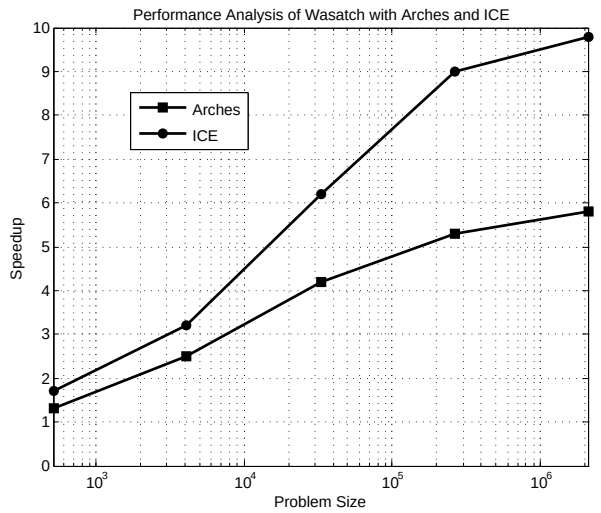

Figure 4: Taylor-Green vortex test results showing speedup of Wasatch over Arches and ICE on problem sizes 8^3, 16^3, 32^3, 64^3, and 128^3. Each test ran on a single processor using a single thread.

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 [15] 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, and POOMA does not support GPU execution.

The Pochoir stencil compiler [16] 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 [17] 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 [18], developed with the Delite compiler [19], 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.

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 parallelizes 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.

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, is as good and often 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 140x 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 [4].

Acknowledgment

The authors gratefully acknowledge support from NSF PetaApps award 0904631 and DOE Cooperative Agreement DE-NA0000740.

References

[1] M. Berzins, Q. Meng, J. Schmidt, and J. C. Sutherland, "Dag-based software frameworks for PDEs," in *Euro-Par 2011: Parallel Processing Workshops*. Springer, 2012, pp. 324–333.

[2] S. G. Parker, "A component-based architecture for parallel multi-physics PDE simulation," in *Computational Science—ICCS 2002*. Springer, 2002, pp. 719–734.

[3] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. A. Wight, and J. R. Peterson, "Uintah: a scalable framework for hazard analysis," in *Proceedings of the 2010 TeraGrid Conference*, ser. TG '10. New York, NY, USA: ACM, 2010, pp. 3:1–3:8.

[4] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland, "Large Scale Parallel Solution of Incompressible Flow Problems using Uintah and hypre," in *International Symposium on Cluster, Cloud and Grid Computing*, Delft, Netherlands, May 2013. [Online]. Available: http://www.pds.ewi.tudelft.nl/ccgrid2013

[5] N. Punati, J. C. Sutherland, A. R. Kerstein, E. R. Hawkes, and J. H. Chen, "An Evaluation of the One-Dimensional Turbulence Model: Comparison with Direct Numerical Simulations of CO/H2 Jets with Extinction and Reignition," *Proc. Combust. Inst.*, vol. 33, no. 1, pp. 1515–1522, 2011.

[6] B. Stoustrup, "The c++ programming language," 1997.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[8] E. Unruh, "Prime number computation," ANSI X3J16-94-0075/ISO WG21-462, Tech. Rep., 1994.

[9] T. Veldhuizen, "Template metaprograms," *C++ Report*, vol. 7, no. 4, pp. 36–43, 1995.

[10] P. K. Notz, R. P. Pawlowski, and J. C. Sutherland, "Graph-based software design for managing complexity and enabling concurrency in multiphysics PDE software," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 1, p. 1, 2012.

[11] G. Taylor and A. Green, "Mechanism of the production of small eddies from large ones," *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, vol. 158, no. 895, pp. 499–521, 1937.

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

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

[14] J. E. Guilkey, T. B. Harman, and B. Banerjee, "An Eulerian-Lagrangian approach for simulating explosions of energetic devices," *Comput. Struct.*, vol. 85, no. 11-14, pp. 660–674, Jun. 2007.

[15] J. Reynders, "The POOMA framework: A templated class library for parallel scientific computing." in *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[16] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir stencil compiler," in *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*. ACM, 2011, pp. 117–128.

[17] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "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*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 9:1–9:12.

[18] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky *et al.*, "Composition and reuse with compiled domain-specific languages," in *Proceedings of ECOOP*, 2013.

[19] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 89–100.