A Composable Domain Specific Language Extension for Spatio-Temporal Data Mining

Matthew Le
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota
lematt@cs.umn.edu

Kevin Williams
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota
kwill@cs.umn.edu

Eric Van Wyk
Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota
evw@cs.umn.edu

***Abstract*—In this paper we describe our work in progress in building modular and composable domain-specific language extensions for C, targeting matrix processing and spatio-temporal data mining. The paper also describes two general purpose language extensions to add tuples and reference-counting pointers. Both types of extensions add new syntax and semantic analysis to the host C language specification and are translated down to plain (parallel) C code. What distinguishes this approach to language extensions is language processing tools automatically and reliably compose language extensions to create a translator for the custom language defined by the set of programmer chosen extensions so that the programmer need not understand the underlying compiler techniques used to create the composed language.**

I. Introduction

As multicore architectures become more and more prevalent, programming language designers continue to search for useful abstractions to relieve the programmer from dealing directly with the many issues that arise when writing multi-threaded code. The problem is further complicated by the fact that both the architectures used and the domains using them (at least in the case of spatio-temporal data mining) are both rapidly changing. Ideally, we would like programmers to be able to customize their languages with new features in way that if they turn out to be useful, one could continue to use them, and if not, they can be removed from the language. Extensible languages address these sorts of problems by allowing new features to be added to a language in a way that the programmer gets to decide the composition of the language.

A canonical example in extensible languages is the addition of language constructs for specifying SQL queries to Java in a way that the query can be parsed and type checked so that informative errors can be reported if necessary. Another example which is more relevant to the HPC community, and what this paper focuses on, is added support for matrices and a rich set of accompanying operations to extend an imperative compiled language like C. These extensions can be statically type checked and produce errors such as a dimension mismatch in matrix arithmetic, furthermore, they are able to generate highly optimized code targeting multicore architectures.

In this paper, we describe our work in progress to develop composable language extensions that we have implemented for a subset of C. Some of these extensions are general-purpose in nature; they add a notion of tuple and reference counting pointers that automatically free their data. Others are domain-specific and are directed at matrix processing, in general, and spatio-temporal data mining in particular. Both types of extensions provide new syntax and semantic analyses to the language before translating the added features down to plain C code (for eventual compilation to executable form).

Spatio-temporal data mining is an inherently interdisciplinary field, which often spans AI, machine learning, statistics, and many other domains. Because of its interdisciplinary nature, we have found that many researchers in this field end up implementing their applications in more than one language. This complicates things tremendously, as researchers often need to work with many libraries and other features in order to make this work. A better situation would be to unify these features into one language, and extend the language by adding new features when a given problem cannot be easily encoded in the language at hand.

The novelty of this work comes from the way that this language and its extensions are implemented and composed. In our approach language extensions are built by people that have some degree of knowledge about programming language processing and translation, but the programmers that use these extensions need no such knowledge. These programmers can import language extensions into their host language compiler in a manner not unlike how programmers use traditional libraries: they get to pick and choose the ones that they want for their task at hand and have some assurance that their selected libraries (or in our case, language extensions) will work together. Such a goal raises a number of challenges in implementing extensible languages. We use attribute grammars [1], [2] and context-aware scanners [3] with LR parsers [4] to specify host languages and language extensions since these formalisms naturally compose. Furthermore we have developed modular analyses of these formalisms that provide strong guarantees that programmer-selected language extensions will compose with the host language to form a working compiler for their customized language [5], [6].

```
1 //Compute the mean of a time series
2 float temporalMean(Matrix float<1> mat) {
3   int n = dimSize(mat, 0); //get length
4   return (with([0] <= [i] < [n])
5             fold(+,0.0,mat)) / n ;
6 }
7 int main (int argc, char **argv) {
8   Matrix float<3>mat = readMatrix("ssh.data");
9   int m = dimSize(mat, 0);
10  int n = dimSize(mat, 1);
11  //Compute temporal means using with loop
12  means = with([0,0] <= [i,j] < [m,n])
13    genarray([m,n],temporalMean(mat[i,j,::]);
14  writeMatrix("means.data", means);
15  return 0;
16 }
```

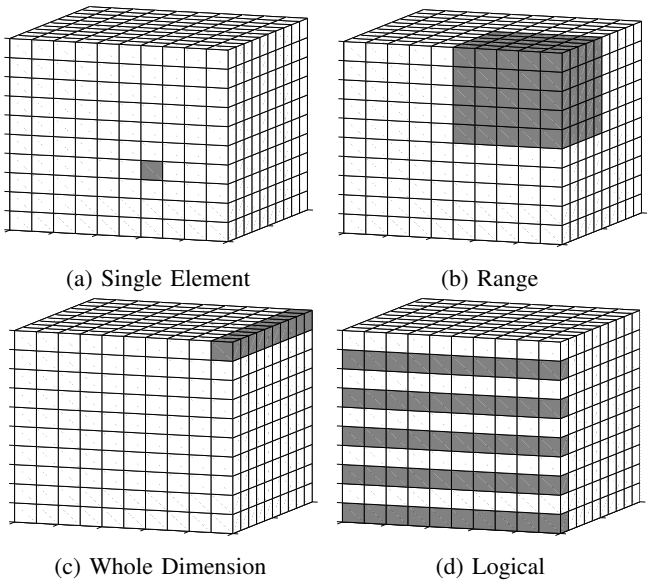
Fig. 1: Temporal Mean Algorithm Written in Extended C

In section II we describe the domain-specific and general-purpose extensions added to the host language C. In section III we show how these are used in an application that detects ocean eddies from sea-surface height data. In section IV we show how efficient parallel code can be generated from these language extensions. Section V briefly describes the tools that are used to build the host language and language extensions. Section VI discusses related work and concludes.

II. Language extensions for C

Our interest in spatio-temporal data mining comes from our collaborative work with experts mining climate data, typically gathered by satellites. Many of their applications are written in MATLAB since climate data can be naturally represented in its multidimensional matrices. Additionally, many of the language features, such as multiple-element indexing and flexible support for matrices of arbitrary rank make many climate applications concise and easier to implement and comprehend than if done in a general purpose language like C of FORTRAN.

We provide an implementation of a subset of C as the "host" language, with extensions that support many of the useful features applicable to spatio-temporal data mining found in MATLAB, Single Assignment C [7], and other general purpose languages. One version of this idea can be seen in Figure 1, where we have written a program to compute a simple temporal mean. In this example, we are reading in and using sea surface height data, which is a three dimensional data set in the shape of $721 \times 1440 \times 954$ where the dimensions correspond to latitude, longitude and time respectively. The objective behind the temporal mean is as follows: for every measured point on the ocean's surface we want to take the mean of all points in time. One could also think of this data set as a matrix, where each row/column element contains a time series with 954 measurements for which we want to compute the mean. Here we can see that we have a program that looks and feels a lot like C, but still has a fair amount of unfamiliar syntax. First, there is a new data type for matrices, which is integrated seamlessly into the type system of the host language. Second, we can see in line 13 that we have a way of indexing multiple elements of a matrix similar to what one would do in MATLAB. Lastly, we have made quite a bit of use of the "with-loop," which has been taken from the Single Assignment C (SAC) language [7], giving us a way of "mapping" (lines 9-11, 13-14) and "folding" (lines 4-5) a function over a matrix". These extended constructs will be explained in further detail below.



(a) Single Element (b) Range

(c) Whole Dimension (d) Logical

Fig. 2: Visualization of variations of matrix indexing.

A. Matrix Extension

First, we have a domain specific extension which adds many features that can be found in MATLAB and SAC.

1) Matrix Type: The first thing we have added is a new data type to our host language's type system, which is for matrices. A variable can be declared as a matrix by using the following type expression:

$$TypeExpr \Rightarrow Matrix\ (int \mid bool \mid float)\ \text{`<'}\ Integer\ \text{`>'}$$

After the keyword "Matrix" is a type that specifies the type of the matrix elements, followed by an integer literal specifying the number of dimensions. As of now, matrices can only contain integers, booleans, or floating point numbers.

2) Matrix Indexing: We have implemented our matrix indexing in the style of MATLAB. These are illustrated in Fig. 2 for a $10 \times 10 \times 10$ matrix with the origin in the forward upper left and allow the programmer to specify the following:

(a) Standard single element matrix indexing – data[6, 6, 0]
(b) Range indexing – data[0::4, **end**-4::**end**, 0::4]
(c) Whole dimension indexing – data[0, end, ::]
(d) Logical indexing – data[v % 2 == 1, ::, 0], where "v" is the one dimensional array: $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$.

These various methods of matrix indexing can be used in any combination for a matrix of arbitrary rank. Additionally,

$$
\begin{array}{rcl}
\textit{WithLoop} & \Rightarrow & \textit{with}\ \text{`('}\ \textit{Generator}\ \text{`)'}\ \textit{Operation} \\
\textit{Generator} & \Rightarrow & \text{`['}\ \textit{ExprList}\text{`]'}\ \textit{ROp}\ \text{`['}\textit{Id}\ (\text{`,'}\textit{Id})^*\ \text{`]'} \\
 & & \textit{ROp}\ \text{`['}\ \textit{ExprList}\ \text{`]'} \\
\textit{ROp} & \Rightarrow & \text{`}<\text{'} \mid \text{`}<=\text{'} \\
\textit{Operation} & \Rightarrow & \textit{genarray}\ \text{`('}\ \text{`['}\ \textit{ExprList}\ \text{`]'}\ \text{`,'}\ \textit{Expr}\ \text{`)'} \\
 & \mid & \textit{fold}\ \text{`('}\ \textit{FoldOp}\ \text{`,'}\ \textit{Expr}\ \text{`,'}\ \textit{Expr}\ \text{`)'} \\
\textit{FoldOp} & \Rightarrow & \text{`}+\text{'} \mid \text{`}-\text{'} \mid \text{`}*\text{'} \mid \text{`}/\text{'} \mid \text{`\&\&'} \mid \text{`||'} \mid \textit{Id} \\
\textit{ExprList} & \Rightarrow & \textit{Expr}\ (\ \text{`,'}\ \textit{Expr}\ )^*
\end{array}
$$

Fig. 3: With-Loop Syntax

this sort of matrix indexing can be used both on the left hand and right hand side of an assignment operator. Looking back at Figure 1 we can see in line 13 inside the body of the with-loop that for every point (i, j) we are extracting a chunk of mat along the third dimension and providing it as an argument to the function $temporalMean$.

3) Arithmetic: Next, we have overloaded the arithmetic and comparison operators in the host language, allowing these operators to be used for matrix arithmetic. Our extended type system is able to verify that these operations are only performed on matrices of the same type and rank. Additionally, we are able to perform arithmetic between matrices and scalar values. For the multiplication operator, we have added another operator to denote element wise multiplication as opposed to matrix multiplication in the linear algebra sense. All other operators are all element wise operators.

4) With-Loop: The next feature we have implemented is the with-loop from the Single Assignment C (SAC) language. The with-loop has the following syntax specified in Figure 3

The $Generator$ portion specifies a set of indices to operate over, and the $Operation$ portion specifies what is to be done over those indices. The number of expressions in both the upper bound and lower bound should match the number of *Id's* provided, which should also match the number of dimensions provided in the $Operation$. Our extended semantic analysis is able to verify that these criteria are met and can produce error messages if necessary. A with-loop $Operation$ is defined as follows:

- $genarray(shape, expr)$ – generates a new matrix with size and rank according to that of $shape$, where each element in the set of indices specified by the generator is equal to $expr$ and 0 elsewhere. We see an example of this in Figure 1 in lines 12-13, where we are generating a matrix of size $m\ x\ n$ where each point in the matrix is the result of calling *temporalMean*. Note that the shape used in the operation may differ from the indices defined in the generator, however, the shape in the operation must be a superset of the indices in the generator, which is something that can be checked at runtime. The idea behind this is that the programmer can perform these operations on subsets of a matrix, rather than being forced to fill out the whole matrix.
- $fold(foldOp, baseVal, expr)$ – extracts the elements specified by the generator and folds them up using the $foldOp$ operator, starting with the value $baseVal$. Again, looking at Figure 1 we can see in lines 4-5 we are adding up all elements of mat by folding the operator $+$ over them. We then divide by the total number of elements n to compute the mean.

```
Matrix int <2> connComp(Matrix float <2> ssh){
  Matrix int<2> labels =
    init(Matrix int<2>, 721, 1440);
  for(int i = -100; i < 100; i++){
    Matrix bool<2> binary = ssh < i;
    ...//compute connected components
  }
  return labels;
}

int main(int argv, char ** argv) {
  Matrix float <3> ssh =
    readMatrix("ssh.data");
  Matrix int <1> dates =
    readMatrix("dates.data");
  ssh = ssh[::, ::, dates >= 01012000];
  Matrix int <3> labels =
    matrixMap(connComp,bw,[0,1]);
  writeMatrix("eddyLabels.data", labels);
}
```

Fig. 4: Connected Component Matrix Map Example

The with-loop provides our extended language a concise way for specifying complex matrix operations. Also, because of the semantics of the generator we have a unique set of elements to access allowing us to perform these operations in parallel, which is discussed in further detail in Section IV.

5) Matrix Map: Lastly, we have the matrix-map construct, which is quite similar to the with-loop in that it allows the programmer to map a function over a matrix, but this allows one to compute over ranges of a matrix, rather than element by element. More specifically, the programmer specifies what dimensions he/she would like to apply the function to, and then the rest of the dimensions are implicitly iterated over. The syntax for the matrix map is

$$
\begin{array}{l}
\textit{Expr} \Rightarrow \textit{matrixMap}\ \text{`('}\ \textit{Id}\ \text{`,'}\ \textit{Expr}\ \text{`,'} \\
\qquad\qquad\qquad\qquad \text{`['}\ \textit{Expr}\ (\text{`,'}\ \textit{Expr})^*\ \text{`]'}\ )
\end{array}
$$

Figure 4 shows an example to help clarify this idea. The goal of this sample program is to take a three dimensional spatio-temporal data set, where for each point in time, we want to label all connected components in space.

To accomplish this, we have written a function, *connComp*, which takes in a two dimensional matrix and labels all connected components. We then map this function, using the *matrixMap*, over the three dimensional matrix, specifying that we want this function applied to the first and second dimensions (denoted by $[0, 1]$) and then a loop iterating over the third dimension is generated. This can intuitively be thought of as the following few lines of semantically equivalent code in Figure 5.

```
for(int i = 0; i < dimSize(bw, 2); i++){
  result[::,::,i] = connComp(bw[::,::,i]);
}
```

Fig. 5: Semantically equivalent code fragment.

This however, becomes much more complicated as the matrix indexing generates more loops, and since this can be run in parallel, we actually lift this out into a new function so that the spawned threads can get direct access to it. One important thing to note here is that when using the matrix-map, the result is always the same size and rank as the matrix getting mapped over, though a generalization of this extension that removes this restriction is being developed.

At first glance, these abstractions may seem general purpose and not specific to the spatio-temporal data mining domain. While this may be true, we designed this language extension with a number of data mining applications in mind. These abstractions have previously been shown to work well in general, however, we found them to be especially useful in our applications.

B. General Purpose Extensions

In addition to these domain-specific extensions, we have also added a few general purpose extensions.

1) Tuples: The first is a notion of tuples similar to what is done in the functional languages such as ML and Haskell. In MATLAB, one can return multiple arguments from a function, which is used commonly in many of the algorithms that we have worked with. Tuples give us a way of doing this same thing, however, they are more general and can be used universally, rather than only with functions. The features provided with the tuple extensions are:

- Tuple declaration – Tuples can be declared by putting a list of comma separated type expressions inside parenthesis, such as "$(int, float, bool)\ t$;" which creates a tuple named t with three elements where the first is an integer, second is a float, and the third is a boolean.
- Element selection – The programmer can select an element out of a tuple by using the hash symbol followed by an integer, such as "$\#1(t)$" which will select the first element out of the tuple, in this case an element of type int.
- Anonymous creation – A tuple can also be created anonymously by putting a list of comma separated expressions inside parenthesis. This is what is commonly done when returning multiple expressions from a function, such as "$return\ (x, y, z)$;".
- Tuple assignment – Tuples can also be assigned to anonymously, such as "$(a, b, c) = f()$;"

In addition to the useful syntax, we can also perform some semantic analyses to ensure that

- In a tuple selection, the programmer doesn't try and select a field that doesn't exist. Continuing with the previous example, "$\#4(t)$" would produce a type error and a message saying that the tuple selection is out of bounds.
- All corresponding members of two tuples are of the same type when assigning two tuples. This is done by recursively checking the types of all members as tuples can be nested within tuples, so "$t = (1, 2, 3)$;" (for the declaration of t above) also creates a type error because the last element is not of type $bool$.

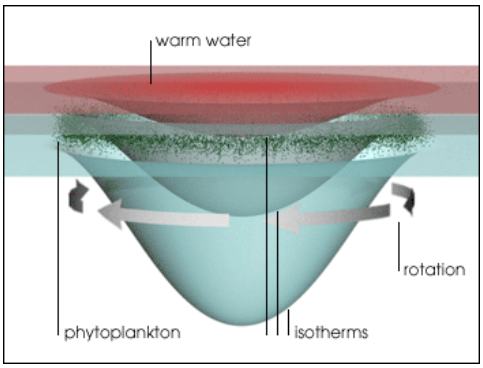


Fig. 6: [8] Rendering of a Cyclonic Ocean Eddy.

2) Reference Counting Pointers: A second general purpose extension that we have added is the reference counting pointer. Reference counting provides us a way of automatically managing memory. The idea is that we attach an extra 4 bytes to every piece of memory that gets allocated as a reference counting pointer, and use this extra 4 bytes to keep track of how many live references there are to that block of memory. If another variable also becomes a reference for that same piece of data, then we increment this counter by one. Anytime a variable goes out of scope, or gets assigned a new piece of data, then we decrement its reference counter by one. If a reference counter ever reaches zero, then we free that data. Reference counting pointers can be declared in our extended language in the same way a regular pointer can be, except the caret symbol (^) is used instead of the star (*) symbol in the type expression. Other than these two differences, reference counting pointers behave in the same way as regular pointers.

III. Ocean Eddy Application

We now shift our focus to a more specific application within spatio-temporal data mining: identifying and tracking ocean eddies. Mesoscale ocean eddies are rotating pools of water in the ocean spanning tens to hundreds of kilometers which can last anywhere from a few days to several months. Ocean eddies are an important phenomenon to monitor as they dominate the ocean's kinetic energy and are responsible for the transport of heat, salt, nutrients, and energy across the oceans [9].

Figure 6 is a NASA image [8] showing that the rotating nature of ocean eddies makes them identifiable in sea surface height (SSH) data, as it causes the center of the eddy to be lower in height compared to its perimeter. One can identify ocean eddies algorithmically by iteratively thresholding the SSH data and searching for connected components that satisfy

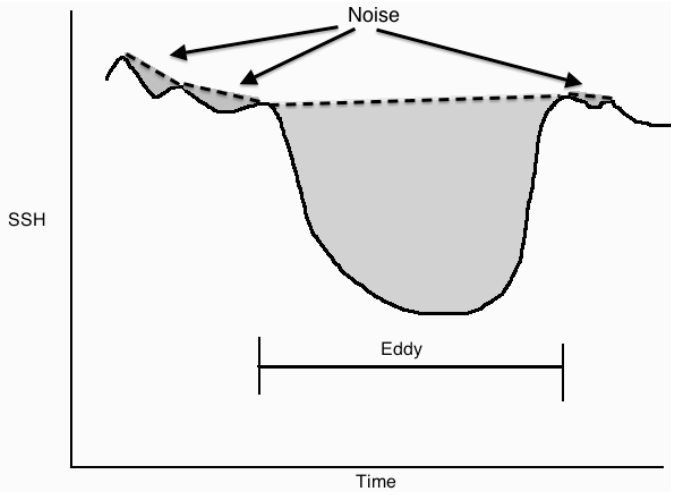


Fig. 7: Sea surface height profile of a single point affected by an ocean eddy over time and computed areas.

certain criteria that are typical of ocean eddies. The problem with this is that it is susceptible to noise in the sea surface height data collected from satellites. One problem with this is that the detection algorithm will miss an eddy for a given time frame, which can have significant impacts on the tracking results [10].

One way to circumvent this issue is to incorporate time into the detection scheme. As ocean eddies travel along a path, they leave a signature in the SSH data over time. In Figure 7 we see such a signature. The graph shows the SSH time series (solid black line) for a given point in on the ocean surface over some number of weeks. What has happened is an eddy traveled through this point, causing the sea surface height to lower, and then as the eddy passed through the point it began to rise again. Both before and after the eddy passed through, there are small "bumps" in the time series, these can be attributed to both the restlessness of the ocean, and inaccurate noisy readings from the satellites. We can quantify these signatures left by eddies in the SSH data by searching for two local maxima in each direction of a local minima, and computing the "area" between that trough and an imaginary line going from peak to peak, as seen in Figure 7. Large areas will then correspond to segments of the time series (troughs), that underwent substantial drops and rises, and those that are shallow, such as the first two and last segments, can be associated with noise. Each point in the trough is then assigned this area, which serves as a way of ranking locations on the map by how likely it is that what is being detected is actually an eddy and not an illusion created by noise.

Figure 8 shows how we can encode such an algorithm using our version of C extended with the language extensions described above. What we have is a function, *scoreTS*, that computes a score for every point in a given time series. In the main function of the program, we map this scoring function over each point in space, applying it to all elements in the third dimension as seen in line 58 (recall dimensions begin at 0, so 2 corresponds to the third dimension).

Within the scoring function, we begin by trimming off the beginning of the time series until we reach our first local maxima. We then continually cut out a chunk of the time series as extracted by the $getTrough$ function, and compute the area of that trough using the $computeArea$ function. Each point on that trough is then assigned the computed area, and put back into the original time series.

The $getTrough$ function simply begins by starting at the first local maxima, and then walks over the time series until it encounters another local maxima, returning this sub-sequence of the data, the beginning index, and the ending index in the form of a tuple.

The $computeArea$ function then computes the equation of a line based off of the two local maxima. In line 27 we compute the dotted line seen in Figure 7 first by creating a one dimensional array containing the elements between zero and the length of $areaOfInterest$, which then gets multiplied by the slope of the line, and added to the y-intercept. In lines 28-32, we then compute the area by subtracting the trough from this line, and sum the differences using a with-loop; lastly, we create an array the same length of the $areaOfInterest$ where each point is the computed area.

IV. Parallel Code Generation

The with-loop and matrix-map extensions specify computations that can be easily parallelized, and we use PThreads to implement these parallel computations. Figure 9 shows the performance gain relative to the sequential version of this algorithm run on a machine with 12 cores (2 Intel 6 core Xeon processors, 2.0Ghz, with 16GB RAM running Ubuntu GNU/Linux 12.04). We see that not only are we able to concisely implement this algorithm using our language extensions, but we are also able to get almost linear speedup, scaling up to 12 cores. We did not attempt to use more.

In order to achieve the sort of performance seen in the previous section, there are a number of issues that we need to address. A naive translation to PThreads will not achieve this linear speedup. First, there is the issue of thread management overhead. Most multi-threaded programs adopt the fork-join model, where threads are spawned when they are needed, and then destroyed as soon as that parallel computation is done. If there is a lot of disjoint parallel computation to be done, then the programmer pays the price of creating and destroying threads each time. To mitigate this problem, we have adopted the enhanced fork-join model from SAC [11] where we spawn the necessary number of threads at the beginning of program execution and send them straight into a spin lock where they sit idle until some parallel work is to be done. When a parallel construct is encountered by the main thread, it flips the condition that keeps the threads spinning, which releases all of them at once to execute the work in parallel. As soon as each thread is done, it passes through a stop barrier and goes right back into the spin lock. The main thread then waits in the stop barrier until all threads have completed their work.

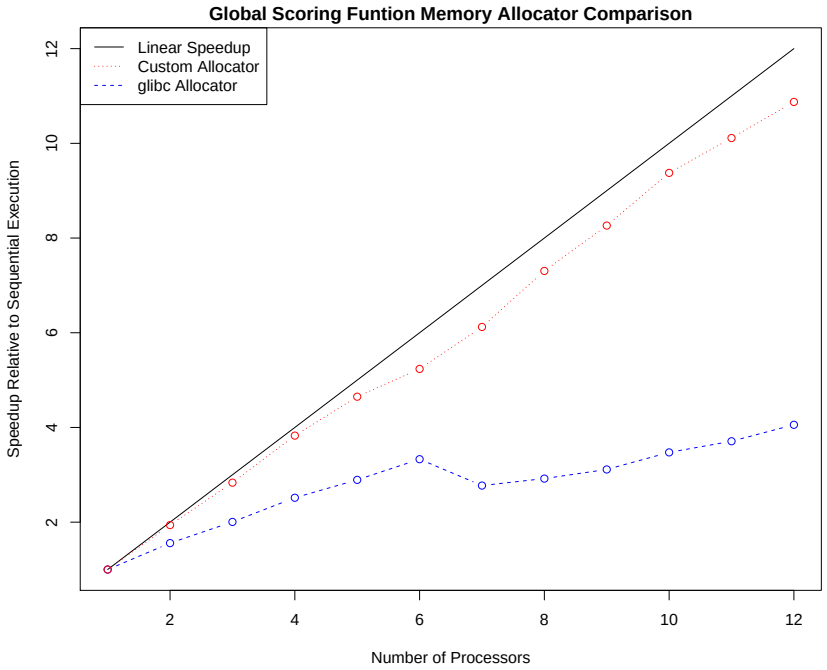


Fig. 9: Performance Statistics of the Ocean Eddy Scoring Function on 1-12 Cores with custom memory management improvements and with standard C library memory allocator.

Another problem that needed to be dealt with was memory management. Some sort of memory management in a domain such as this is absolutely essential. The dimensions of the SSH dataset used in the scoring algorithm are $721 \times 1440 \times 954$, which comes to about 7 gigabytes of data, not to mention all of the intermediate matrices that get created in computing these scores. To take care of this problem we build the underlying implementation of matrices on top of the reference counting pointers described in section II-B2.

This sort of memory management works very well in this setting, as most allocations made are relatively infrequent and large compared to those made in say a functional programming language. Another benefit to reference counting is that the overhead associated with memory management is more evenly distributed over the lifetime of the program compared to garbage collection [12].

A third obstacle that we were faced with was memory allocation. The issue is that some implementations of malloc are naïvely implemented using a mutex lock to deal with contention over the heap. More recent implementations separate the heap into "arenas" as soon as contention is detected [13], however our results and those shown in [12] show that these "off the shelf" memory allocators do not scale well in this setting. This can be seen in Figure 9, where the higher dashed line represents the relative speedup using the memory allocator that we have implemented with these extensions, and the lower dotted line represents the same execution, but using the standard glibc memory allocator. We can see that the memory allocator that we have implemented scales almost perfectly with the number of cores, but the standard implementation of malloc tapers off around 6 cores. We briefly explored the use of the Hoard allocator, however, as noted in [12], when allocation requests exceed a certain threshold, they are mapped directly to virtual memory using $mmap$ and $munmap$ which incurs significant overhead.

The basic idea behind our memory allocator is to create a separate heap of a fixed size for each thread at the time of program creation. Each thread only requests memory from its personal heap. If its heap ever becomes depleted, then a new block of memory of the same fixed size is added to its heap's free list, unless the requested size is greater than the fixed size, in which case the requested size is added to the heap and immediately given to that thread.

V. Building Extensible Languages with Silver

Extensible languages are defined by two primary components: a specification for a host language and a set of specifications for the language extensions to that host language. The extensions are constructed by independent parties focusing on their own domain-specific language extensions that may introduce new syntax (notations), semantic analyses (error checking), and optimizations to the host language and its translator. These developers need have no knowledge of one another. The programmer using an extensible language is free to choose the set of extensions that fit his or her problem at hand and direct a set of translator-generating tools to construct the translator for their customized language. This translator reads programs in the extended language, performs semantic analysis for type checking and optimization, and then outputs a semantically equivalent program written in the host (non-extended) language.

```
1 (Matrix float <1>,int,int)
2 getTrough(Matrix float<1>ts,int i) {
3   int beginning = i;
4   int n = dimSize(ts, 0);
5   //Walk Downwards
6   while(i+1 < n && ts[i] >= ts[i+1])
7       i = i+1;
8   //Walk Upwards
9   while(i+1 < n && ts[i] < ts[i+1])
10      i = i+1;
11  //Return the trough
12  return (ts[beginning::i], beginning, i);
13 }
14
15 Matrix float<1>
16 computeArea(Matrix float<1> areaOfInterest) {
17  float y1 = areaOfInterest[0];
18  float y2 = areaOfInterest[end];
19  int x1 = 0;
20  int x2=dimSize(areaOfInterest,0)-1;
21  //compute slope
22  float m = (y1-y2) / ((float)(x1-x2));
23  //compute y intercept
24  float b = y1 - m*x1;
25  Matrix float<1> Line = (x1::x2)*m+b;
26
27  float area = with([x1] <= [i] < [x2])
28    fold(+,0.0,line-areaOfInterest);
29
30  return
31    with([0] <= [i] < [dimSize(Line, 0)])
32      genarray([dimSize(Line, 0)], area);
33 }
34
35 Matrix float<1>scoreTS(Matrix float<1>ts) {
36  Matrix float<1>scores =
37    init(Matrix float <1>, dimSize(ts, 0));
38  int i = 0;
39  while(ts[i] < ts[i+1])  // trimming
40      i = i+1;
41  int n = dimSize(ts, 0);
42  int beginning;
43  Matrix float [0] trough;
44  while(i < n-1) {
45    (trough, beginning, i)
46      = getTrough(ts, i);
47    scores[beginning::i]
48      = computeArea(trough);
49  }
50  return scores;
51 }
52
53 main (int argc, char **argv) {
54  //Shape of SSH: 721 x 1440 x 954
55  Matrix float <3> data
56    = readMatrix("ssh.data");
57  Matrix float <3> scores;
58  scores = matrixMap(scoreTS, data, [2]);
59  writeMatrix("temporalScores.data", scores);
60  return 0;
61 }
```

Fig. 8: Ocean eddy scoring implementation.

Language translation/compilation can be split roughly into two steps. First, the text is converted into a syntax tree by using a scanner and parser. Second, semantic analysis is run on the syntax tree and, if it passes, translates the syntax tree into the required output (e.g. executable, host language, etc). Generating these tools from a composition of specifications presents a number of interesting challenges.

A. Scanning and Parsing

The language's scanner converts text into a stream of tokens based on a set of regular expressions for terminal symbols (keywords, literal integers, variables, etc). This stream of tokens is supplied to the parser which generates the syntax tree according to a context-free grammar specification that defines the language syntax.

It is possible that two different languages will want to use the same keyword (such as "with" in the with-loop construct). To solve this, Copper, our parser and scanner generator, constructs a context-aware scanner [3]. Such a scanner uses the "context" of the parser to determine which of the overlapping keywords is to be recognized. In cases where context is not enough, it allows the programmer to add a simple annotation to determine which extension the keyword is to belong to. Details of this process can be found in a previous paper [3].

There can also be problems in the parsing of extended languages in which the composed context-free grammar is ambiguous. To avoid this problem we require that the composed grammar be in the class of deterministic (and thus unambiguous) class LALR(1) [4]. The restrictions of LALR(1) are significantly eased by using a context aware scanner since overlapping keywords, and more generally, overlapping regular expressions for terminals, are allowed.

Because the composition of LALR(1) grammars does not always result in an LALR(1) grammar, an analysis has been developed that imposes further restrictions on the contents of extension grammars to ensure that the composition of these restricted grammars is LALR(1) [5]. The analysis is run by the extension developer on his/her extension to check that it is in this restricted class. Formally, this can be stated as follows:

$$\begin{aligned}&(\textbf{for each } i \in [1,n].\ isLALR(CFG^H \cup CFG^E_i) \wedge \\ &\qquad\qquad isComposable(CFG^H,\ CFG^E_i)\ ) \\ &\Rightarrow isLALR(CFG^H \cup \{CFG^E_1, \ldots, CFG^E_n\})\end{aligned}$$

It is the *isComposable* check that determines if the extension grammars are in the more restricted set. The important point about this analysis is that if the programmer chooses only extensions that have passed the analysis, then they have a strong guarantee that the generated scanner and parser for their chosen set of extensions will be LALR(1) and thus a working and correct scanner and parser can be generated.

The domain-specific matrix extension does pass this test. The tuples extension does not, however, since the initial symbol for tuple expressions is a left-paren, "(", which violates the restriction that a unique initial terminal symbol is needed on extension syntax. The "with" terminal, for example, is such an initial terminal. Thus the tuples extension will be packaged

as part of the host language. One could modify the tuple terminals to be "(." and ".)" and thus be distinguishable from other symbols in the host language and thus pass this analysis. For a complete discussion of this analysis and the restrictions it imposes please see the original paper [5].

B. Semantic Analysis and Translation

The language extension described in this paper utilizes semantic analysis to find errors and, if none are found, outputs a C translation of the code. These operations are specified using Silver [2], a tool for language developers to create extensible languages through attribute grammar specifications.

Attribute grammars (AG) [1] provide a means for decorating program syntax trees with attributes describing, for example, the type of an expression, or the C translation of a statement. These values are computed based on a set of equations that define attribute values based on other attributes in the tree. The problem with composition of AGs is that composition does not guarantee that the resulting AG is well-defined (that all needed attributes have defining equations). Silver has a modular well-definedness analysis, similar in format to the modular determinism analysis for Copper described above, that extension designers can run on their extension. It guarantees that if only extensions that pass this analysis are chosen, then the composition of them will be well defined. All extensions described above pass this analysis. Full details of this analysis can be found in a previous paper [14].

VI. Conclusion

A. Related Work

There have been many efforts to improve the practice of scientific software development. These include the development of new programming languages and language constructs allowing one to write programs at a high-level level of abstraction, as compared to low-level constructs in languages such as C and FORTRAN. For example, constructs such as *nested* data parallel arrays can be found in domain-specific languages such as NESL [15] that improve on performance and the ease of use of *flat* data parallel arrays found in versions of Fortran. Fortress [16], X10 [17], and Chapel [18] are entirely new languages that have also been developed. However, many of these developments have not found as wide of an audience as they might warrant, for various technical and non-technical reasons. Sometimes the new constructs that may be useful to a developer cannot be used because they are spread across multiple incompatible languages or the domain-specific language that might be used does not fit into the existing development process for an existing application.

Thus an *extensible* approach in which new domain-specific features, as chosen by the application developer in the same way that he or she chooses to use different libraries in traditional development, may be a viable alternative to these *monolithic* approaches described above. We are not the only ones interested in extensible languages and tools. One approach is to use domain-specific "embedded" languages, in which domain-specific features are added as libraries but to host languages that can give these features the same look and feel of host language features. The DeLite project is a prime example of this in which the host language is Scala; of note is the use of a meta-programming technique called lightweight modular staging [19] which allows the Scala code to analyze and optimize code. There are also approaches, similar to ours, in which compilers and translators are generated from some (composed) declarative specifications of language syntax and semantics. JastAdd [20], also based on attribute grammars has been used to develop an extensible Java compiler [21] and the Spoofax [22] system, based on the Stratego term-rewriting system [23] has also been used to add domain-specific and general purpose extensions to Java [24]. These approaches are similar in spirit to ours, however these tools do not offer the modular analysis that ensure that the composition will actually result in a working compiler or translator. They often do work just fine, though someone with knowledge of language design is often needed to be involved in the composition process. This differs from our approach in which we expect non-language experts to simply pick the extensions that they want to use and have a guarantee that they will, in fact, work together.

B. Discussion

Many components of the matrix language extension shown above are based on MATLAB and the SAC language. Our aim here is not to claim that these language features are of our creation or that these are the best ones for this application, but to demonstrate that such features can be provided to programmers as modular, "plug-able", additions to their programming language. This is not possible with MATLAB and SAC as they are monolithic languages not designed to be extended.

It is our hypothesis that by giving programmers the freedom to choose the (set of) domain-specific features that fit their task at hand will provide a better solution than requiring programmers to pick the (combination of) monolithic, stand-alone languages that happen to have the language features that they feel that they need. Our work in progress reported here shows that interesting language features from existing scientific and high-performance languages can be added as language extensions.

Our future work is to add to this set of features with additional constructs useful in scientific and high-performance computing, and to provide additional optimizations of program computations and to the generated code.

We are also in the process of developing an extensible specification of the full version of C programming language that these kind of language extensions can be added to. This will enable programmers to experiment with these kinds of language extensions on existing C applications. This allows for an incremental approach to using these tools. Programmers can easily experiment with an extension or two on an existing program without having to rewrite the entire program into another language. This ease of use is a significant advantage for extensible approaches and one we expect to leverage once the full C host language is complete and a rich set of extension are available for use.

ACKNOWLEDGMENT

We thank Vipin Kumar, Micheal Steinbach, James Faghmous and the rest of the members of the University of Minnesota Climate Data Mining group for their collaboration. This work is partially supported by NSF Award #0905581.

REFERENCES

[1] D. E. Knuth, "Semantics of context-free languages," *Mathematical Systems Theory*, vol. 2, no. 2, pp. 127–145, 1968, corrections in **5**(1971) pp. 95–96.

[2] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, "Silver: an extensible attribute grammar system," *Science of Computer Programming*, vol. 75, no. 1–2, pp. 39–54, January 2010.

[3] E. Van Wyk and A. Schwerdfeger, "Context-aware scanning for parsing extensible languages," in *Intl. Conf. on Generative Programming and Component Engineering, (GPCE)*. ACM, 2007, pp. 63–72.

[4] A. Aho, R. Sethi, and J. Ullman, *Compilers – Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[5] A. Schwerdfeger and E. Van Wyk, "Verifiable composition of deterministic grammars," in *Proc. of Conf. on Programming Language Design and Implementation (PLDI)*. ACM, June 2009, pp. 199–210.

[6] T. Kaminski and E. Van Wyk, "Integrating attribute grammar and functional programming language features," in *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, ser. LNCS, vol. 6940. Springer-Verlag, July 2011, pp. 263–282.

[7] C. Grelck and S.-B. Scholz, "SAC: a functional array language for efficient multi-threaded execution," *Int. J. Parallel Program.*, vol. 34, no. 4, pp. 383–427, Aug. 2006.

[8] Nasa earth observatory. [Online]. Available: http://earthobservatory.nasa.gov/Features/Eddies

[9] L. Fu, D. Chelton, P. L. Traon, and R. Marrow, "Eddy dynmaics from satellite altimetry," *Oceanography*, vol. 23, pp. 14–25, Feb 2010.

[10] J. Faghmous, M. Uluyol, L. Styles, M. Le, V. Mithal, S. Boriah, and V. Kumar, "Multiple hypothesis object tracking for unsupervised self-learning: An ocean eddy tracking application," in *Proc. 27th AAAI Conference on Artificial Intelligence*, 2013.

[11] C. Grelck, "Shared memory multiprocessor support for functional array processing in sac," *J. Funct. Program.*, vol. 15, no. 3, pp. 353–401, May 2005.

[12] C. Grelck and S.-B. Scholz, "Efficient Heap Management for Declarative Data Parallel Programming on Multicores," in *3rd Workshop on Declarative Aspects of Multicore Programming (DAMP'08), San Francisco, CA, USA*, M. Hermenegildo, L. Peterson, and N. Glew, Eds. ACM Press, 2008, pp. 17–31.

[13] Linux programmer's manual. [Online]. Available: http://man7.org/linux/man-pages/man3/malloc.3.html

[14] T. Kaminski and E. Van Wyk, "Modular well-definedness analysis for attribute grammars," in *Proc. of Intl. Conf. on Software Language Engineering (SLE)*, ser. LNCS, vol. 7745. Springer-Verlag, September 2012, pp. 352–371.

[15] G. E. Blelloch, J. C. Hardwick, S. Chatterjee, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," in *Proc. of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP)*. New York, NY, USA: ACM, 1993, pp. 102–111.

[16] S. Microsystems, "The fortress language specification, version 1.0," March 2007, http://research.sun.com/projects/plrg/fortress.pdf.

[17] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, 2005.

[18] Cray, "Chapel language specification 0.750," 2007, http://chapel.cs.washington.edu/spec-0.750.pdf.

[19] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Proceedings of the ninth international conference on Generative programming and component engineering*, ser. GPCE '10. New York, NY, USA: ACM, 2010, pp. 127–136.

[20] T. Ekman and G. Hedin, "The JastAdd system - modular extensible compiler construction," *Science of Computer Programming*, vol. 69, pp. 14–26, December 2007.

[21] ——, "The JastAdd extensible Java compiler," in *Proc. of Conf. on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*. ACM, 2007, pp. 1–18.

[22] L. C. L. Kats and E. Visser, "The Spoofax language workbench. Rules for declarative specification of languages and IDEs," in *Proc. of Conf. on Object Oriented Programming, Systems, Languages, and Systems (OOPSLA)*. ACM, 2010, pp. 444–463.

[23] E. Visser, "Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9," in *Domain-Specific Program Generation*, ser. LNCS, C. Lengauer *et al.*, Eds. Spinger-Verlag, June 2004, no. 3016, pp. 216–238.

[24] M. Bravenboer and E. Visser, "Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions," in *Proc. Conf. on Object-oriented programming, systems, languages, and applications (OOPSLA)*. ACM, 2004, pp. 365–383.