Climate Data Domain Specific Language

Yannan Wang, Ark Xu, Matthew Le
Faculty Supervisor: Erik Van Wyk
04/18/2012

Computer Science & Engineering
Minnesota Extensible Language Tools

University of Minnesota
College of Science & Engineering

Domain Specific Languages

What are Domain Specific Languages?

- A programming language that is dedicated to a particular problem domain.
- Typically have a higher level of abstraction than multi-purpose programming languages such as C, C++, Java, etc.
- Syntax of the language forms nicely to the problem.

Why Use Domain Specific Languages?

- New language constructs specific to the problem domain
  - Ex: Matrices and Lists.
- Domain specific optimizations
  - Matrix operations such as multiplication can be run in parallel
  - List operations such as map and fold can also be run in parallel
- Error checking is provided for the high level code
  - Ex: multiplying a 10x10 by a 20 x 20 matrix. In our domain specific language, a dimension mismatch error is given, however, in C a segmentation fault occurs.
- Higher level of abstraction
  - Code is easier to understand, so fewer mistakes are made
  - 2002 study reports that developers spend 80% of their time fixing bugs rather than writing code. [1]

Introduction

Climate change is the defining environmental challenge facing our planet. Climate researchers deal with very large data sets, and are often faced with the challenge of writing efficient, scalable, and portable data- intensive applications. Are research goals are:

- Translate DSL code to C code.
  - C is typically faster than Matlab and has more opportunities for optimization.
- Make use of parallelization libraries such as OpenMP and CUDA
- Implement the domain specific language as an extension to C
  - Able to gain benefits of a domain specific language without losing the vast features of a multi-purpose language

Outline

Climate Data

In the past few years, climate data has become very abundant. The problem is that climate scientists typically have little knowledge when it comes to computing, and computer scientists typically have little knowledge of climatology. The domain specific language that we are developing aims to close this knowledge gap. The following data structures and functions help to do just this.

- **Matrices** – Much of the climate data comes in a form that is easy to represent as a matrix. Ex: Sea surface temperature has an associated location and date, so we can represent this as a 3-dimensional matrix to keep all data organized.
- **Read matrix function** – File input/output is a tedious task, we have provided a function called readMatrix, which will do all of this for the programmer.
- **Overloaded arithmetic operators** – multiplication is a fairly complex task when dealing with matrices. We have overloaded the '*' to perform matrix operation so the programmer doesn't have to worry about it.
- **Tic/Toc timer** – Many climate data algorithms are computationally expensive and runtime is an important factor. We have added a very simple and easy timing mechanism. The programmer writes their code in between the tic and toc keywords and the time it takes to compute is printed to the screen.
- **Transpose** – data frequently needs to be manipulated. One way of doing this is transposting a matrix (switching the rows and the columns. This can be a tricky task, however a matrix can easily be transposed by adding a single quote or 'prime' after a matrix name.
- **Lists –** Lists are a special type of matrix (one dimensional) that have special operations that can be performed on them. Another nice attribute that lists have is that they can easily change their size.
- **Map –** map is function that allows the programmer to apply a function to every element within a list.
- **Fold –** compacts a list into a scaler based on a function given to it
- **Filter –** given a condition, filter removes all elements of a list that do not meet the condition.

Acknowledgements

We would like to acknowledge the National Science Foundation's REU program for giving us this opportunity and funding our research.

Result

```
Matrix comparisonMatrix[years, years] i,j =
    if j <= i then 0.0
    else
        let
            // diff these two years
            Float diff;
            diff = 0;
            Integer k;
            for(k = 0 to season_length-1) {
                diff = diff + pt[i,k] - pt[j,k];
            }
        in
            diff / season_length
        end;
```

⬇

```
   Matrix_t * comparisonMatrix;
   comparisonMatrix = (Matrix_t*)malloc(sizeof(Matrix_t) * 1);
   comparisonMatrix->data = (float *)malloc(sizeof(float) * (years
* years));
   comparisonMatrix->rows = years;
   comparisonMatrix->cols = years;
   {
      int i;
      i = 0;
      while ( ( !(( !(i < years-1 )))) )
      {
         int j;
         j = 0;
         while ( ( !(( !(j < (years - 1)))) )
         {
     comparisonMatrix->data[((i * comparisonMatrix->cols) + j)] =
(( !(( !(j < i)) && ( !(j == i)))) ? 0.0 : ({
            double diff;
            diff = 0;
            int k;
            k = 0;
            while ( ( !(( !(k < season_length - 1 )))) )
            {
               {
                  diff = ((diff + pt->data[((i * pt->cols) + k)]) -
pt->data[((j * pt->cols) + k)]);
               }
               k = (k + 1);
            }
             diff / season_length ;}));
            j = (j + 1);
         }
         i = (i + 1);
      }
   }
```