Is D the Answer to the One vs. Two Language High Performance Computing Dilemma?

Ralph Butler, Chrisila Pettey, and Matthew Wang
Department of Computer Science, Box 48
Middle Tennessee State University
Murfreesboro, Tennessee 37132, USA

(ralph.butler, chrisila.pettey)@mtsu.edu, mw3n@mtmail.mtsu.edu

Abstract – *During the course of our careers we have written a variety of high performance computing programs (parallel Genetic Algorithms [12, 13], genetic sequence analysis [4], automated reasoning applications such as theorem provers [5], and an asynchronous, dynamic, load-balancing library [1, 3, 7, 10]). In our projects we often found ourselves using a bilingual programming model to gain the advantages of both high performance execution and advanced language features like garbage collected data structures. D gives us access to both facilities in a single language. While D [8] is not a new language in the Computer Science scheme of things, it has only recently advanced to the point where we could begin to consider it as a solution to our dilemma. In this paper we discuss how we use D for rapid development of high performance code, and how we link it to legacy code such as MPICH2 [11] and ADLB.*

Keywords: D Programming, Bilingual Computing, High Performance Computing

1 Introduction

As computer scientists we recognize the value in learning multiple languages. Each language brings a unique set of features to the table that has the potential for providing elegant solutions to various problems. However, in a perfect world, we would only need to be an expert in one language, and that language would have all the features needed to easily create well-designed code. Since the world is not perfect, we have typically in the past focused on one language per project. In practice, the high performance world typically relies on Fortran or C, and in our case it was C. But the lure of scripting languages such as Python would occasionally draw us in. When our project needed the ease of built-in advanced data structures such as associative arrays along with automatic garbage collection of those structures, we used a high level scripting language such as Python. When high performance was absolutely critical or if we were doing low-level programming such as memory management, then we typically used some language that was a C-derivative.

Over time, we began to incorporate both scripting and compiled languages into one project in such a way that distinct pieces of the project might be in different languages, but the interaction between pieces written in different languages was minimal. For example, initially the *mpd* component of *MPICH2* [11] was written in Python while the rest of the modules were written in C [9]. The *mpd* component is a stand-alone process management system that has a trivial interface for connecting to other systems.

Because we always want both performance and ease of development, we have ended up doing bilingual computing. In the bilingual model of computing the interactions between the components written in different languages involve the concept of shared high-level data structures. The reason for two languages was sometimes to overcome a missing feature in C (the high-level data structure) [5]. And sometimes we used two languages to overcome a limiting feature in Python (the global interpreter lock) [6].

However, bilingual computing can be problematic. Besides the obvious requirement of being expert in two languages, bilingual computing introduces the problem of what parts of the project to do in each language and how to interface the various pieces with each other. These are not always trivial problems to overcome. So we are left with the dilemma of do we choose only one language for a project and lose important features of the other language, or do we choose two languages and deal with the problems of interfacing the two languages? And if we choose two languages, someone has to maintain both. As we mentioned previously, MPICH2 was written in C, but its process management component was written in Python. The MPICH2 team, however, felt compelled to rewrite the *mpd* component in C not for performance reasons but largely for ease of maintenance.

The solution to our dilemma is that we need a single programming language that has all the features to elegantly solve our problems. The D programming language [8] was initially developed with the idea of improving C++, and has recently stabilized into what we believe to be the answer to our dilemma. In this paper we begin by discussing salient attributes of D. We then describe our interfaces to legacy code – specifically MPICH2 [11] and ADLB [1, 3, 7, 10].

2 Why D?

The obvious question to ask is, "Why D?" It doesn't even appear on the TIOBE Programming Community Index [14] of the top 20 most used programming languages. It is interesting to note, however, that the top five programming languages on the list are C derivatives, and four of the remaining languages in the list are scripting languages.

As mentioned before, D was initially developed with the idea of improving C++. Specifically, the developers asked this question:

> Can the power and capability of C++ be extracted, redesigned, and recast into a language that is simple, orthogonal, and practical? Can it all be put into a package that is easy for compiler writers to correctly implement, and which enables compilers to efficiently generate aggressively optimized code? [8]

Because of this, it has many of the features of C/C++ that we consider to be important for high performance computing:

- It is so compatible with C that it will link with existing C programs.
- It has all the features of C from pointers to inline assembly language.
- It has all the features of C++ including a simplified method of handling templates.
- Programming can look like C or C++.
- Performance is equivalent to C.
- The developers claim that on average it compiles 100 times faster than C++ and four times faster than GO. [2]

So we have the performance and features of the C derivative languages. But we also want the high-level data structures, automatic garbage collection, and rapid development time of scripting languages. D actually has many of these features as well. For instance:

- Rapid development can be done with the rdmd wrapper to dmd (the original D compiler) that allows for compiling, linking, and executing without appearing to compile and link.
- D has advanced data structures such as lists and associative arrays.
- D has automatic garbage collection.
- It has Perl-compatible regular expression handling.

In addition to having the aforementioned features of the C derivative languages and the scripting languages, D has some features that can be very helpful for high performance computing projects. Features such as:

- built-in language support for it's own thread model (it should be noted that memory is not shared by default, but it can easily be annotated as shared),
- built-in language support for unit test – both with compiler options and within actual code,
- facilities for contract programming

3 Interfacing D to Legacy HPC Code

Because D is link-friendly with C, i.e., it just links with C functions, one would be inclined to believe that it would also be compile-friendly with C. What we mean by that is you might think that it would allow you to include C header files. But that is not the case. Instead D imports its own modules. For example, you might expect to be able to include the mpi.h header file. Instead, the mpi.h header file has to be converted to a D

```
/* -*- Mode: C; c-basic-offset:4 ; -*-
*/
/*
 *  (C) 2001 by Argonne National
Laboratory.
 *      See COPYRIGHT in top-level
directory.
 */
/*src/include/mpi.h.  Generated from
mpi.h.in by configure.*/
#ifndef MPI_INCLUDED
#define MPI_INCLUDED
/* user include file for MPI programs */
/* Keep C++ compilers from getting
confused */
#if defined(__cplusplus)
extern "C" {
#endif
/* Results of the compare operations. */
#define MPI_IDENT     0
#define MPI_CONGRUENT 1
#define MPI_SIMILAR   2
#define MPI_UNEQUAL   3
typedef int MPI_Datatype;
#define MPI_CHAR
((MPI_Datatype)0x4c000101)
#define MPI_SIGNED_CHAR
((MPI_Datatype)0x4c000118)
#define MPI_UNSIGNED_CHAR
((MPI_Datatype)0x4c000102)
#define MPI_BYTE
((MPI_Datatype)0x4c00010d)
```

Figure 1. A portion of mpi.h.

module. To facilitate this process, there is a *htod* program. *htod* is quite useful in that it accomplishes most of the task, and most of ***what needs to be done by hand*** is annotated in the converted file with comments. Admittedly there is some hand-crafting necessary to make the project work. The transformation process is shown in Figures 1 – 3 where we list a portion of the mpi.h header file, what that same portion looks like coming out of *htod*, and what the final hand-crafted portion looks like.

```
/* Converted to D from
\mpich2i\include\mpi.h by htod */
module mpi;
/* -*- Mode: C; c-basic-offset:4 ; -*-
*/
/*
 *  (C) 2001 by Argonne National
Laboratory.
 *      See COPYRIGHT in top-level
directory.
 */
/* src/include/mpi.h.  Generated from
mpi.h.in by configure. */
//C     #ifndef MPI_INCLUDED
//C     #define MPI_INCLUDED
/* user include file for MPI programs */
/* Keep C++ compilers from getting
confused */
//C     #if defined(__cplusplus)
//C     extern "C" {
//C     #endif
/* Results of the compare operations. */
//C     #define MPI_IDENT     0
//C     #define MPI_CONGRUENT 1
const MPI_IDENT = 0;
//C     #define MPI_SIMILAR   2
const MPI_CONGRUENT = 1;
//C     #define MPI_UNEQUAL   3
const MPI_SIMILAR = 2;
const MPI_UNEQUAL = 3;
//C     typedef int MPI_Datatype;
extern (C):
alias int MPI_Datatype;
//C  #define MPI_CHAR
((MPI_Datatype)0x4c000101)
//C     #define MPI_SIGNED_CHAR
((MPI_Datatype)0x4c000118)
//C     #define MPI_UNSIGNED_CHAR
((MPI_Datatype)0x4c000102)
//C     #define MPI_BYTE
((MPI_Datatype)0x4c00010d)
```

Figure 2. A portion of mpi.h that has been run through *htod*

```
/* Converted to D from
\mpich2i\include\mpi.h by htod */
module mpi;
/* -*- Mode: C; c-basic-offset:4 ; -*-
*/
/*
 *  (C) 2001 by Argonne National
Laboratory.
 *      See COPYRIGHT in top-level
directory.
 */
/* src/include/mpi.h.  Generated from
mpi.h.in by configure. */
//C     #ifndef MPI_INCLUDED
//C     #define MPI_INCLUDED
/* user include file for MPI programs */
/* Keep C++ compilers from getting
confused */
//C     #if defined(__cplusplus)
//C     extern "C" {
//C     #endif
/* Results of the compare operations. */
//C     #define MPI_IDENT     0
//C     #define MPI_CONGRUENT 1
//C     #define MPI_SIMILAR   2
//C     #define MPI_UNEQUAL   3
const MPI_IDENT = 0;
const MPI_CONGRUENT = 1;
const MPI_SIMILAR = 2;
const MPI_UNEQUAL = 3;
//C     typedef int MPI_Datatype;
extern (C):
alias int MPI_Datatype;
//C     #define MPI_CHAR
((MPI_Datatype)0x4c000101)
//C     #define MPI_SIGNED_CHAR
((MPI_Datatype)0x4c000118)
//C     #define MPI_UNSIGNED_CHAR
((MPI_Datatype)0x4c000102)
//C     #define MPI_BYTE
((MPI_Datatype)0x4c00010d)
const MPI_CHAR =
cast(MPI_Datatype)0x4c000101;
const MPI_SIGNED_CHAR =
           cast(MPI_Datatype)0x4c000118;
const MPI_UNSIGNED_CHAR =
           cast(MPI_Datatype)0x4c000102;
const MPI_BYTE =
cast(MPI_Datatype)0x4c00010d;
```

Figure 3. A portion of mpi.h that has been run through *htod* and then hand-crafted to work correctly

While the conversion process was a bit tedious, it was not difficult, and we were successful in using a small subset of MPICH2 in D programs. We turned our results over to the current MPICH2 support team, and discussions are being held to determine if MPICH2 is going to support D as it already does C, C++, and Fortran.

Having this experience behind us, it was relatively trivial to perform the same operation on ADLB and begin using ADLB in D programs. Encouraged by these results, this semester we have provided D as an option in our graduate level parallel processing class which uses both MPI and ADLB.

As a purely intellectual exercise to convince ourselves that D would be a good replacement for C and/or Python, we did the necessary work to convert portions of mpd, ADLB, and the theorem prover into D. In cases where our major concern was nothing but performance, we found that writing D code was essentially equivalent to writing C code. If we wanted to, we could use pointers and write memory managers. On the other hand, if we wanted to let D handle memory management we could. In fact, D was able to handle practically all that we wanted it to handle. There were only two places where D was somewhat less than perfect for our needs. The first tiny flaw is that D's definition of associative arrays is not as elegant as Python's. If you want to map a single data type to another single data type, D is fine. But if you want to use multiple data types as the key, you have to use a *Variant* – so technically D is up to the task but it is not as elegant as Python. The second tiny flaw is that as far as we can tell, there is no serialization library right now in D. There is one proposed that has not been accepted. But to fully re-implement some of the mpd code, we would need serialization. As part of our exercise, we implemented a small serialization library suitable for our own needs, but it would be nice to have serialization as part of the distribution libraries.

4 Conclusions and Future Work

During the course of several decades of writing high performance code our goals have evolved. Initially the aim was simply to improve performance – which meant programming in C (or some C derivative). Over time our goals evolved to allow more features of scripting languages for small parts of a large project – features such as advanced data structures and automatic garbage collection. Eventually our goals would require the sharing of data structures between high-level scripting code and low-level performance code in a single project. The desire for both caused us to develop a bilingual programming model, thus requiring us to deal with the problems associated with sharing data structures between languages. This was not an ideal situation.

We believe the dilemma of whether to code a project in a single language or two languages can be solved by coding in D. D provides the power and capability of the C derivative languages. It also provides the flexibility of the scripting languages. And D has some added attributes that aren't available in the others. It is trivial to link D code to C code, and it is fairly easy to interface D to legacy code using the *htod* software. The interface to legacy code does require some manual labor to get the project to work, but it is minimal.

We have begun using D in our graduate Parallel Processing class as well as in our research. The elegance of the D threading model and its ease of use with MPI and ADLB have led to plans to use D next year in a Software Design and Development course which might not otherwise be able to use high performance computing facilities.

While we can use D on our own cluster, on machines like the BlueGene/Q, this might not currently be possible. Typically you can only be guaranteed C/C++ and Fortran. Since D is under consideration by the MPICH2 team, this may lead to availability of D on the bigger clusters/machines.

Just as people constantly argue about what language is better. They also have frequent discussions about whether or not languages are scalable. For example, they might say, "Perl's fine for quick and dirty hacks, but it's not scalable." Or they might say, "Python is elegant and scalable." We doubt that anyone would argue that C/C++ is not scalable. On the other hand, the large projects done in C/C++ may not be as elegant or as maintainable as those done in Python. All arguments aside, if you want performance, rapid development, elegance, scalability, safe language features, ease of maintenance, and advanced software engineering functionality, then it is really hard to find it all in one place other than D. At this point we see no reason to use anything other than D for new projects. From our point of view it scales elegantly, and gives us all the features we need.

5 References

[1] ADLB: Asynchronous Dynamic Load Balancing, http://www.cs.mtsu.edu/~rbutler/adlb Accessed March 2012.

[2] Alexandrescu, A., "Three Cool Things About D – The Case for the D Programming Language," Google Tech Talk, July 29, 2010, http://www.youtube.com/watch?v=RlVpPstLPEc Accessed March 2012.

[3] ASCR SciDAC Universal Nuclear Energy Density Functional project: A Closer Look at Nuclei, Building a Universal Nuclear Energy Density Functional. http://www.scidac.gov/physics/unedf.html Accessed March 2012.

[4] Butler, R., Butler, T., Foster, I. Karonis, N., Olson, R., Overbeek, R., Pfluger, N., Price, M., and Tuecke, S., "Aligning Genetic Sequences," Chapter 11 in Strand: New

Concepts in Parallel Programming, Foster and Taylor, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[5] Butler, R., and Pettey, C., "A Bilingual Theorem Prover for Evaluating HPC Systems," Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, June 2007, pp 1000 – 1003.

[6] Butler, R., Ells, D., and Pettey, C., "PySMO: Python Shared Memory Objects," Proceedings of the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, July 2010, pp 203 – 209.

[7] Butler, R., Pettey, C., and Manifold, B., "Go2ADLB: An Interface for Using ADLB Within Go," Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV, July 2011, pp 54 – 58.

[8] D Programming language http://dlang.org/index.html Accessed March 2012.

[9] Gropp, W., Lusk, E., Ashton, D., Balaji, P., Buntinas, D., Butler, R., Chan, A., Krishna, J., Mercier, G., Ross, R., Thakur, R., and Toonen, B., MPICH2 Installer's Guide, September 14, 2007, ftp://ftp.mcs.anl.gov/pub/mpi/mpich2-doc-install.pdf, Accessed March 2012.

[10] Lusk, Ewing L., Pieper, Steven C., Butler, Ralph M., “More Scalability, Less Pain,” SciDAC Review 2010 http://www.scidacreview.org/1002/html/adlb.html Accessed March 2012.

[11] MPICH2, www.mcs.anl.gov/research/projects/mpich2/ Accessed March 2012.

[12] Pettey, C. adn Leuze, M., "A Theoretical Investigation of a Parallel Genetic Algorithm," Proceedings of the Third International Conference on Genetic Algorithms, 1989.

[13] Pettey, C., An Analysis of a Parallel Genetic Algorithm, PhD dissertation, Vanderbilt University, May, 1990.

[14] Tiobe Programming Community Index for February 2012 http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html Accessed March 2012.