J Supercomput (2006) 36: 123–134
DOI 10.1007/s11227-006-7953-6

Rapid prototyping frameworks for developing scientific applications: A case study

**Christopher D. Rickett · Sung-Eun Choi ·
Craig E. Rasmussen · Matthew J. Sottile**

© Springer Science + Business Media, LLC 2006

Abstract In this paper, we describe a Python-based framework for the rapid prototyping of scientific applications. A case study was performed using a problem specification developed for Marmot, a project at the Los Alamos National Laboratory aimed at re-factoring standard physics codes into reusable and extensible components. Components were written in Python, ZPL, Fortran, and C++ following the Marmot component design. We evaluate our solution both qualitatively and quantitatively by comparing it to a single-language version written in C.

Keywords Components · CCA · Python

1. Introduction and motivation

The development of scientific software tends to be different from the development of commercial applications. Experimentation, through the application of the scientific method is central to scientific effort, and software tools and practices should support this work pattern. Many software engineering techniques have been proposed to facilitate this work pattern, ranging from design methodologies (e.g., object-orientation, components) to self-contained software environments (e.g., problem solving environments, interpreted languages). In this paper, we perform a case study on one technique that is gaining momentum in the scientific

Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36, LA-UR-04-4655.

C. D. Rickett (✉)
South Dakota School of Mines & Technology, Rapid City, SD 57701
e-mail: Christopher.Rickett@gold.sdsmt.edu

C. D. Rickett · S.-E. Choi · C. E. Rasmussen · M. J. Sottile
Los Alamos National Laboratory Los Alamos, NM 87545
e-mail: {crickett, sungeun, crasmussen, matt}@lanl.gov

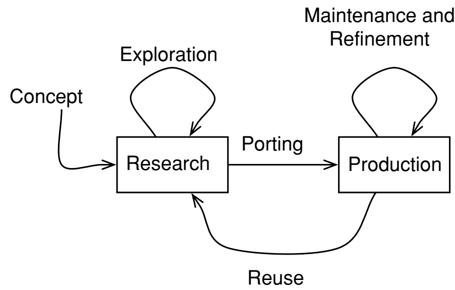

Fig. 1 The software life cycle of scientific applications

community. The Marmot test problem proposes a component-based software design for solving a basic physics problem. We demonstrate the application and examine the experience of implementing this design in a Python-based software framework. Our goal is to demonstrate the viability of this design pattern and software framework in terms of usability, performance and flexibility.

Figure 1 shows the typical life cycle of scientific software. The life cycle begins with a physical concept that is to be tested. At this point, the research phase is entered, wherein the scientific concept is modeled by a software prototype and the results of the model are analyzed. Should refinement of the model be needed (the typical case), the software is modified, the model is rerun and the output of the model is re-evaluated, as denoted by the arc labeled Exploration. Once the scientist is satisfied with the results, the research phase ends with the publication of the findings in a scientific journal.

Sometimes, the results of the software model are so important that a production phase is warranted. For example, atmospheric models [6, 13] are run every day to predict weather patterns affecting virtually all life on our planet. Software prototypes developed during the research phase are modified and improved as they enter production. Because these models are run frequently and the results are so important, effort is spent to improve the efficiency of the code and to verify and validate the output. Once put into production, the code is largely static except for maintenance and minor improvements denoted by the arc labeled Maintenance.

What may not be well understood, is that the research phase of the life cycle of scientific software never really ends. New hypotheses are formed and new software prototypes are created (or old ones modified), as the exploration cycle is repeatedly followed. Sometimes the results of new research are important enough that the production code is modified to reflect these results.

What is particularly difficult about the scientific software life cycle are the two, very different, modes of operation. The research mode favors rapid prototyping environments such as Mathematica [22] or Matlab [20], while the production mode favors the type safety and performance of compiled languages such as Fortran or C++. It is difficult to share software between the two environments.

Component-based software design has been proposed to help alleviate this problem. For example, a large software project sponsored by the National Aeronautics and Space Administration (NASA), the Earth System Modeling Framework (ESMF) [5], was begun in order to increase the ease of use, interoperability, and reuse of software used in numerical weather prediction, data assimilation and other Earth science applications. The Marmot project [12] was started for similar reasons, i.e., to examine the feasibility of using component-based software development for applications important to the Department of Energy (DOE). The

philosophy is that components should provide well-defined and stable (in time) interfaces, so that they can be developed, extended, and reused by a large developer and user community.

While projects, like Marmot and ESMF, will very likely aid the development of scientific software, we suggest that they don't completely address the two different modes of operation shown in Fig. 1. ESMF is largely a framework for Fortran components (layered on a C++ infrastructure) and Marmot is a C++ framework based on templates. Both frameworks support the production mode (with a high value placed on performance) better than they do the research mode (with a high value placed on exploration and rapid prototyping).

The Common Component Architecture (CCA) forum has proposed additional requirements on components beyond well-defined interfaces [1]. Software components should be language neutral (programmable in various languages) and load dynamically at runtime to support a rapid prototyping mode of operation (although they may also be linked statically for production runs). In this paper, we describe a case study to consider if a rapid-prototyping framework can be developed that supports the sharing of components between the research and production modes shown in Figure 1. Specifically, we use a Python-CCA component framework to solve a test problem proposed by the Marmot project. In this study, the Marmot component design [11] is carefully followed, although components have been created using a variety of languages (not just C++) by using the Python extension model [21]. We attempt to adhere to the Marmot C++ interface design to the point that any Marmot C++ physics component can be run within the Python framework. We evaluate our solution both qualitatively and quantitatively by comparing to a single-language solution using C.

The goal of this paper is to examine the feasibility of using a rapid prototyping environment to develop large, complex scientific applications based on components that can be shared between research and production environments. There are three general issues we seek to address:

- Are components and standardized interfaces helpful? There is a nontrivial cost associated with producing well-designed interfaces in order to produce reusable components. Is this cost worth the results?
- A rapid prototyping environment may be great for prototyping, but is it helpful in producing code that is production oriented? Can components be easily shared between the research and production environments?
- Is a mixed language environment beneficial? Can data structures be efficiently shared between languages?

It should be noted that components, rapid prototyping, and mixed language programming are not new topics. There exist excellent implementations of these ideas in other scientific frameworks (e.g., [15, 18]) However, we suggest that there is not yet a consensus among scientific application developers that these ideas are worth the bother and that they are not just a fad that will wither with time. Our goal is to evaluate the usefulness of these ideas with this case study.

The remainder of the paper is organized as follows. In Section 2, we give a brief overview of the Marmot project. In Section 3, we give implementation details of the rapid prototyping environment, as well as the Marmot test problem that was selected to be studied. In Section 4, we present a qualitative and quantitative evaluation of our Python-CCA solution as compared to a single-language solution. Finally, in Section 5 we make conclusions and suggest future work.

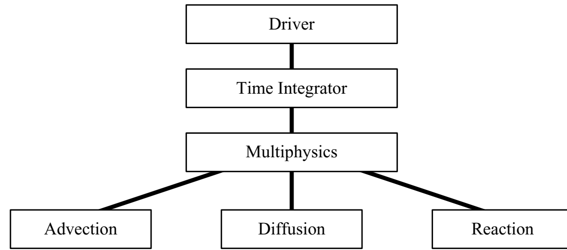

Fig. 2 Relationship of the Marmot components

2. Marmot project

The Marmot project is a computational physics effort at Los Alamos National Laboratory (LANL) to build extensible, high-performance multi-physics simulation codes based on modern component programming techniques [11, 12]. The Marmot project has carefully defined a set of interacting components with well-defined boundaries and C++ interfaces. We have adapted the Marmot C++ design to work within a Python CCA framework.

As a first step toward demonstrating the feasibility and performance of component-based programming techniques for simulations of this type, the Marmot Advection-Diffusion-Reaction (ADR) test problem was posed. While this test application is small, it represents the kernel of more general and realistic ADR applications. The additional complexities associated with more complex boundary conditions, diffusion coefficients, or reactions rates (for example), of a more realistic application, are not felt to materially affect the conclusions of this study.

The scope of the Marmot project extends far beyond the simple test problem implemented in this study. The project's aim for the first year is to produce interfaces and prototype component implementations, that can be further extended into production components suitable for reuse across projects. However, component reuse is beyond the scope of this paper, which is more concerned with development environments for scientific applications.

This section presents a summary of the test problem. Details regarding parameters, boundary conditions, and spatial and time discretization can be found by referring to the original papers [10, 11].

The goal of this test problem is to write a component-based application that produces a numerical approximation of the ADR equation given by:

$$\partial_t u + \nabla \cdot \vec{f}(\vec{x}, u) - \nabla \cdot (d(\vec{x}, u)\nabla u) + r(\vec{x}, u)u = q(\vec{x}, u) \tag{1}$$

where $u(\vec{x}, t)$ is a scalar unknown, $\vec{f}(\vec{x}, u)$ is an advective flux, $d(\vec{x}, u)$ is a diffusion coefficient, $r(\vec{x}, u)u$ is a reactive term and $q(\vec{x}, u)$ is a source term. Six major components are involved in the solution of the ADR problem. The relationship of these components is shown in Fig. 2. The Driver component replaces the "main" of a traditional application and is responsible for reading initial data, running the time loop to advance simulation time, and saving state. The Driver component uses a time Integrator component to advance one time step. The time Integrator (with the support of the Multiphysics component) implements the particular time integration method used. An operator-split time integration method allows each component of the ADR Eq. (1) to be written independently, reflected directly in the component decomposition of the software itself. These three physics components are shown at the bottom of the figure and are described separately below. It is assumed that a 2-D solution space is discretized into $M \times N$ cells.

Advection. This component uses an upwind Godunov scheme [8] to approximate the solution to the advection term,

$$\frac{u^{n+\frac{1}{3}} - u^n}{\Delta t} + \nabla \cdot \vec{f}^n = 0. \tag{2}$$

Traversal of the domain once along each axis is used to apply the Godunov scheme to the 2-D domain.

Diffusion. The Diffusion component solves the diffusion term,

$$\frac{u^{n+\frac{2}{3}} - u^{n+\frac{1}{3}}}{\Delta t} - \nabla \cdot \left(d^{n+\frac{1}{3}} \nabla u^{n+\frac{2}{3}}\right) = 0, \tag{3}$$

implicitly on a regular, face-centered grid. This reduces to a linear system of the form,

$$A\vec{u} = \vec{q} \tag{4}$$

where A is a symmetric positive definite $M \times N$ by $M \times N$ matrix. Due to the five point stencil of the finite difference approximation of Eq. (3) the matrix A is sparse with a banded structure containing at most five non-zero elements per row. Two steps are required to implement this component. First, the A matrix and $\vec{q}$ vector must be assembled based on the current cell-centered grid, and then the unknown vector $\vec{u}$ must be solved by invoking a basic linear solver.

Reaction. The update for the reaction equation can be written as:

$$\frac{u^{n-1} - u^{n+\frac{2}{3}}}{\Delta t} + r^{n+\frac{2}{3}} u^{n+1} = q^{n+1}. \tag{5}$$

This portion of the system is fairly simple to implement, as at any grid point, the solution is independent of the solution at neighboring cells.

3. Implementation details

In this section, we give a description of the implementation of the components used in the Marmot ADR test problem as well as a description of the monolithic C application used for comparison.

3.1. Components

A CCA framework [1] was used to dynamically create components at runtime and to compose them into a running application. The implementation of a small CCA software layer allows a component to describe itself via "uses" and "provides" ports (a collection of interfaces) at runtime. For example, the Multiphysics component "uses" a physics port, "provided" by the Advection component. We used a pure Python implementation of the CCA framework.

All of the components in Fig. 2 were first written in Python. A large number of co-operating tools and extension modules exist for Python specifically for scientific users. These include modules for graphics and plotting, optimization, integration, special functions, signal and image processing, genetic algorithms, ODE solvers and more [17]. In our

implementation, we used the Python Numeric and PySparse extension modules [7, 17], which include support for high-performance arrays and linear solvers. Each component implements a **setServices** method (four to five lines of code) to allow it to run in a CCA framework.

The physics components were then ported from Python to a compiled language: ZPL for Advection, Fortran 95 for Diffusion, and C++ for Reaction. The use of different languages allowed us to anticipate the environment where components are developed by separate individuals or teams, each with an expertise in a different language.

Chasm [16] was used to provide multi-language interfaces to the array data structures shared between components. Fortran 95 uses a rich array descriptor for representing arrays, as opposed to the simpler pointer and length scheme used by languages such as C and C++. The Chasm array-descriptor library allows array data to be shared between languages (Fortran, ZPL, Python, and C++) without the unnecessary copying of data. Chasm was also used to facilitate the creation of component bridging code, called by Python to interface to the various implementation languages used. Other language interoperability tools could have been used equally as well, for example, SWIG [2] or Babel [9].

Advection component. The Advection component was written in ZPL [3, 19]. ZPL is an array programming language designed from first principles for fast execution on both sequential and parallel computers. ZPL was chosen because of its simple and natural expression of the stencil operations that dominate the Advection phase. The underlying implementation of ZPL represents arrays similar to Fortran by using array descriptors and is used to allocate the $M \times N$ array shared by each of the components. By default, the data is distributed in a block manner by the runtime libraries, where the processors involved are organized into a grid and blocks of data are assigned to each processor. Any data that lies on a block border within the extent of the stencil (i.e., within one location for the Advection) is replicated per processor.

The initialization phase for the Advection component includes initializing the ZPL runtime system with information such as the distribution and number of processors. The same ZPL code that runs on a sequential machine runs on traditional supercomputers (such as the Cray X1) and large-scale clusters with efficiency comparable to hand-coded message passing. ZPL code is compiled to low-level C and the most appropriate communication mechanism (e.g., MPI, SHMEM, etc.). This is the only component that is already parallelized.

All of the array data structures are allocated by ZPL and shared with other languages. A Fortran component, for example, has access to the information regarding how an array is distributed across processors and the Fortran component can be parallelized using the chosen communication mechanism. We believe this is the first time ZPL has been used in a multi-language environment in this fashion.

Diffusion component. The core of the Diffusion component solves a system of linear equations. For this component, the SuperLU sparse solver library [4] (version 3.0) was used. The SuperLU routines perform an LU decomposition with partial pivoting and triangular system solves through forward and back substitution. SuperLU is written in C.

Reaction component. The Reaction component was implemented in C++. The C++ implementation reduced the complexity of indexing through the use of iterators and conversion routines to convert between cell-center indices and corresponding cell-face indices. While this notation is verbose and explicitly loops over cell indices, there was a net overall reduction in complexity.

3.2. Single language version

When the Python prototype components were complete, the components were translated to a more traditional application written in C. The C version maintained the component decomposition, though it was not written to be run within the Marmot or CCA component frameworks. It used the same solver library as the Fortran Diffusion component. The distinguishing feature of the C implementation was its array representation. Since C does not have adequate support for multidimensional arrays, the C version implements two-dimensional arrays as a dynamically allocated array of pointers to a data array. While this results in an $O(M)$ storage overhead, it greatly simplifies indexing over the typical scheme of flattening multiple dimensional arrays.

4. Evaluation of the framework

In this section, we present a quantitative and qualitative evaluation of our framework, including a response to the questions posed in Section 1.

4.1. Quantitative evaluation

Table 1 shows a comparison of the pure Python prototype, the multi-language, and the C versions in terms of *source lines of code* (SLOC) minus comments and blank lines. Note that the pure Python Driver, Integrator, and Multiphysics components are also used in the multi-language runs. SLOC is often criticized for being a poor metric of comparison, but we cite it here because it gives some sense of program complexity and maintainability. The multi-language code (excluding the *Other* category) was about 4.6 times larger than the pure Python version, while the C version was only 2.2 times larger. The C version was larger as it required explicit looping for every component except Reaction, where the flattening of the array also required extra lines of code. The *Other* category counts source files used for support code such as array support for the C version and the Python component wrappers. The bulk of the code for the multi-language version in the *Other* category (approximately 800 lines) consisted of automatically-generated code for the ADR components, and thus does not necessarily reflect direct programmer effort.

All experiments were run on a 2.6 GHz Intel Xeon computer running Linux kernel version 2.4.X with 2.0 GB of memory. Table 2 shows the various compilers, libraries, and flags used for building the components.

Table 1 Source lines of code for the various versions

Component	Pure Python	Multi-language	C
Driver	62	62	97
Integrator	38	38	158
Multiphysics	73	73	—
Advection	90	128	156
Diffusion	99	161	197
Reaction	26	92	28
Sub-total	388	509	636
Other	22	1497	269
Total	410	1905	905

Table 2 Tool versions used for the ADR components

Tool	Flags	Component
Python 2.2	—	Driver, Integrator, Multiphysics
ZPL compiler 2.0.0 (compiled to C)	−O1	Advection
Intel fortran 95 compiler 7.0		Diffusion
SuperLU 3.0	—	Diffusion
Intel C/C++ compiler 8.0	-O3 -tpp7 -axKW -restrict -std = c99 -ip -mcpu = pentium4 −march = pentium4	Advection, Diffusion, Reaction

Table 3 Execution times for ADR code

Version	Size	Loop	Sum	Advection	Diffusion	Reaction
Multi-language	100 × 100	2.371 sec	2.37 sec	0.00346 sec	2.365 sec	0.000648 sec
C	sparse	2.3444 sec	2.34 sec	0.00106 sec	2.341 sec	0.000699 sec
Multi-language	200 × 200	34.635 sec	34.63 sec	0.0144 sec	34.612 sec	0.00273 sec
C	sparse	34.97 sec	34.96 sec	0.00843 sec	34.95 sec	0.00293 sec
Multi-language	300 × 300	165.46 sec	165.44 sec	0.0324 sec	165.4 sec	0.006 sec
C	sparse	165.48 sec	165.23 sec	0.0133 sec	165.21 sec	0.00636 sec

Table 3 shows the execution times for the computational portion of the Marmot ADR test problem. Loop time is the time spent in the time loop (advancing computational time), while sum is the total time spent in the three ADR physics components (neglecting the overhead of the Python Driver, Integrator, and Multiphysics components).

The Diffusion and Reaction components performed similarly but the ZPL advection component was 2–3 times slower than the C version. This is expected because the ZPL generated code is already parallelized and thus contains index translations, checks for communication, and other significant details required for a parallel version.

Perhaps, the most important result is that there is a small difference between the loop and the sum timings for the multi-language runs which is comparable to that of the C version. This indicates that the overhead of the Python CCA framework (as compared to doing the same thing in C) is negligible; there was simply no performance cost associated with using the Python prototyping environment to run the Marmot ADR components.

Note that we did not perform a study implementing every component in every possible language, as raw performance is not the ultimate metric for success in this work. The choice of languages was motivated by seeking good performance for the most appropriate languages for expressing the given computational task. Quite often high performance, particularly with languages such as C and Fortran, comes with algorithmic and conceptual obfuscation that ultimately degrades the productivity of the researcher.

4.2. Qualitative evaluation

Is a mixed language environment beneficial? Can data structures be efficiently shared between languages?

We found a mixed language environment to be beneficial for several different reasons. The first of these is programmer choice and proficiency. Among the authors, there was a range

of proficiencies in the chosen implementation languages. Some preferred C, others Python, two had experience in ZPL, and one extensive experience with Fortran. In this study, each author worked in the language or languages they were proficient at, making the programmer more efficient and less error prone.

There are large differences in the way loops are expressed in the five implementation languages. ZPL is the most concise with loops expressed as array-valued expressions applied over ZPL regions. Fortran and Python have an array-slice notation and C++ has iterators. Regions, slices, and iterators were all used effectively to reduce the complexity of explicit loop indexing. We found errors associated with looping and array indexing to be the most common error made in implementing the components.

However, there was no clear cut choice for the easiest language to program in. ZPL is a good choice for components that access dense arrays in a regular fashion. Parallelization is another factor to consider. Though not taken advantage of here, ZPL components are implicitly parallel, thus reducing the time needed to develop parallel components.

As noted above, there were clear performance differences in the component implementations. However, there was no single language that performed best. C performed best for the Advection component, while Fortran was best for the Diffusion component.

There was a cost associated with the multiple language environment. Python wrappers had to be created for each multi-language physics component. While this was partially automated, it was tricky to learn and slowed down development for the programmer unfamiliar with Python extension modules. In general, an expert should be able to create a Python wrapper for a given component interface in a few hours or less, given existing tools [2, 9, 14, 16]. This is a one-time cost that can be amortized across many components sharing the same interface. As the ADR physics components all have a single, well defined interface, the Python wrapper need only be created once for each implementation language.

Data were easily shared between implementation languages for the initial Marmot ADR test problem. This was due to the ability to use arrays as the primary data structure. It is planned for the Marmot project to experiment with more complicated data structures on unstructured meshes in the future. It is unlikely that ZPL would be a language of choice under these circumstances. The ability to use Python and Fortran components for more complicated data structures in the future will ultimately depend on their C++ representations.

A rapid prototyping environment may be great for prototyping, but is it helpful in producing code that is production oriented? Can components be easily shared between the research and production environments?

It is generally accepted that prototyping environments such as Matlab and Mathematica are useful during the initial research phase of development, where models are being developed and tweaked. Transitioning these “worksheets” to real code has been less successful. In fact, there have been many efforts (including companies) to automatically translate such prototype codes to Fortran or other compiled versions, but most have met with limited success. Based on experience with such environments, we believe this is due to the fact that they are too “free-form.” In our opinion, Python (and the Python Numeric library in particular [17]) provides a nice balance between free-form experimentation and a real program. The Numeric module provides many high-level data structures and functions that resemble the mathematics; this enables easy translation of formulas to code, a desire expressed by many scientists. In addition, Python allows the programmer to consider real issues like program control flow, file I/O, and performance.

Porting the Python prototype components to compiled languages was fairly trivial—errors encountered were typically minor and in most cases they were found by simply looking at the original Python code (no debugging was necessary). The major difficulty lay in the use of Python's high-level data structures which meant porting their functionality to the compiled languages. For example, Python has several ways in which to access array slices and whole arrays and keeping the semantics of these straight was an annoyance at times, even for those familiar with Python.

The importing of production components (represented by our ZPL, Fortran and C++ components) to the rapid prototyping environment, which was performed by a novice to Python, was more difficult than expected. The difficulty was due to version sensitivity of the Python interpreter and our inexperience with shared libraries in an interpreted environment. Python and similar interpreted languages are notoriously poor with respect to version sensitivity, and thus a large amount of time was spent moving between versions. We found the loading of dynamic libraries much more difficult in an interpreted environment. In a compiled environment, the linker can fully identify dependences so that the proper libraries can be loaded at runtime. In the Python environment, there is no final link step and thus we had to manually link in many obscure libraries which we were unaware of.

Note again that this importing was performed by someone who was not familiar with the Python environment, thus our recommendation would be for an expert to perform the integration of production components.

Are components and standardized interfaces helpful? There is a nontrivial cost associated with producing well-designed interfaces in order to produce reusable components. Is this cost worth the results?

The factoring of the Marmot ADR test problem into components was likely the most important factor making a shared prototyping and production environment possible. The design provided each software module with a clear responsibility and a specific interface through which it interacted with other components. Given a component interface, a Python wrapper of the component can be created. If component interfaces are changed infrequently, then the cost of creating the Python (and other language) wrappers can be amortized over a long period of usage.

While several man-months went into the creation of the Marmot design, only a few man-weeks were needed to implement a prototype. The existence of the design shortened component development time. The separation into components allowed separate team members to implement and test a particular component, largely in isolation from other ongoing code development.

5. Conclusions and future work

In this paper, we performed a case study of a rapid prototyping environment for scientific applications using the Marmot ADR test problem. The scientists involved in the Marmot project, spent several man-months factoring the problem and defining the set of components and their interfaces. This component factoring was essential for our rapid prototyping environment, as we were able to quickly implement a reference implementation in Python, which was then used to implement the ADR components in the most natural language for the particular task. For comparison purposes, the Python version was also used to implement

a single-language version in C. (It would have been convenient to use the Marmot C++ implementation for comparison, but it has not been completed yet.)

We found the existing Marmot component specification to be the most important factor in creating a prototyping environment that shared components with the production environment. We also found that Python provides an excellent prototyping environment via extension modules that provide syntax and functions that closely map to the actual mathematics being performed. Using the prototype as a reference implementation proved very useful, though porting the high-level syntax to a lower-level language required care. Sharing array data between components implemented in different languages was straight forward and efficient using the Chasm array-descriptor library.

Components were implemented in several languages (Python, ZPL, Fortran, and C++). No language displayed a clear cut advantage, either in terms of ease of implementation or in terms of performance. The performance overhead of using the Python prototyping environment was minor, since this test problem and other like problems are computationally intensive.

Future work. We have only implemented (or wrapped) a portion of the Marmot C++ framework in Python in order to test the concept of prototyping Marmot physics components. More work needs to be done in order to fully share components between the two environments. In addition, the sharing between components of the more complicated data structures associated with unstructured meshes needs to be studied. We expect this will present interesting new challenges.

References

1. Armstrong R, Gannon D, Geist A, Keahey K, Kohn S, McInnes LC, Parker S, Smolinski B (1999) Toward a common component architecture for high-performance scientific computing. In Proceedings of High Performance Distributed Computing, pp 115–124
2. Beazley DM (2003) Automated scientific software scripting with SWIG. In Future Generation Computer Systems, vol 19. Elsevier,
3. Chamberlain BL (2001) The design and implementation of a region-based parallel language. PhD thesis, University of Washington
4. Demmel JW, Gilbert JR, Li XS (2003) SuperLU users' guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory
5. Dickenson RE, Zebiak SE, Anderson JL, Blackmon ML, DeLuca C, Hogan TF, Iredell M, Ji M, Rood R, Suarez MJ, Taylor KE (2002) How can we advance our weather and climate models as a community? In Bulletin of the American Meteorological Society, vol 83
6. European Centre for Medium-Range Weather Forecasts. see www.ecmwf.int
7. Geus R, Arbenz P (2003) A Python framework for large scale sparse linear algebra. In PyCon DC 2003
8. Godunov SK (1959) A finite difference method for the numerical computation of discontinuous solutions of the equations of fluid dynamics. Matematichesky Sbornik 47, pp 271–290
9. Kohn S, Kumfert G, Painter J, Ribbens C (2001) Divorcing language dependencies from a scientific software library. In Proceedings of the 10th SIAM Conference on Parallel Processing
10. Lowrie R, Evans T, Dilts G, Turner J, Ferenbaugh C, Dahl J, Urbatsch T (2003) Numerical methods for the advection-diffusion-reaction (ADR) project. Technical Report LA-UR-04-4031, Los Alamos National Laboratory
11. Lowrie RB, Evans TM, Dilts G, Ferenbaugh C, Urbatsch T, Dahl J, Turner J, Wingate C, Clark B, Hubbard M (2004) Code design for the advection-diffusion-reaction (ADR) project. Technical Report LA-UR-04-4032, Los Alamos National Laboratory
12. Lowrie RB, Evans TM, Dilts GA (2003) Vision and scope for the advection-diffusion-reaction (ADR) project. Technical Report LA-UR-04-4035, Los Alamos National Laboratory
13. National Oceanic and Atmospheric Administration. see www.noaa.gov
14. Peterson P. F2PY: Fortran to Python interface generator. see cens.ioc.ee/projects/f2py2e

15. Pyre: A Python Framework. see www.cacr.caltech.edu/projects/pyre
16. Rasmussen C, Lindlan K, Mohr B, Striegnitz J (2001) Chasm: Static analysis and automatic code generation for improved Fortran 90 and C++ interoperability. In Proceedings of the Los Alamos Computer Science Symposium
17. Scientific tools for Python. see www.scipy.org
18. SCIRun: A Scientific Computing Problem Solving Environment. Scientific Computing and Imaging Institute (SCI), software.sci.utah.edu/scirun.html, 2002
19. Snyder L (1999) Programming guide to ZPL. MIT Press, Cambridge, MA, USA
20. The MathWorks. see www.mathworks.com
21. The Python Language Home Page. see www.python.org
22. Wolfram Research, Inc. see www.wolfram.com