Scientific Software Frameworks and Grid Computing

Improving Programming Productivity

Bill Appelbe[1], Louis Moresi[2], Steve Quenette[1], and Patrick Sunter[1]
1 Victorian Partnership for Advanced Computing (VPAC)
PO Box 201, Carlton South, Victoria 3188 Australia
{bill, steve, pat}@vpac.org
2 School of Mathematical Sciences
Monash University, Clayton, Victoria 3800, Australia
louis.moresi@sci.monash.edu.au

Abstract Scientific research applications, or codes, are notoriously difficult to develop, use, and maintain. This is often because scientific software is written from scratch in traditional programming languages such as C and Fortran, by scientists rather than expert programmers. By contrast, modern commercial applications software is generally written using toolkits and software frameworks that allow new applications to be rapidly assembled from existing component libraries. In recent years, scientific software frameworks have started to appear, both for grid-enabling existing applications and for developing applications from scratch. This paper compares and contrasts existing scientific frameworks and extrapolates existing trends.

Introduction

A *software framework* is *an organized collection of reusable software components that is used to implement software applications*. Software frameworks are now in widespread use in commercial software development, and software development using such frameworks is referred to as Component Based Software Development or Engineering [1]. The two most widely used commercial software frameworks are SUN's Java J2EE, and Microsoft's .NET. The J2EE and .NET frameworks consist of many thousands of classes that provide the basic infrastructure necessary to implement GUIs, web sites and web services, and relational database interfaces. Frameworks provide significant software productivity improvement through reuse of their components [2].

The notion of *grid computing* [3] has several interpretations, ranging from a rather restricted interpretation (an application run remotely), to a much more expansive view of an application that is in use by a distributed community that uses the grid to share and exchange models (developed using the application), data, and research outcomes. In this paper, we adopt the more expansive view of a grid-based application, in line with the evolution and growth of the grid and communities that use it.

So the key aim of software frameworks, as applied to scientific grid-based applications, is to reduce the development, maintenance, and support time for scientific applications and models, comparable to that achievable for commercial software development by reuse. Realistically, widespread large productivity increases for scientific software are unlikely for a variety of reasons, not the least of which is the limited "market" for specialized scientific applications and the corresponding lack of resources. However, it is arguable that the progress in scientific software frameworks has been significant over the past few years, with both successes and failures, which point to emerging opportunities and likely evolution.

The definition of software frameworks adopted above (or equivalent definitions such as in www.wikipedia.org) is inherently subjective and qualitative – it provides no clear measure of what is, and is not, a framework. However, the definition of frameworks adopted above does leads to several key characteristics of software frameworks that can be used to assess, or quantify, the extent to which a software product, library, or *toolkit,* is a framework:

1. *Extensibility* – to what extent can the framework be extended or adapted, by mechanisms that include specialization, introduction of new components, and modification of existing components? Inevitably, extensibility implies object-orientation – in the architecture and implementation of the framework.
2. *Integration* – to what extent can the components of the framework be combined or interoperate? Is there an underlying architecture that facilitates assembly of components?
3. *Scope* – what fraction of an application can be implemented by applying framework components? This fraction is obviously dependent upon the application and the extent to which customization or adaptation is needed. A broad framework scope provides for significantly reduced development time, but comes at a corresponding cost and complexity of the framework.

Frameworks for Grid Computing

One view of grid computing is that it will eliminate or reduce the need for traditional "command line" scientific computing – where a user logs in to a remote or local computer, typically running Linux, then runs the scientific application as a batch job, supplying it with whatever files and command arguments are required. Such command-line computing has been the standard model for using scientific software since the 1960's. Command-line computing often requires considerable application expertise to configure and run, as every application has its own input and

output formats, files, and constraints. In addition, there are typically suites of applications or programs that need to be run to create model and post-process the output (e.g., to create a visualization or plots). When used repetitively, such a suite of applications and programs is referred to as a *workflow*[4]

Grid computing can simplify scientific computing by replacing the command line model by a *web portal*, that uses web pages to provide a standardize www interface for scientific data and computations – setting up, running, archiving, and post-processing output of scientific applications running on remote computing systems (e.g., blast searches on NCBI's website [5]). The goal of web portals is to provide a scientific computing www environment that is as convenient and simple as that for www applications such as online news, encyclopedias, or travel.

For well-developed and standardized scientific applications, such as blast, web portals can be highly successful. However, it is often very labor intensive to create and maintain effective customized web portals for less widely used scientific applications (in spite of the availability of robust toolkits such as gridsphere for creating web portals). Thus there are two key limitations of the strategy of replacing command line scientific computing with web portals:

1. Individual scientists and applications often have their own nuances and techniques for using scientific applications, based on their research objectives and peculiarities of the application – a "one size fits all" approach for scientific computing is not feasible. The strength of command-line computing is its flexibility, albeit at a cost in complexity. Conversely, the weakness of portal-based computing is that it is limited by whatever workflows and input are built into the portal to support the user community.
2. The other limitation of portals is that they do not provide for customization of the scientific application – beyond what customization was provided for by the implementer of the scientific application. Almost invariably, the implementer of the portal is not the same as the implementer of the application (as portals and scientific applications use different skills and programming technology). In effect, a portal "shrink wraps" an application, but does not readily provide for customization of the scientific application inside of the shrink-wrapping (such as changing a solver or a constitutive law).

A partial solution to the lack of flexibility in portals, and the lack of standardization of inputs to scientific applications is *superstructure frameworks*. A superstructure framework is simply a framework that provides components to replace the "top-level" of an application, and/or tie together applications or application components into a workflow. Notable examples of such superstructure frameworks include:

- Kepler [4] – for workflows
- Pyre [6] – a Python toolkit for staging, monitoring, and visualization scientific applications.

While superstructure frameworks do provide more capability for customization of applications and their interfaces than web portals, they still do not facilitate the internal customization of an application. Superstructure frameworks generally use interpreted languages and tools that are unsuited to customization of low-level high-

performance numerical algorithms used by scientific applications. A further limitation of superstructure frameworks is that they require the end-user to become familiar another programming paradigms – that of the superstructure framework. By contrast, web portals generally do not require any end-user knowledge of the portal technology/framework. Of course, specialist applications support developers can provide the expertise needed for end-users of superstructure frameworks, but that just shifts, rather than removes, the complexity introduced by superstructure frameworks.

Superstructure frameworks do have an important niche in interfacing scientific applications and standardizing interfaces to scientific applications. However, they do not significantly simplify the development and support of the algorithms and data structures of parallel scientific applications. For such a task, it is necessary to develop frameworks that reduce the development and maintenance cost for applications themselves. Such frameworks are referred to either as *infrastructure* or *scientific frameworks*.

The Evolution of Scientific Applications

Scientific applications software is always faced by the challenge of the moving target of scientific research and models. Once a problem is "solved" it is no longer research, and the frontier of research is always moving towards ever more complex problems and models. This is the "catch-22" of scientific software for research – software application and models often need to evolve to be relevant to the research community. In disciplines with large research communities and well-understood physics, such as quantum chemistry and atmospheric physics, the community may be supported by commercial or *community codes* – well-established scientific applications that are in widespread use and supported by a funded development and support team (e.g., Gaussian or NAMD [7] for Chemistry). In such disciplines, the community need is for maintenance and extension of existing codes, rather than development of new codes. However in other research disciplines, such as geosciences, there are relatively few well-supported community or commercial codes in widespread use. In such disciplines, software development and modeling is dominated by a patchy landscape of specialized codes each with a small band of loyal followers. Reflecting on the difficult environment in which these codes originated and thrive, they are colloquially referred to as "*hero codes*" [8].

A hero code is simply a scientific application that was written to model a specific scientific research problem. Common characteristics of hero codes include:

- Design and coding is done by one person, often a scientist with relatively little formal training in programming or software design
- High priority is given to modeling a very specific problem, or family of problems, for a publication or graduation deadline
- Low priority is given to documentation, adaptability, or reuse of the software
- The user-community for a given code often coincides quite closely with a particular school of thought within a discipline

Hero codes correspond to what is known in commercial software as a *rapid prototype* – where speed of development is initially a primary goal. Rapid prototypes are intended to be "throw away" code [9] – used once, but then thrown away as the code was never built with maintainability and robustness. However, experience in commercial software and scientific computing is that such prototypes, or hero codes, are rarely thrown away. Instead what happens is that successful hero codes are copied and adapted, sometimes very painfully, to meet new requirements, such as:

- Changed scope of boundary and initial conditions beyond those originally foreseen in the initial code development.
- Different discretizations of the primary variables such as new meshes, elements, or interpolation functions of higher order or adaptive refinement.
- Changing the solution scheme, for performance or accuracy reasons
- Applying the code to include additional physics or to work on multi-scale, multi-physics (coupled) problems
- Scaling the code up to much larger problems – inevitably by parallelizing the code in one or more dimensions to be run in parallel on dozens or hundreds of processors

Inevitably, once the original successful hero code is made available to a community it is copied and adapted by individual researchers, with little coordination or consistency in adaptation. Multiple version of the hero code exist, and each new version just adds more complexity – there is often no incentive or funding mechanism for an individual researcher or research organization to perform a "cleanup" of the hero code and its versions. By contrast, in commercial software, the maintenance costs of software a budget line-item, so there is strong incentive to drop or replace legacy applications that are expensive to maintain, or incrementally improve legacy applications through *refactoring* [10] – preserving the same functionality, but making incremental improvements in the design to enable reuse and extension.

Scientific Software Frameworks

Math libraries have been widely used for numerical software development since the 1950's. The evolution of such libraries gradually extended from simple scalar functions (e.g., sine or exp), to higher-level libraries:

- BLAS (Basic Linear Algebra Subprograms, 1979) – matrix and vector operations
- LINPACK and LAPACK (1990) [11] – solvers, built on BLAS

These libraries gradually evolved to the point that optimized versions were available for most shared memory and vector processors by the 1990's. However, the rise of distributed memory systems complicated the implementation and use of such libraries. Such libraries generally did not meet the criteria of a framework (extensibility, integration, and scope), and were functional rather than being object-oriented. Several key evolutionary steps were needed before libraries could evolve

into frameworks that effectively support portable scientific applications for distributed memory systems:

a) A portable standard for message passing (MPI, 1992)
b) Decoupling the library implementations of matrices and vector (e.g., sparse, dense, and how these are distributed across processors) from the interface to these abstractions. This decoupling of interfaces from implementations is a powerful object-oriented mechanism for adaptability – the client sees only the interface, which can be dynamically replaced by any implementation of the interface.
c) Provision of partial differential equation (PDE) solver libraries – where much of the complexity of many applications lies, with the greatest opportunities for reuse and productivity gains.
d) Implicit parallelism – hiding the details of parallel decomposition and communication from users of the library (while allowing a choice of parallel decompositions through interfaces). The design and implementation of portable, scaleable, and efficient parallel numerical methods requires considerable specialist labor and expertise.

PETSc (1994) [12], the Portable Extensible Toolkit for Scientific computation was the first scientific library in wide use to incorporate these features. The main application domain for PETSc is scientific applications based around linear solvers, although it includes support for other domains such as non-linear solvers.

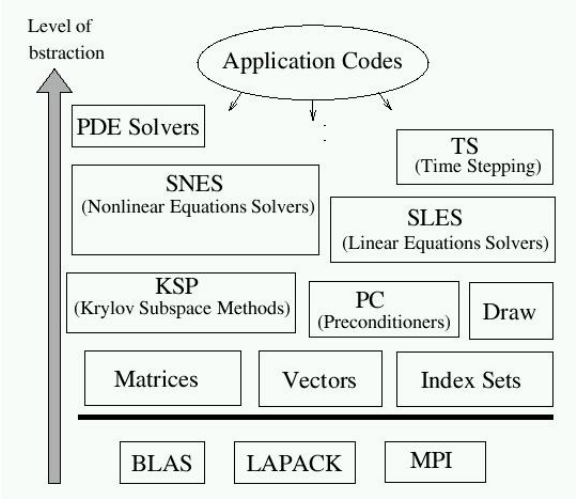

Fig. 1. The Architecture of PETSc

What distinguishes PETSc from earlier numerical libraries is its use of object-oriented abstractions, and an overall architecture. Even a solver is an object, or *context*, that is filled in at runtime with information about the solver. Within the domain of PDE solvers, PETSc is arguably the first scientific framework, as it satisfies the three key criteria of extensibility, interoperability, and scope (a large fraction of a many PDE based scientific applications can be implemented in PETSc).

PETSc has been widely used, deployed, and extended. Since PETSc, there have been several other frameworks for scientific computing developed, with somewhat different goals notably: Trilinos [13], CCA [8], and StGermain [14, 15].

Trilinos has a somewhat different philosophy and implementation style to PETSc. Firstly, it is implemented in C++, rather than C. Secondly, it is composed of contributed packages, as opposed to PETSc's structure of a single integrated subroutine library. The use of C++, which is an object-oriented "extension" of C scientific computing is increasing, and it is arguable that since almost any framework will be object-oriented, it is better to implement frameworks in an object-oriented programming language. The downside of using C++ is that C++ is a considerably more complex programming language than C, especially when using advanced features of C++ such as templates and the BOOST libraries. Such complexity may make it difficult for end-users, who are scientists rather than computer scientists, to use the framework. In addition, the package structure and "contributed component" structure of Trilinos may lead to inconsistencies and incompatibilities between components as Trilinos evolves. Arguably, a consistent modular architecture is central to the success and evolution of frameworks.

Both PETSc and Trilinos use object-oriented abstractions, and interfaces, between components, and abstractions for the foundation classes of numerical software (Matrices and Vectors – the Petra libraries in Trilinos and the implicit Mat and Vec abstractions in PETSc). This means that PETSc and Trilinos should be interoperable, if appropriate wrappers and adapters are built.

CCA, the Common Component Architecture, is an alternative approach to frameworks based on standardization of component interfaces, including a formal specification of the architecture. So CCA is not really a software framework itself, but rather a meta-framework. The success of CCA will depend upon the extent to which is adopted by the community and the perceived value of such a framework. In practice, such "top-down" standardization efforts have not been very successful in the past (e.g., HPF). Standardization works best in a "bottom-up", tightly integrated fashion where the standard covers a quite limited application domain (e.g., MPI).

StGermain

StGermain is a framework, like PETSc or Trilinos, but has key architectural differences:

- It is component-oriented (using classes, implemented in C, and a component-architecture)
- It is hierarchical

The traditional definition of object orientation is that all data is encapsulated in classes, and common behavior is factored out using inheritance and polymorphism (dynamic binding of methods or functions). But merely being object-oriented does not ensure that an application or framework can be readily extended or adapted. Thus modern software frameworks have extended object-orientation into component-orientation, to facilitate reuse of components using a variety of

programming techniques, design patterns [16], and architectural principles. These include:

1. Run or load-time class definition – new classes can be created "on the fly", and the data and methods of those classes specified at runtime
2. Reflection [17] – at runtime, an application can query the properties of any class or object
3. Factories [16] – once a class is defined, it is registered in a factory. An application can create objects dynamically of any registered class

Such component orientation is needed to support declarative programming [18] – a scientist specifies *what* the model should be, in a declarative language such as XML, rather than *how* the model should be implemented, in an imperative programming language such as C or Fortran. The objects and classes are then generated from the declarative language. StGermain supports this approach. The advantage is that at the top-level, a scientist who is not a programmer can specify a model using XML

A scientific application can be viewed at several different levels or layers, depending on the expertise and objective of the user at that layer

Model Layer - Scientist

- Models or solves real world problems, using applications frameworks developed by computational scientist

Application Layer - Computational Scientist

- Develops application framework from constitutive laws (equations) using solver frameworks

Solver Layer - Numerical Methods Developer

- Implements numerical methods and solver frameworks using generic software components

Component Layer - Software Engineer

- Implements generic software components (e.g., Object, Class, Entry Points, Mesh, Particle Swarms) and interfaces

Hierarchical software design and minimizing coupling between software components, have long been recognized as foundations of good software design (Myers, 1978 [19]). However, most scientific applications and other frameworks such as PETSc and Trilinos, do not tend be hierarchical: application I/O and model setup, physics, solvers, and component library calls are intermixed. This makes scientific programs very difficult to understand and maintain (as a user or maintainer needs to understand all the layers). By contrast, in StGermain the layers are all separate and hierarchical. Such separation also facilitates reuse. On top of the one component layer, many solver layers can be built, and on each solver layer many

application frameworks can be built (by solver, we mean any numerical or analytical method for solving a collection of PDEs, not just an implicit "Ax=B" solver). We use the term application framework, as a StGermain application is highly parameterized – the input to the application can control the solvers and constitutive laws used, as well as the initial and boundary conditions. The application layer input is an XML file (a Model).

The Component Layer of StGermain is *StgDomain* and it includes approximately 200 classes (implemented using C structs). The key components of the StgDomain are:

- Object and class support
- Local generic data structures: e.g., Vector, Set, Dictionary, and BTree
- Local numerical classes: e.g., Topology, Tensor, and ConvexHull
- Distributed numerical classes: e.g., Decomposition, Field, Mesh, and Swarm (particle collection)

The StgDomain layer contains no "solver" components, such as Matrices (and hence makes no use of solver libraries such as PETSc). Abstractions such as Vector and Set are there to hide or encapsulate the implementation of data structures such as a bitset or C vector. Both Mesh and Swarm use the Decomposition class and are abstract classes (almost interfaces, but they have properties associated with them). Implementations of these classes can be created at load time. The distributed numerical classes use MPI for communication.

Many different solvers can be built on the StgDomain layer, such as Finite Element, Finite Difference, and Explicit solvers such as SPH. In most cases, it is not necessary or worthwhile to develop new Matrix solvers from scratch, as there is a wealth of existing solvers available, from toolkits such as PETSc and Trilinos. So StGermain often follows the strategy of "wrapping" existing solvers, in particular StGermain solvers make frequent use of PETSc for implicit solvers and the preconditioner and multigrid abstractions of PETSc. Despite the frequent use of PETSc solvers (and solvers that PETSc itself wraps, such as HYPRE), it is not really correct to say that StGermain is "built on" PETSc, but rather that StGermain "uses" PETSc, where appropriate, though a fairly narrow interface that allows other solvers to be incorporated.

The following solvers have been developed on top of StgDomain and are in production use:

1. StgFEM – an implicit finite element solver framework, that uses PETSc
2. PICellerator – a Particle In Cell (PIC) Lagrangian integration scheme and constitutive rule framework, that uses StGFEM. PICellerator is a successor to the PIC algorithms and technology first developed in Ellipsis [20]
3. Snac – a explicit finite-element continuum solver, similar in formulation to FLAC (Finite Lagrangian Analysis of Continua)
4. SPModel – a surface-process (erosion and transport) solver framework, that extends the technology used in Cascade

The Solver layer includes software tools, built using StgDomain, such as gLucifer – a visualization framework that provides both interactive and background rendering but concentrates on the latter for rendering frames of computationally

intense, remotely running applications (i.e. generates 4-D streaming movies while the calculation is running). It uses open source, freely available libraries such as OpenGL, X11, VTK, and libfame (a movie encoder).

All the StGermain solver frameworks are 3D and 2D parallel, and distributed (using MPI and parallel libraries such as PETSc).

Computational scientists can develop new applications from scratch by using solvers programmatically, but many scientists prefer to specify models just from a configuration file or script to an application framework. A StGermain Application Framework specializes a StGermain Solver to an application domain, adding precompiled components and plug-ins appropriate to that domain, and makes all this available to a scientist using a declarative XML model configuration. The StGermain application frameworks include:

1. Underworld – for geodynamics applications (stokes flow) built on PICellerator
2. Xanthus – for metal alloy deformation modeling, built on PICellerator
3. Gale – for geodynmics applications (long-term lithospheric deformation), a Generalized Arbitrary Lagrangian Eulerian model, built on PICellerator

Each application framework takes an XML model configuration file as input, which controls:

- Geometry and mesh setup – dimensionality, mesh size and shape
- Simulation parameters – e.g., timesteps and convergence/error tolerances
- Boundary and initial conditions
- Output – variables output to a journal file, visualization parameters (viewports)
- Contexts and plug-ins (e.g., for PDEs or material laws)

StGermain applications make heavy use of both contexts and plug-ins, which are object-oriented design patterns that facilitate code adaptation. A context is a set of functions that can be used to customize the behavior of a solver or model. For example, viscosity can be calculated using several different laws in Underworld, including:

- Arrehenius (viscosity is a function of depth and temperature)
- Frank Kamenetskii (viscosity is a function of temperature)
- Non-newtonian (viscosity is stress dependent)

Each of these laws is implemented by a different function, or *strategy*, called from the same places in the solver, with the same interface (function specification). In traditional scientific applications, alternative strategies would be supported either by editing and recompiling the code or by “case” statements that control which strategy is called. Experience is that maintaining applications with many case statements soon becomes unmanageable due to the interdependencies between various case statements and the tight coupling between the strategy functions and the code that they are embedded in. In StGermain, strategies are bundled together into contexts, and at link-time the strategies used by a model are specified by the XML Configuration file written by the scientist. A computational scientist or numerical analyst can *extend* an application or solver framework respectively by defining a new strategy (e.g., a new viscosity law), or *generalize* an application or solver framework

by replacing a fixed piece of code by a set of strategies (e.g., for calculating higher-order elements in a Finite Element Solver).

Conclusion

Scientific Software Frameworks for *superstructure*, such as web interfaces to scientific applications, are fairly well supported – there is no lack of toolkits for building websites or for producing bindings to legacy applications.

By contrast, scientific software frameworks for *infrastructure*, such as PETSc, Trilinos, or StGermain, face many challenges, including:

- *Portability* – to different programming languages and parallel computing platforms.
  While C/C++ and Fortran are the dominant languages for scientific computing, there are others. While there is not as much diversity of parallel computing platforms as there was a decade ago, there is still considerable diversity across Linux boxes, and many boutique parallel systems and emerging systems (such as Cell processors). Some scientific developers have gone down the track of using a specification or Interface Definition Language (IDL) that can be compiled or linked to any other language (e.g., Babel/SIDL [8]), but such efforts are more in a research stage than production, and IDLs have not historically had much impact (e.g., UNCOL).
- *Performance* – scalability to peta-scale computing platforms.
  There is a major push to scale scientific applications up to peta-scale computing (tens of thousands of processors). At such scales, any minor overheads in communication or load imbalance will have catastrophic performance impacts. Inevitably, peta-scale computing requires careful platform-specific tuning of an entire application, including and libraries used by the application. For scientific software frameworks, this presents a real challenge, as tuning the framework adds complexity, and tuning for one application/platform may detune the framework for another.
- *Install-ability* – of the framework on a new platform.
  Portability measures the effort or complexity of extending the framework by the *framework developers*. By contrast, install-ability measures the effort or complexity of installing and using the framework by *scientists*. Most frameworks have a high degree of dependence on other libraries and platform tools, and so installing and upgrading a framework can be daunting for scientists, who generally have limited programming expertise.
- *Complexity* – of the framework and its applications.
  A single scientific application, no matter how difficult to maintain, will always be "simpler" than a framework. As a framework grows in capability and sophistication it becomes more and more complex – and harder to learn and apply.

All the above challenges are solvable, but at a cost in development, support, documentation, and training that is high relative to the size of the community.

Inevitably, tradeoffs need to be made between such challenges. Also framework support costs cannot be justified by an economic return by commercialization of scientific software frameworks – the academic research community is just too small and under funded to justify extensive commercialization of scientific frameworks. The NAG framework is a commercial scientific framework; albeit somewhat less sophisticated than, say PETSc or StGermain. However, NAG has had fairly limited impact due to the scientific communities focus on open-source or free software. Another approach that has been mooted is the use of "plug-ins" to commercial packages such as Matlab or Mathematica. Matlab and Mathematica are interactive environments in widespread research and teaching use, with a mathematics-oriented input language, that enable scientists to create scientific models faster than with traditional programming languages such as C, C++, and Fortran. They both support parallel plug-ins, but little infrastructure support for developing parallel solvers or models. In effect, both Matlab and Mathematica are convenient superstructure frameworks that any infrastructure framework, such as PETSc or StGermain could be coupled to.

While the use of scientific software frameworks is increasing, the funding is low relative to the demand, and there have been many "false starts" on developing frameworks by well-intentioned computer scientists. Inevitably, the use of frameworks and their sophistication will increase, but it will be a long, slow process.

References

1. W. Kozaczynski, and G. Booch, Component-Based Software Engineering, *IEEE Software* 15(5), 34-36 (1998).

2. B. Boehm, Managing Software Productivity and Reuse, *IEEE Computer*, 32(9), 111-113 (1999)

3. http://en.wikipedia.org/wiki/Grid_Computing.

4. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao, and Y. Zhao, Scientific Workflow Management and the Kepler System, *Concurrency and Computation: Practice and Experience* 18(10), 1039-1065 (2005)

5. http://www.ncbi.nlm.nih.gov/blast/

6. http://www.cacr.caltech.edu/projects/pyre/

7. http://www.ks.uiuc.edu/Research/namd/

8. B. Allan, R. Armstrong, A. Wolfe, J. Ray, D. Bernholdt, and J. Kohl, The CCA core specification in a distributed memory SPMD framework, *Concurrency and Computation: Practice and Experience* 14(5), 323-345 (2002)

9. http://en.wikipedia.org/wiki/Software_prototyping

10. M. Fowler, *Refactoring: Improving the Design of Existing Code* (Addisson-Wesley, New Jersey, 2000)

11. http://www.netlib.org/lapack/

12. S. Balay et. al., PETSc Users Manual, Argonne National Laboratory, Technical Report No. ANL-95/11 - Revision 2.3.2 (2006)

13. M. Heroux et. al., An Overview of Trilinos, Sandia National Laboratory, Technical Report No. SAND2003-2927 (2003).

14. https://csd.vpac.org/twiki/bin/view/Stgermain/

15. S. Quenette, B. Appelbe, M. Gurnis, L. Hodkinson, L. Moresi, and P. Sunter, An Investigation into Design for Performance and Code Maintainability in High Performance Computing, ANZIAM J. 46(e) pp. C1001—C1016, 2005

16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software* (Addison-Wesley, Massachusetts, 1995)

17. http://en.wikipedia.org/wiki/Reflection_%28computer_science%29

18. http://en.wikipedia.org/wiki/Declarative_programming

19. G. Myers, *Composite/Structured Design* (Van Nostrand Reinhold, New York, 1978)

20. L. Moresi, F. Dufour, and H. Mulhaus, A Lagrangian Integration Point Finite Element Method for Large Deformation Modeling of Viscoelastic Geomaterials, *Journal of Computational Physics*, 184, 476-497 (2003)