An investigation into design for performance and code maintainability in high performance computing

S. M. Quenette* B. F. Appelbe† M. Gurnis‡
L. J. Hodkinson§ L. Moresi¶ P. D. Sunter‖

(Received 29 October 2004; revised 12 September 2005)

Abstract

Maintaining and adapting scientific applications software is an ongoing issue for many researchers and communities, especially in domains such as geophysics, where community codes are constantly evolving to adopt new solution methods and constitutive laws. Traditional high performance computing code is written in C or Fortran,

*VPAC, Melbourne, Australia. mailto:steve@vpac.org
†VPAC, Melbourne, Australia. mailto:bill@vpac.org
‡California Inst. of Technology, Pasadena, USA. mailto:gurnis@gps.caltech.edu
§VPAC, Melbourne, Australia. mailto:lhodkins@vpac.org
¶Monash University, Melbourne, Australia.
mailto:louis.moresi@sci.monash.edu.au
‖VPAC, Melbourne, Australia. mailto:pds@vpac.org
See http://anziamj.austms.org.au/V46/CTAC2004/Quen for this article, Published October 7, 2005. ISSN 1446-8735

which offer high performance but are notoriously difficult to evolve and maintain. Object-oriented and interpretive programming languages (such as C++, Java, and Python) offer better support for code evolution and maintenance, but have not been widely adopted for scientific programming, for reasons including their performance and/or complexity. This paper describes our approach to developing scientific codes in C that provides the flexibility of interpreted object-oriented environments with the performance of traditional C programming, through techniques including entry points, plug-ins, and coarse grained objects. This approach has been used to implement two very differently formulated scientific codes in active use and development by the geophysics scientific community.

Contents

1 Motivation

Scientific software maintenance and evolution (adaptation) often requires changing the program's internal structural boundaries [1] (not just localized replacement or extension of one component or structure). Such structural changes are required for improving performance, changing a numerical technique or a constitutive relationship. Yet internal structural boundaries, such as the data stored on a mesh point and its access and distribution, are the lowest point of usable abstraction in the program. Changing these structural boundaries is changing the very essence of the program.

For example, if an existing finite element method code is to have a Lagrangian integration point scheme added to it, the mathematical model for integration is significantly different [2]. To efficiently implement this there needs to be a element/shape-function template for each element, instead of a single global element/shape-function template. Unless the original code was designed with this adaptation in mind, major recoding is needed.

Adaptation is commonplace for computational modellers. Constitutive rules are becoming increasingly complex, incorporating a greater spectrum of physics. Similarly, numerical techniques are drawing in more mathematics, and the chosen technique is often a balancing act between performance and accuracy that is specific to a problem domain. The need for adaptation is compounded because implementations are often experimental, iterative steps towards explaining real world observables.

The criterion for completion of a scientific research project is usually one or more publications. At that point software development stops, with little or no concern for future work. In general, scientists try to minimize the complexity and sophistication of the software they develop, and focus instead on research outcomes.

The problem is not unique to computational science. The issue of increasing complexity and associated development cost has been a significant

software engineering topic for the last 30 years. Ironically, in the late 80s, the mathematical software community was acclaimed for the successful early adoption of reuse (for example the BLAS libraries) [1, 3]. This success has been attributed to this form of mathematics (linear algebra) possessing a rich, standard nomenclature [1]. However, modelling of geodynamics, is not so well defined and understood. It is experimental—the code changes. There are no further "obvious" opportunities for reuse.

Since the mid 80's the adoption of three related software engineering techniques has led to more adaptable software, driven largely by commercial software development:

1. Object-Oriented (OO) programming
2. Component-Based Software Development
3. Rapid Applications Development (RAD) tools, environments, and languages

OO programming languages, such as Java, C++, and C#, are now the dominant programming languages for modern commercial software development. However, despite several major efforts, such as Java Grande [11] and POOMA [10], OOP has made little inroads into mainstream scientific software development. The primary reason for this has been performance: dynamic OOP languages incur a significant performance penalty. A secondary reason is that objects and classes are difficult abstractions to apply to continuum physics, beyond the coarse grained abstractions of matrices and vectors.

Component-based development means building an application by assembling components from existing frameworks (large-scale integrated component libraries, typically OO). Framework libraries such as J2EE and .NET are in widespread use for commercial software development, and contain thousands of reusable and adaptable classes. By contrast, scientific libraries, such

as PETSc [9] and NAG, are used on a relatively small scale and generally are useful only for implicit solvers.

Rapid Application Development (RAD) techniques helped to validate the Evolutionary Development [4] approach. This has since spawned into a collection of techniques collectively known as Agile Software Development (for example, XP, FDD, and Scrum) [5, 6]. Hence the popularity of tools such as Python [7]. Python makes rapid development a realistic goal, and it does this by focusing on enabling capabilities, such as: component integration, reducing artificial complexities, build-cycle turnaround, whilst still being a full featured programming language [7]. However, it is not feasible to write fine-grain computational code with Python, because of its performance overheads.

What is needed is the introduction of OO frameworks and python-like abilities into traditional computational science languages such as C and Fortran, without sacrificing performance. We describe how we have done this with the StGermain framework.

2 Python and adaptation

To minimise the need to do adaptation, and to minimise the cost of future adaptation, a Python software project is encouraged to deal with change from the outset. Python's popularity is a direct function of its philosophy, that is, to deal with change. This is evident in many of its features:

1. Python is interpretative: build time is eliminated. Large compiled projects take considerable time to build and this affects a developer's choice on when and how often to make changes, rebuild, and retest.
2. Python supports OO development: software components can represent real-world objects, raising the program structure closer to the appli-

cation design. OO development implies encapsulation and information hiding which tend to localize code changes needed to support adaptation.

3. Python supports fully dynamic typing: the type of an object can change at run-time. If a class (template of a real world object) does not possess properties required by another piece of code, the program can add this property to the class at run-time. The user of the class does not need to ensure this property is available at compile time. In strongly typed languages, such as Java and C++, class definitions and types are fixed, and users of this class are constrained by this. Full dynamic typing is useful in scientific software development as it allows behaviors to be added to user-defined class without defining new classes or recompiling.

4. Python is easily extensible: creating wrappers of code written in other languages, and installing these as python modules for use is easy, consistent and standardised. This encourages reuse, localises changes, and easily allows derivative works without needing to modify the source module(s).

Python's flexibility comes at a cost in performance. All data items are complex runtime entities, including member functions. For this reason we do not use Python to write complete HPC codes. Rather, it is utilised in a coarse-grain manner, to write the superstructure or top-level main control program, calling upon fine grain code written in languages such as C and Fortran. This works well for stable codes, where the use of the code is typically of a production nature (that is, executing large runs). However, when the numerics or physics requires change, then fine grain and structural code change is required. The challenge is to support such structural change without sacrificing performance

3 Bringing adaptability to HPC

HPC languages and typical applications are philosophically quite opposite to that of Python. They are designed for run-time speed. Any choice that can be made at compile time is eliminating unnecessary work at run-time and permitting further hardware based optimizations by a compiler. Computational codes typically perform the same subset of mathematical operations on thousands or millions of discretised points, for many time-steps, in "inner loops". If the run time for these mathematical operations in inner loops is doubled, then the total simulation time will double too. This can halve the number of simulations that a scientist can run, or the size of the problem that can be modeled, hence rendering adaptability of little value.

Typical HPC languages, such as Fortran and C, are compiled, not interpreted. However, an approximation to interpreted scripting can be achieved by using input files to set all possible parameters to a problem, both data and algorithmic, rather than hard-coding them as constants in the program. Coding in C has a major advantage as C is a "machine oriented high-level language". By using C, almost anything can be implemented including highly adaptable applications (ranging from C++ compilers to Java interpreters). There is, however, a sliding scale on the performance costs in introducing adaptability.

There are two key issues in adaptability:

1. Data — are objects allocated dynamically, and at what level?
2. Algorithms — are functions and operations dynamically specified, and at what level?

TABLE 1: Array allocation methodologies

	Methodology	Average time in seconds (relative) GCC no optimisations	GCC optimised
1	Static	22 (1)	21 (0.95)
2	Dynamic	22 (1)	21 (0.95)
3	Coarsely object-oriented	22 (1)	21 (0.95)
4	Finely object-oriented	35 (1.59)	34 (1.55)
5	Coarsely and finely	35 (1.59)	34 (1.55)

3.1 Data adaptability

As an example, for a finite element or finite difference code, an obvious parameter for sourcing from an input file is the mesh size. This parameter has a direct mapping to the amount of memory required to maintain the mesh and the qualities maintained upon it. There are many other parameters of the same nature. Traditional Fortran methodology relies on knowing the array allocation size at compile time for compiler optimization and hence faster execution. For example, in supercomputing's golden days, a Fortran compiler could map this array to its own memory *segment*, where segment registers were a relatively scarce resource, but were fully resolved as part of memory fetching [8]. Other pieces of data, such as a particular material property, would reside inside one of these segments as an offset. In Fortran these segments are referred to as *common blocks*. However, processor memory models have evolved, allowing programming languages to utilise the same high-speed low-level processor infrastructure. That is, programming constructs such as structs and classes can be segments, not just the traditional programming constructs of stack, code, heap, common block, etc.

Table 1 compares averaged times of the idealised representations of array allocation methodologies. Each create arrays indicative of storing and manip-

ulating coordinates from velocities of a mesh following a different methodology: static, dynamic, coarsely object-oriented, finely object-oriented. These are a natural progression from a traditional-Fortran hard-wired problem (1), to run-time provided problem size ability (2), to enabling component reuse at compile and run-times (3,4,5). The step from hard-wired to run-time problem descriptions are now commonplace, and essential for maintainability which is in-turn essential for experimental science. The ability to adopt OO methodologies are also essential for reuse, reduction in complexity and contemporary levels maintainability. The essential factor is how "fine" objects are defined, as a compiler's implementation of objects cost more than traditional methods. We define coarse object orientation as creating abstractions of higher level or closer to real world items, such as "Mesh" or "Solver". There are few of these in the system and they are large scale contiguous memory structures. Conversely, fine refers to items that are numerous, and small scale, such as "Node" or "Point".

The suite of idealised codes can be found at `http://csd.vpac.org/CtacAnziamPaperRuns`. The tested platform is a Pentium 4 with sufficient physical memory (1 GB) to prevent paging. The number of nodes is 25 000 000 (approximately 200 MBs), and the equation is iterated 50 times. An initial loop is used to page-in the array and initialise it with values. For simplification, 1D arrays are used. For control, C is the only language used; classes are mimicked via structures. GCC is the compiler, with no optimisation achieved through the flag `-O0` and optimisation achieved through the flags `-march=pentium4 -O3`. The aim is not to get definitive results, but to ascertain confidence in established rules of thumb. CPUs have many finite registers and other resources that are not exhausted in these experiments.

The results are interesting because the performance of all but the finely OO methodology are the same. The finely object-oriented methodology yields execution times that are over 50% slower. Memory accesses to any data item, whether it be a global variable (common block), on the stack, on the heap or inside a structure, is accessed in the same way; by base-offset pairs. The

TABLE 2: Procedural methodologies

	Methodology	Average time in seconds (relative) GCC no optimisations	GCC optimised
1	inlined (coarsely OO)	22 (1)	21 (0.95)
2	per iteration static function	22 (1)	21 (0.95)
3	per index static function	28 (1.27)	21 (0.95)
4	per index function pointer	28 (1.27)	24 (1.09)
5	per index entry point	48 (2.18)	26 (1.18)

variable itself is merely an offset to a base memory location. The processor maintains a relatively small (roughly 32) register bank for these base pointers. These registers are optimised for calculating the final address of the base-offset pairs required as part of typical processor operations. However, loading an address into a base register is an extra cost. In the finely object-oriented approach, the base changes for each index (as opposed to once), and hence invoking an extra memory operation per loop.

The conclusion is simply that fine-grained OO data allocation should be avoided, but large-scale coarse-grain OO allocation incurs no significant penalty (and large-scale allocation can contain fine-grained objects).

3.2 Algorithmic adaptability

Table 2 compares averaged times of the idealised representations of procedural methodologies. Each modify the coarsely object-oriented example above with the following function calling methodologies: inlined, per iteration static function, per index static function, per index function pointer, per index dynamic function pointer. These take a natural progression from a hard-wired monolithic code (1), to a hard-wired piece-wise code (2), to run-time in-

terchangeable code (4). The step from hard-wired monolithic to piece-wise breaks of the problem into many discrete functions. This is done to reduce code complexity and promote code reuse.

The step to run-time interchangeable enables functionality without having to recompile code. It is where the actual function that gets run is chosen/ resolved at run-time (say through a configuration file option), rather than at compile time. This can be achieved in C by a function pointer. The scheme enables changes without changing already compiled code. However, in practical applications the ideal point for creating the function interface is often at a "per index" level (3). For example, in building the element stiffness matrix: the behaviour changes as the physics modelled changes, but it gives unique values per element. Furthermore, assembling the composition of each element's contribution into the global stiffness matrix is a complicated and critical task. The practical problem is further compounded by the desire to take a working model and add to it. Continuing the above example, plasticity is to be added to an elastic stiffness matrix build routine to model elasto-plastic behaviour. Essentially the way to enable this is to add a second function to the one function pointer. We define an entry-point as equivalent to a function pointer that may have more than one actual function to run (5). It requires coding infrastructure beyond that supplied by C's standard libraries or simple function pointers.

Once again the results are interesting. As expected the difference between inlined and calling a function that has all the work inside it is negligible. Similarly, the difference between a static function and function pointer (in the non-optimised case) is negligible, most likely attributed to prefetching. As expected, the overhead in running an entry-point is significant (in the non-optimised case).

While the optimisation abilities of the compiler had little influence in the array allocation comparisons, they have significant benefits in the cost of calling functions. In essence, dynamic adaptability of algorithms and operations in C comes at low cost (except in obvious cases such as invoking

a function call for a trivial operation in an inner loop that might have been inlined).

4 A plugin model for adaptable HPC

The performance cost of coarsely object-orientation is negligible, and hence it makes sense to adopt it for adaptability. To enable functional adaptability, the only real option is to use entry-points (function pointers). This is approximately 20% slower in the optimised case, but the overhead decreases as the work done per function increases. Fine object-oriented methods also provide functional adaptability, but are generally not advisable, even in C, due to the number of objects that may be introduced, and the penalty for loading many base pointers and lack of locality. Our results show that it is better to have many arrays rather than an array of a struct.

Plugins are (compiled) code modules that are loaded into a running program. The running program needs to know what to do with the plugin, and the plugin has to subscribe to the requirements or conventions dictated by the running program. If the program has entry points at key points of change, then a plugin may add functions to that entry-point. Plugins only requires that the entry-point manager is passed in from the running program. This is similar to importing in Python, but in Python's case, the management is hidden from the users perspective. However, note that the location of these entry-points are numerical scheme dependent, and the key to effective maintainability. Learning where to place them is an important design activity, although adding new entry-points does not usually require major structural change.

An infrastructure framework can support entry-points and plugins for users. The generic creation of entry-points, their management, and the loading of plugins are all reusable components. Also useful and reusable is a scheme for creating new arrays and attaching them to the running system

(that is, mimicking dynamic typing). Another reusable component is wrapping the application in a Python style component interface, so that applications can be coupled. Our implementation of these, and many other features, is in a framework named StGermain (http://csd.vpac.org/StGermain).

The user community (meaning the scientific programmer and modeler community) for StGermain is quite large and growing (over 20 users spread over 6 institutions over 2 years). This augers well for the long-term success of our approach.

4.1 Snac

Snac is a mixed discretisation finite difference, finite element solid mechanics code suited to crustal deformation problems. It is explicit, 3D and parallel. It utilises a regular hexahedron element mesh. There are fourteen plugins providing features such as

- elastic, plastic and viscous material models,
- Cartesian, cylindrical and spherical geometry models,
- temperature modelling,
- remeshing, and
- coupling to CitComS (another computational code).

Snac was primarily motivated by the need for a 3D parallel version of FLAC (Finite Lagrangian Analysis of Continua). It uses the infrastructure described in this paper to achieve rapid development. Further information can be found at http://csd.vpac.org/Snac.

4.2 Snark

Snark is a finite element code originally created for mantle convection problems. That is, a solver for incompressible Stokes flow (that is, no inertial terms). It is implicit, 3D and parallel. It also utilises a regular hexahedron element mesh. Snark is able to switch between using an Lagrangian integration point integration scheme, and traditional finite element integration schemes. There are approximately 30 plugins providing features such as

- various viscosity models,
- various input conditions,
- various energy solvers,
- various momentum solvers, and
- various Lagrangian integration schemes.

By utilising the infrastructure described in this paper, Snark is now a collection of geodynamics codes, including mantle convection, slab initiation and basin evolution modelling. The infrastructure enables high levels of code reuse, so that the differences are only the particular numerics, physics or boundary conditions specific to the problem. Furthermore, some people within the group are only concerned with the further development of numerical techniques, and through this mechanism they have a mechanism to ensure the group can use them. Further information can be found at http://csd.vpac.org/Snark.

5 Conclusion

Scientific software development had been impeded by the lack of any effective compromise between adaptability and performance. Most scientific soft-

ware developers target performance, by coding in C or Fortran from scratch. Those developers that targeted adaptability, using C++ and Java, have not gained traction for performance and complexity reasons. Our experience, applications, and benchmarks show that performance need not be sacrificed for adaptability. Using course-grain objects and plugins, and the flexibility of C to support dynamic typing, we have built a framework that supports both adaptability and performance, using conventional programing languages targeted at performance. The growing size of the user community for our framework augers well for its widespread adoption and influence.

Acknowledgment: The VPAC Computational Software Development team members, who are supported by:

- ACcESS, http://www.access.edu.au,
- GeoFramework, http://www.geoframework.org,
- APAC, http://www.apac.edu.au.

And our project collaborators:

- Snark: Louis Moresi, David May, Dave Stegman, Robert Turnbull at the University of Monash;
- Snac: Mike Gurnis, Michael Aivazis, Eun-seo Choi, Puruv Thouriddey, Eh Tan at the California Institute of Technology. Luc Lavier at the University of Texas.

References

[1] F. Brooks, Jr. *The Mythical Man-Month (20th Anniversary edition).* Addison–Wesley, 1995. C1003, C1004

[2] L. Moresi, D. May, J. Freeman, B. Appelbe. *Mantle convection modeling with viscoelastic/brittle lithosphere: Numerical and computational methodology.* Lecture Notes in computer Science 2660: Computational Science ICCS2003, p.281, 2003 C1003

[3] B. W. Boehm, P. N. Papaccio. *Understanding and Controlling Software Costs.* IEEE Transactions in Software Engineering, p.1472, 1998 C1004

[4] S. McConnell. *Rapid Development.* Microsoft Press, 1996 C1005

[5] http://www.agilemanifesto.org C1005

[6] D. Anderson. *Agile Management for Software Engineering.* Prentice Hall, 2003. C1005

[7] http://www.python.org C1005

[8] G. Silberschatz. *Operating system concepts (5th Edition).* Addison–Wesley, 1998 C1008

[9] S. Balay, V. Eijkhout, W. D. Gropp, L. C. McInnes, and B. F. Smith. *Efficient Management of Parallelism in Object Oriented Numerical Software Libraries.* Modern Software Tools in Scientific Computing. E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, pages 163-202, Birkhauser Press, 1997 C1005

[10] W. Humphrey, R. Ryne, T. Cleland, J. Cummings, S. Habib, G. Mark, and J. Qiang. *Particle beam dynamics simulations using the POOMA Framework.* Lecture Notes Computer Science 1505, 25 (1998), Proceedings ISCOPE'98 (Santa Fe, NM 1998) C1004

[11] http://www.javagrande.org C1004