

An investigation into design for code maintainability in HPC

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

24 September 2004

Abstract

There are two ways to interpret a title such as “A Plug-in based design for code maintainability in HPC”, based on whether you are through and through a HPC traditionalist, and then there is everybody else. But with realisation of computer software development cost, the commoditisation of clustering and the requirement of cross disciplinary science, scientific code evolution and maintenance for researchers is a real issue. This paper investigates what performance costs does one bare for the flexibility and maintainability of HPC software. If we can’t be fully flexible, what can be? With all modern HPC platforms facilitating expected features such as dynamic libraries, concepts such as plugins can be considered. We explain how and where and why such concepts may be utilised. Then we offer two indicative examples of two very differently formulated scientific codes.

Contents

1	Motivation	2
2	Software Maintenance and Evolution	3

*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, Melbourne, AUSTRALIA. <mailto:louis.moresi@sci.monash.edu.au>

||VPAC, Melbourne, AUSTRALIA. <mailto:pds@vpac.org>

3	Python and Change	4
4	Bringing the ability to change to HPC	5
5	A Plugin model for HPC	8
5.1	Snac	8
5.2	Snark	9
6	Conclusion	9

1 Motivation

Many of the real needs for upgrading a program, such as performance, especially attack its internal structural boundaries [1]. This is not just relevant to performance, but other facets of scientific computing, such as numerical techniques and constitutive relationships. Yet in here lies a problem: internal structural boundaries are the lowest point of usable abstraction in the program. It forms the language and hence constraint on how the real problem is defined. Changing these boundaries is changing the very essence of the program.

For instance, if an existing Finite Element Method code is to have the Lagrangian Integration point scheme added to it, the mathematical model for integration is significantly different [2]. To efficiently implement this, instead of implementing a single element/shape-function template, there needs to be an equivalent for each element. Was the original code designed with this in mind? How much time is spent enabling it?

Such activities are common place for computational modellers. Constitutive rules are becoming increasingly complex, incorporating a greater spectrum of physics. Similarly, numerical techniques are drawing in more mathematics, of which the chosen technique is balancing act between performance and accuracy, and this is often specific to a problem domain. This is compounded in that for a computational modeller, any implementation is experimental, and a hopeful iterative step towards explaining real world observables. The criterion for completion of the project is the ability to produce a paper. At that point development stops, with little or no concern for future work. And what happens if then the modeller whats to share or exchange their models?

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 had heralded acclaim for the successful early adoption of reuse (e.g. BLAS, etc). [1] [3]. This success has been attributed to this form of mathematics being arcane and possessing a rich, standard nomenclature [1]. However, modelling of geodynamics, for example, is not so well defined and understood. It is experimental... the code changes.

Recent development in software development techniques focus on dealing with change. The Rapid Application Development (RAD) technique helped to validate the Evolutionary Development [4] approach. This has since spawned into a collection of techniques collectively known as Agile Software Development (e.g. XP, FDD, Scrum) [5] [6]. Hence the popularity in 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 tangible to write fine-grain computational code with Python. So how do we introduce pythonesque abilities into traditional computational science languages? At what performance cost?

2 Software Maintenance and Evolution

Software maintenance is the changing of a program after delivery. These changes are typically to repair design defects found by end users. The number of defects increases with the number of users. Some of the required change attack internal structural boundaries of the program. It is hence no wonder that maintenance is often assumed at 40% of development cost [1].

Software evolution is the response to changing requirements on an existing program [9]. This may, for example, be to add a new feature. Often it is treated separately to maintenance, however, it exhibits the same impact in terms of change. It also raises the question, at what point is the inability to easily change a defect?

Undoubtedly both are relevant to computational software developments. Evolution is inherent in the quest of science, and hence the ability for change is an implicit requirement of the software. Agile methodologies assume this for all developments. Furthermore, there is little requirement to distinguish costs based on defects and those based on requirement changes in academic research. For this reason we simplify the terminology and call both maintenance.

3 Python and Change

To minimise the need to do maintenance, and to minimise the cost of maintenance, a software project is encouraged to deal with change. 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. it is interpretative: Build time is eliminated. Large projects take time to build and this effects a developer's choice on when and how often to build.
2. it enables Object Oriented methods: Software components can represent real world objects, making communication with the client more effective, and hence reducing the likelihood of change. Plus changes are localised.
3. it has dynamic/un-typing: The specification of an object can change at run-time. If a class (template of a real world object) does not posses properties required by another piece of code, it can add it on the fly or assume it just be. This is significant. In strongly typed languages, class definitions are fixed, and users of this class are constrained to this fixed definition of the class. Yet the Object-Oriented view-point does not assume this at all; users of this class can perceive a greater definition to the class. In languages such as Java and C++, the Object-Oriented specialisation feature is typically used to mimic this ability, or the class must be changed.
4. the language is easily extensible: Creating wrappers from other languages, installing these and 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: performance. All data items are complex, including member functions. For this reason raw Python is not used to write HPC codes. Rather, it is utilised in a coarse manner, calling upon fine grain code written languages like C and Fortran. In this manner, Python is configuring the system to a specific problem. This works well for established codes, where the use is typically of a production nature. However, when the numerics or physics requires change, then the fine grain code is where the change is required.

4 Bringing the ability to change to HPC

Languages and typical applications for HPC are philosophically quite the 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 hardware based optimisations. Why is this significant? Because computational codes typically perform the same subset of mathematical operations on thousands or millions of discretised points, for many time-steps. If the cost of doing this subset of mathematical operations is say ten times more for a change-enabled program, then a previously 1hour simulation would take 10hours. This is likely enough to discourage a scientist from attempting the or as many experimental runs, hence rendering the ability for change worthless.

Typical HPC languages, such as Fortran and C are not interpretive. Yet an approximation to scripting can be achieved by better utilising input files. An effort can be made to place all scientific choices in input files, both data and algorithmic. There is, however, a sliding scale on the performance costs in introducing this feature.

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. The fortran-esque methodology infers that knowing the array allocation size at compile time was clearly advantageous at providing faster execution. For example, in supercomputing's golden days, a Fortran compiler could map this array to its own segment, where segment registers were a relatively scarce resource, but were fully resolved as part of memory fetching [10]. Whereas normal data would reside inside a segment, such as a common block as a relative offset. Since then, however memory models of processors have evolved, providing the same operational level for constructs within a program. This means, not only traditional segments such as the stack, code, heap, common block, etc; but items like structs and classes as well.

Table 1 compares averaged times of the idealised representations of array allocation methodologies. Each create arrays indicative of storing and manipulating 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-esque 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 common place, and essential for maintainabil-

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)

ity which is in-turn essential for experimental science. The ability to adopt Object Oriented 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 the end user understands what they are. Conversely, fine refers to items that are many in the system, such as “Node”.

The tested platform is a Pentium4 with sufficient physical memory (1GB) to prevent paging. The number of nodes is 25000000 (approximately 200MBs), 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 mimiced via structures. GCC is the compiler, with no optimisation achieved through the flag “-O” 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 object-oriented 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 variable itself is merely an offset to a base, of which the processor has a finite register bank for. 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.

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)

Table 2 compares averaged times of the idealised representations of procedural methodologies. Each modify the coarsely object-oriented example 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 interchangeable code (4). The step from hard-wired monolithic to piece-wise is the breaking 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. This enables changes without changing compiled code. In practical applications, however, the ideal point for creating the function interface is often at a “per index” level (3). For example, 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 or its standard libraries.

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 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). Whereas the optimisation abilities of the compiler had little influence in the array methodology comparisons, they have significant benefits in the cost of calling functions.

5 A Plugin model for HPC

The performance cost of coarsely object-orientation is negligible, and hence it makes sense to adopt it. With regard to procedural methodologies, to enable practical changes, the only real option is to use entry-points. This is approximately 20% slower in the optimised case, but is likely to be less as the work done per function increases. The finely object-oriented method shares a similar property, however it is not advised due to the number of classes that may be introduced, and the penalty for loading many base pointers. It confirms 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. The plugin only requires that the entry-point manager is passed in from the running program. This is similar to importing in Python, but the management is hidden from the users perspective. Note however, the location of these entry-points are numerical scheme dependant, and key to effective maintainability.

This also becomes an activity of infrastructure building. The generic creation of entry-points, their management, and the loading of plugins are all required items. Also to consider is a scheme for creating new arrays and attaching them to the running system (i.e. mimicing dynamic/un-typing). Another is to wrap the application into a component interface, such that it can be used in coupling. Our implementation of these, and many other features, is in a framework named StGermain (<http://csd.vpac.org>).

5.1 Snac

Snac is a mixed discretisation Finite Difference / Finite Element 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,
- coupling to CitComS (another computational code).

5.2 Snark

Snark is a Finite Element code suited to mantle convection problems. 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 schemes. There are approximately 30 plugins providing features such as:

- Various viscosity models
- Various input conditions
- Various energy solvers
- Various momentum solvers
- Various Lagrangian integration schemes

6 Conclusion

The realisation of computer software development costs with respect to scientific code maintenance for researchers has been a real issue. Practices in scientific code development proliferate this by concentrating solely on speed of execution. However, with the requirement of cross disciplinary science and increase in constitutive complexity, scientific codes need to be able to change. To our advantage processor and compiler technology have adjusted to also optimise programming constructs that promote maintainability. The use of Python for fine grain computational science may still be out of the picture, but we can mimic some of its abilities in languages like C by creating generic infrastructure. In both the Snac and Snark cases, the development and maintenance of these plugins span across two organisations.

Acknowledgements: The VPAC Computational Software Development (CSD) 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.
- [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, pp 281, 2003
- [3] B. w. Boehm, P. N. Papaccio. Understanding and Controlling Software Costs. IEEE Transactions in Software Engineering, pp 1472, 1998
- [4] S. McConnell. Rapid Development. Microsoft Press, 1996
- [5] www.agilemanifesto.org
- [6] D. Anderson. Agile Management for Software Engineering. Prentice Hall, 2003.
- [7] www.python.org
- [8] M. Lutz. Programming Python (2nd Edition) O’Rielly, 2001
- [9] G. Booch. Object-Oriented Analysis and Design with Applications (2nd Edition) Addison-Wesley, 1994

- [10] G. Silberschatz. Operating system concepts (5th Edition) Addison-Wesley, 1998