LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-674736

The Future of Software Engineering for High Performance Computing

G. Pope

July 17, 2015

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

The Future of Software Engineering for High Performance Computing

DOE ASCR requested that from May through mid-July 2015 a study group identify issues and recommend solutions from a software engineering perspective transitioning into the next generation of High Performance Computing. The approach used was to ask some of the DOE complex experts who will be responsible for doing this work to contribute to the study group.

The technique used was to solicit elevator speeches: a short and concise write up done as if the author was a speaker with only a few minutes to convince a decision maker of their top issues. Pages 2-18 contain the original texts of the contributed elevator speeches and end notes identifying the 20 contributors. The study group also ranked the importance of each topic, and those scores are displayed with each topic heading. A perfect score (and highest priority) is three, two is medium priority, and one is lowest priority. The highest scoring topic areas were software engineering and testing resources; the lowest scoring area was compliance to DOE standards.

The following two paragraphs are an elevator speech summarizing the contributed elevator speeches. Each sentence or phrase in the summary is hyperlinked to its source via a numeral embedded in the text. A risk one liner has also been added to each topic to allow future risk tracking and mitigation.

On the topic of software engineering management:

Multi-physics simulation codes may not be ready to operate efficiently on the new hardware by 2017, so do not set software readiness expectations concurrent with the hardware schedule. 1 Hiring and retaining staff to accomplish the multi-physics code porting tasks will be challenging. Most software developers prefer writing their own code in contemporary languages rather than modifying somebody else's legacy code. 2, 3 Development teams should consist of physical science domain experts and computer scientists both skilled in software engineering best practices. 5 A graded approach should be used to accommodate priorities of projects that produce robust simulation codes 8, as well as support researchers who focus on innovation and novel approaches 6.

On the topic of software engineering process and tools:

Development lifecycle and DOE compliance 4, 13 must favor best practices 10 that support agility 11 and automation 7 to solve the software engineering challenges 9, 12 posed by next generation HPC. Automated tools include support of test driven design, continuous integration, regression test execution, static and dynamic code analysis, the build /test/release process 18 and version control, all of which are essential. There are a variety of new generation of tools that take full advantage of future COTS developments 15, 26, such as embedded SQA tools within the delivered codes to expeditiously resolve user concerns 16, automated and optimized porting capabilities across various HPC architectures 17, visual debugging tools 22, memory optimization 21, 27, and tools to help automate the threading process 20. In addition, there have been advancements in timely vendor support of desirable library 25 and compiler standards 14, standards for data description 24, and the ability to better manage simulation results as ensembles 19. To this end, ample HPC resources should be allocated for dedicated rigorous code testing 23.

The future of Software Engineering for High Performance Computing

Collections of elevator speeches by those who will help make it happen.

DRAFT 7/10/2015

Management

Schedule Expectation: Rank 2.2

The time frame for getting the software running on the new architecture is too optimistic; the next gen hardware delivery has become associated with the year 2017. There is a natural tendency to assume the software will also be ready to run on and take advantage of the new architectures in 2017. However porting the software to the new architecture will require a significant amount of effort which may or may not coincide with hardware delivery. [1]

RISK: Schedule overly optimistic

Staffing: Rank 2.4

Staffing shortages, finding computational computer scientists and SQEs who relish rewriting other people's code when they could work on new code could be challenging.. When entry level and mid-career computer scientists look for positions are they going to want re-code someone else's C++ or Fortran code so it can run under OpenCL threading or are they going to want to work for the Googles of the world writing new code in Java? Finding the resources to modify the codes is going to be very challenging. [2]

RISK: Inability to attract CS top talent

Hiring and Retention of Staff: Rank 2.4

The large HPC codes which we develop are complicated beasts with a large number of interacting pieces. It takes a new hire a significant amount of time to get up to speed on one of our projects and become fully productive. That's an investment we shouldn't want to lose. However, we face the two fold challenge of having Silicon Valley over the hill from us. The firms there pay well, and that has the side effect of raising the cost of living throughout the Bay Area. We need to have a compensation package in place which makes us competitive with what computer scientists are making there, not only to keep the staff we have, but also to attract new staff members. We've had at least two offers declined in recent weeks because the salaries we were offering were not sufficient to cover the cost of living differences the candidates would experience if they came here. [3]

RISK: Attrition of existing employees

Software Development Process:

Managing the Ever-Increasing Need for Compliance-Driven Agile Development: Rank 1.8

The needs of the research community increasingly require highly-complex computing environments and simulations. As these needs continue to increase, the software methodology and associated tools must mature to allow scientists and engineers to rapidly and efficiently develop state-of-the-art simulation capabilities while maintaining alignment with national and international consensus standards. The use of automated tools in support of test-driven development, continuous integration, regression test execution, and version control practices aligned with strong software management standards is essential for future software development environments. Keeping these strategies in mind and enabling modeling and simulation to be developed in a fraction of the time previously required will revolutionize predictive simulation.[4]

RISK: Standards and compliance overly restrictive

Management of Advanced Code Projects Rank 2.2

As we tackle more and more complex science and engineering applications, with sophisticated algorithms and analysis capabilities, and make strides towards improved SQE, there is too much for a single PI to manage. Application projects must be set up with separate leadership roles to cover the targeted science, software design, discretization/solver choices, multi-core strategy, and SQE tools and processes. One management model is to have a project PI, who owns the choice of equations and analysis, and a separate code owner, who owns the software design and quality.
Moving in this direction requires science and engineering application development teams to cede some leadership to Computational Science experts that may have little expertise in the application domain, and cultural changes to the organizational funding and reward systems.[5]

RISK: Lack of SE experience

Graded Approach: Rank 2.7

While I think this appropriate SE and SQE processes are important, I have serious doubts any attempt at a grand unification will have much real impact, if any. In my interactions with others at joint ASC/OASCR meetings and workshops, I've noticed a significant difference of opinions and approaches to these issues (and other technical concerns as well). I think this is due largely to the differences in developer goals, reward systems, and development environments. Office of Science software projects are mostly driven by research goals, developers are rewarded for developing novel things and writing papers, and the developers and users are loosely connected. ASC code efforts (at least at LLNL) are driven by programmatic goals and user demands, we are rewarded for delivering on mission goals (often timeline driven), and developers and users interact daily. Processes and practices reflect these stark differences very clearly.[6]

RISK: Level of rigor too high or too low

Use of Appropriate Software Engineering Practices Rank 2.8

Two critical factors facing the sustainability of HPC software are the productivity of developers and the long-term maintainability of the software. The use of appropriate software engineering practices can greatly help in both of these areas. In terms of developer productivity, software engineering methodologies and tools can greatly reduce the amount of effort developers must devote to developing and modifying HPC software. In terms of long-term maintainability, using good software engineering practices will help developers write code that will be easy to update and modify as machines and science change over time. The biggest win in both cases is that the amount of effort scientists must devote to software can be reduced freeing up more time for other scientific endeavors. In both of these cases, the key factor is having appropriate software engineering practices. While the field of software engineering has a number of practices that have been show effective in traditional software engineering, there is a general belief that many of those practices will not work in scientific HPC domains. While there may be a grain of truth to this belief, the better answer is that proven software engineering practices need to be tailored to fit within the constraints of the scientific HPC environment. There is a need for long-term interactions between software engineering experts and scientific HPC experts to develop and validate these tailored practices. In addition, there is a need for mechanisms to share successes in this area to help other scientific HPC developers find practices that can be adopted on their projects.[7]

RISK: Development in hero mode and code correctness/maintainability suffer.

Upping the Perceived Value of Software Engineering Rank 2.8

The scientific software community for too long has had an unbalanced perspective on the value of good software engineering, and the advent of exascale computing is making that gap more important to address. With obvious exceptions, too often researchers are rewarded solely on the merit of publications of results, and not on building quality software that is usable and maintainable by a broader set of users. With the value of an exascale computer reaching about $150k/day in capital costs and NRE, and the operating costs (electricity, system administration, operators, etc…) adding perhaps another $75k/day - we cannot afford to have software running on these machines that has not undergone a rigorous process of static and dynamic analysis, regression testing, performance analysis, and debugging on a broad set of problems. Researchers should be encouraged to adopt at set of Best Practices for scientific computing assembled by experts in the field who have successfully developed production level software, and continued funding dependent upon demonstration that these practices are being fulfilled. Multidisciplinary teams included expertise in computer science and software engineering should be part of teams seeking exascale funding, and publications in how to improve on graded Best Practices valued in the research community.[8]

RISK Best practices not communicated

Attributes of a highly effective software environment Rank 1.9

- Communication conducted in a shared and searchable database
    - Periodic conference calls
- Tightly defined and enforced coding standards
- Documentation included in code, extractable on command

- Small builds
- Automated testing and processing of testing reports
    - Digital test reports
- Specific code objectives, with delivery dates
    - Flexible / agile approaches to achieving the code objectives
    - Specific requirement development part of coding and testing[9]

RISK: Development approach not agile

The computational science community needs more awareness, knowledge, understanding, and experience with best practices in software engineering.

Rank 2.5

We need to raise awareness within the community of the value of software engineering both to software quality (which is fundamental to producing credible and reproducible simulation results), and to enhancing software, and ultimately scientific productivity.

We need to expand the level of knowledge and understanding of software engineering best practices within the community. Most DOE computational scientists do not receive any formal training in software engineering during their education. My experience is that most of what people know is self-taught from books, and word of mouth. We need to do a much better job of educating software developers on software engineering.
They need not only knowledge of the tools and techniques, but also the understanding of the ideas underlying them that they can figure out how to adapt what they read or hear about to their own situations, without getting caught up in the SE miracle technique of the week.
Complementary to this, we also need to make a concerted effort to capture the software engineering experience of people in HPC computational science, and understand how we're both different from and similar to "classic" industrial software development.

Finally, we need to encourage and incentivize people to actually use SE best practices. I think an important aspect of this, especially as we're concurrently working on the knowledge and understanding issues above, is going to be personal interactions between experienced "software engineers" and software development teams to assess current practices, understand pain points, formulate plans to address the pain points, and coach the team through their implementations (with an assessment that contributes back to the understanding aspect of the previous point).

Another aspect of increasing the level of actual experience with SE best practices might be to ensure that the basic tools and infrastructure to support these practices are readily available to DOE software developers. There are many tools, and many ways they can be provided.
But today, they are mostly left up to individual projects, many of which do not have the skills, or interest in setting up and maintaining their own infrastructures. Many will be able to cobble together something that they consider adequate based on third-party services, poorly advertised, supported, and maintained institutional services, and the like. If quality tools are widely advertised and readily available at low or no direct costs to projects, they are more likely to be used. Further incentives deserve thought too. I could go on, but I won't. The elevator doors are opening.[10]

RISK: Contemporary tools not available

Introduction

Rank 2.0

Development of production-quality software from a research-driven computational science and engineering (CSE) / high performance computing (HPC) project is challenging. CSE/HPC software products tend to be long-lived and multi-component. Ideally they should have reusable components and rely on external components developed by other expert groups in order to ensure state-of-the art capabilities. However, this ideal is seldom achieved. Commercial software can become unavailable and software from other research organizations can be unreliable and poorly supported. Many issues must be considered by a research-driven software project: research productivity and credibility, reuse and upgrades, maintenance, support, shared development, continued research with mature software, balancing backward compatibility and change, and more.

While a great deal of work has been done in the general area of software lifecycle models, Lean/Agile lifecycle models seem particularly attractive for most CSE projects [ImplementingLeanSoftwareDevelopment2007, AgileSoftwareDevelopment2003, XP2]. There seems to be little work attempting to define software lifecycle models for research-driven CSE/HPC software.

The primary purpose of this short paper is to propose research into the broad adoption of a modern Lean/Agile-consistent software lifecycle model and framework that take into account the particular needs of the CSE/HPC community for both research and production projects. It is based on the proposed TriBITS Lifecycle Model [TribitsLifecycleModel_eScience2012].

Self-Sustaining Software

The primary goal of the proposed lifecycle model is the development of ***Self-Sustaining Software*** which is defined to have the following attributes:

Open-source: The software has a sufficiently loose open-source license allowing the source code to be arbitrarily modified and used and reused in a variety of contexts.

Core domain distillation document: The software is accompanied with a short focused high-level document describing the purpose of the software and its core domain model [DomainDrivenDesign2004].

Exceptionally well tested: The current functionality of the software and its behavior is rigorously defined and protected with strong automated tests [CSESoftwareTests].

Clean structure and code: The internal code structure and interfaces are clean and consistent.

Minimal, controlled internal and external dependencies: The software has well structured internal dependencies, and minimal external upstream software dependencies that are carefully managed.

Properties apply recursively to upstream software: All of the external upstream software dependencies are also themselves self-sustaining software (terminating in ubiquitous universal standards like C++98, Boost, etc.).

All properties are preserved under maintenance: All maintenance of the software preserves above properties (by applying Agile/Emergent Design, Continuous Refactoring, and other good Lean/Agile software development practices).

Software with the above properties poses minimal risks to downstream customer CSE/HPC projects. To the extent that a piece of software is not consistent with any of the above properties it poses a risk to downstream customer projects. The motivation for these properties and other issues are discussed in detail in [TribitsLifecycleModel_eScience2012].

An Agile Lifecycle for Research-based CSE/HPC Software

While the goal of the proposed Lean/Agile lifecycle model is to produce self-sustaining software, other properties of the software are also important and the process by which software is first created in a research project and is later matured has to be considered. Therefore, the proposed lean/agile lifecycle model defines several different maturity levels for CSE/HPC software:

Exploratory (EP): Primary purpose is to explore alternative approaches and prototypes.

Research Stable (RS): Developed in a Lean/Agile consistent manner with strong verification tests (i.e. proof of correctness) written at various levels (e.g. unit, component, and system levels) as the code/algorithms are being developed. Has a very clean design and code base maintained through Agile practices of emergent design and constant refactoring [EmergentDesign2008]. However, generally does not have higher-quality documentation, user input checking and feedback, space/time performance, portability, or acceptance testing. But is appropriate to be used by "expert" users. Provides a strong foundation for creating production-quality software.

Production Growth (PG): Includes all the good qualities of Research Stable code. Provides increasingly improved checking of user input errors and better error reporting. Has increasingly better documentation as well as better examples and tutorial materials. Maintains clean structure through constant refactoring of the code and user interfaces to make more consistent. Maintains increasingly better regulated backward compatibility with fewer incompatible changes with successive releases. Has increasingly better portability and space/time performance characteristics. Has expanding usage in more customer codes.

Production Maintenance (PM): Includes the good qualities of Production Growth code. Primary development includes mostly bug fixes and performance tweaks. Maintains rigorous backward compatibility with typically no deprecated features or breaks in backward compatibility. Could be maintained by parts of the user community if necessary (i.e. as "self-sustaining software").

The transition between the **EP**, **RS**, **PG**, and **PM** phases is meant to be smooth and without risk. Existing software is grandfathered in using the ***Legacy Software Change Algorithm*** [WorkingEffectivelyWithLegacyCode2005]. There are many other details and considerations related to

the definition and proposed implementation of this lifecycle model that cannot be discussed here for lack of space (see [TribitsLifecycleModel_eScience2012]).

Summary and Research Opportunities

We propose the adoption of a Lean/Agile lifecycle model for research-driven CSE/HPC software. The proposed model, if widely adopted, could dramatically improve the productivity and impact of CSE/HPC research and applications by providing a wide range of compatible high-quality advanced software capabilities from a wide range of foundational areas.

The primary research questions for the proposed Agile lifecycle model relate to how well it will work at scale across many organization over a long time period, what level of training will be needed to get is used effectively, and finding ways to measure the impact using various local and global measures. There is an ongoing attempt to implement this lifecycle model in the Trilinos project [TribitsDevelopmentPracticies]. A broader effort would include applying and adapting this model to other projects as well.[11]

RISK: Dev approach not agile enough

HPC Software Engineering Centers Rank 2.3

The future of Software Engineering (SE) for next generation High Performance Computing (HPC) will be like the past but worse if there are not some fundamental changes to how the work is funded, rewarded, and managed. The emphasis on research, publications, and proposal writing – for science *and* software research – creates a software development quandary, especially for the small teams and limited resources that seem to be common in today's research funding climate. Those individuals doing development are often torn between meeting science goals and writing quality software, with the science taking precedence in order to enable continued funding. Add to this the increasing complexity of writing, debugging, porting, maintaining, and running science software at scale and the future of HPC software seems bleak.

Part of the problem is the inherent nature of computational science. Meeting research deadlines often leads to the development of prototype, one-off, and stand-alone software programs. These programs are generally written by people trained in and rewarded for the science, not the development of quality software. Generalizing and hardening these codes for production use requires expertise, time, and funding, which tend to be scarce in an environment driven by science goals. This paucity of resources results in software developer's time often split between multiple, different projects that have distinct goals and employ different tools and technologies. The outcome is cognitive overload, reduced productivity, and the risk of eventual burnout.

A paradigm shift is needed that reflects the importance of the development of quality software to ensure a more positive outcome for the future of HPC Software Engineering. A significant portion of funding needs to be earmarked and distributed for establishing, maintaining, managing, and rewarding SE-HPC expertise. Additional funding is needed for work – separate from the science and research – on developing, productizing, and maintaining tools, system software, and libraries directed at reducing the

burden faced by software developers during and after the transition to new architectures to include more robust and scalable capabilities for interacting with the file system; launching, monitoring, analyzing, and visualizing the results of ensemble simulation runs; and debugging, testing, and porting HPC software. [12]

RISK: Funding for tools and process improvement not available

Compliance to DOE Standards: Rank 1.8

There will be large technical challenges for software moving to next gen hardware platforms. DOE needs to relax compliance rigor of standards, orders, and guidance to support a research environment. Demanding excessive non-value added rigor and documentation will kill creativity, slow progress, and drive out the best talent.[13]

RISK: Compliance to standard will be cumbersome

Standards support: Rank 2.3

Finally, we need to have the vendors of our HPC platforms commit to supporting language and library standards in a timely manner (18-24 months after release of the standard?) for the lifetime of the platform. We write codes which need to run on a variety of HPC platforms, and having different levels of standards compliance means that we have to code to the lowest common denominator. For example, it's been four years since the C++ 2011 standard was released. It contains many new features like lambda functions, auto type declaration, move semantics, etc which can improve the clarity, performance and maintainability of a C++ code. However, we currently cannot use C++ 11 on the BGQ platforms because the IBM C++ compilers do not, and will never, support C++ 11. Yes there are other compilers we could use on Sequoia, but we cannot suffer the ~20% performance loss we'd take by using them. An example of an important library standard is OpenMP 4.0. Much of our current planning for using next generation hardware revolves around using OpenMP 4 directives to offload computation to coprocessors (GPUs or Intel Phis), so we'll need robust support for this.[14]

RISK: Latest compilers, Libraries and operating systems not available or supported

Tools:

COTS Tools: Rank 2.0

The over one billion dollar per year commercial software engineering tool market is driven by market demand, so tools that help developers with threading and other HPC development challenges are going to be targeted for the most widely used platforms, such as Windows applications running on Intel processors. Tool providers are not motivated to produce tools for niche markets and platforms that are not widely used. It is important for the future of SE for HPC to make choices of operating systems, platforms, compilers, parallelism that are widely used to take advantage of commercial tool development and innovation or be prepared to have to build these tools in house. [15]

RISK: Not able to take advantage of main stream productivity tools

On Board SQA: Rank 2.1

Because of the complexity of the next gen software, SQA functions should be built right into the delivered software code to expedite product maturation. This SQA functionality could consists of:

1 Built in Tests
2 Instant Replay
3 Problem reporting
4 Problem resolving

Built in test allows the automated running of known problems validated from independent sources and check for expected results. These problems are specially designed to set up and run end to end simulations touching the major functions of the software. This function can be engaged on any new platform or platform variation as a confidence check that it is okay to proceed with new problems. Failure of the built in tests would indicate a hardware problem, configuration problem, or software problem.

The instant replay tool allows the user to back the simulation up to a previous check point and proceed forward with debugging enabled. This may help identify or isolate the source of the concern. Use of visual debugging tools would fit nicely into this on-board function.

Built in problem reporting allows user to capture and report to developers all the important environmental information about the hardware, software, configuration, versions, modes, compilers. flags, displays, processors, cores, stacks, etc. if a problem is detected either running built in tests or user supplied simulations. The reported issue is given a unique tracking number and status is set to investigate. The necessary information to repeat the problem is gathered and sent automatically.

The problem resolving tool tracks the reported problem through assignment, fix, ready for retest, and resolved. It asks the issuer of the problem to verify that the issue is resolved. Use of continuous integration allows developers to push out new releases multiple times per day.

Each of these SQA supportability features will be built into the main code and easily accessible to users. Because of the likelihood of numerous reported issues from users the development staff must plan on staff resources to fix and locally test the reported issues. SQEs can also be a resource for fixing issues found by users. The combination of the built in test tools, automation, and developer responsiveness using continuous integration to push out fixes rapidly allows the reliability of the next gen codes to mature at accelerated rates. [16]

RISK: Insufficient resources to support user needs

Maintaining a large scientific software base on multiple architectures (RAJA)

Rank 2.2

Suppose we have a large scientific software project containing tens of thousands of loops, and we must support that application on multiple resources simultaneously, such as a CPU and GPU. How might we

structure the software to make this simple to manage? First, we recognize that the GPU is most efficient when there are many floating point operations per byte, and the CPU is most efficient when there is a lot of random data access, and not as much work. We then come up with a mechanism to map loops to architectures based on type of workload.

Many frameworks and languages support the concept of a parallel for statement, so we can leverage that commonality. We create a new parallel for statement in C++ :

```
forall<policy>(iteration bounds)
{  body ; }
```

And we define specific policies that can help map loops to hardware resources:

```
#define work_intensive    bind_gpu
#define data_injtensive   bind_cpu
```

Code that contains many loops can be written like this in C++ using lambda functions for the loop bodies and specific loops will be bound to the CPU or GPU:

```
IndexSet domain_bounds ;
forall<work_intensive>(domain_bounds, [] (int i){
  /* body 1 */ ;
} ) ;
forall<work_intensive>(domain_bounds, [] (int i){
  /* body 2 */ ;
} ) ;
forall<data_intensive>(domain_bounds, [] (int i){
  /* body 3 */ ;
} ) ;
```

Furthermore, by changing the resource definitions for the loop workloads, we can quickly change which resource subsets of the loops will run on (here we bind all loops to the CPU):

```
#define work_intensive    bind_cpu
#define data_injtensive   bind_cpu
```

Using this general technique, applications can pick broad categories that characterize loops based on work intensity, data intensity, branching intensity, etc. and then implement ‘forall’ execution schemes that can most efficiently execute loop algorithms on underlying architectures. Tools such as ROSE could be used to automate some of this refactoring. This is the basis of the RAJA programming model developed at Lawrence Livermore National Laboratory[17]

RISK: Manual recoding adds defects to reliable codes

Tools for Automating Processes

Rank 2.3

The second area is in the use of tools for automating processes. As the workload on our existing staff increases, it's important to automate more of building/testing process. We support a variety of platforms, and developers do not have the time to build and test for all of them. That's where continuous integration tools like Jenkins or Atlassian's Bamboo can help. They can monitor a software repository, and after new code is committed, they can get a copy of the new code, build it for all the platforms/compilers of interest, and run the unit, regression, nightly, etc tests on those platforms. They then present the results of all of this in a concise manner with the ability to get more detailed information, if desired. They can also notify a developer that his/her commit has broken the code on one or more platforms. There needs to be institutional support for at least one of these tools. There may also be a need for increased computational resources for developers to allow these builds/tests to be done in a timely manner.[18]

RISK: Not testing all platform types supported

Simulation Ensembles:

Rank 2.2

The unit of simulation is the ensemble study, not the single run. We need to invest more in tools to support ensembles of simulations. Most of our software engineering work in support of simulation goes into codes and tools that operate at the level of *individual simulation runs*, i.e. languages, libraries, build tools, MPI, OpenMP, debuggers, revision control systems, performance instrumentation and analysis tools, load balancing schemes, etc. But no one ever runs a simulation just once. In any realistic study a simulation code has to be run hundreds to millions of time, over a multidimensional space of options, parameter values, and random seeds. Such a collection of related runs of the same code is an *ensemble study*, and it is fair to say that *the ensemble study is the primary unit of simulation*, not the individual run.

Managing an ensemble of simulation runs involves a lot of complexity:

1. Hundreds or thousands or more of different config files and input files have to be prepared systematically.
2. Output data has to be organized into databases and/or a hierarchy of files and directories for offline processing and visualization.
3. Checkpoint/restart files have to be managed.
4. Performance data has to be saved and analyzed.
5. Failures of all kinds must be logged and analyzed to see if the failed runs should just be repeated (perhaps with a larger timeout) or if a bug has to be fixed first.

Ensemble studies fall into distinct patterns:

1. Logic testing ensembles that are designed to exercise the code on data whose output is analytically tractable, to test either correctness, numerical accuracy, statistical distributions, etc.
2. Performance testing ensembles for doing scaling studies, or memory management studies, or load balancing studies, etc.
3. Parametric ensembles for doing sensitivity studies
4. Optimization studies designed to explore a parameter space to find optimal parameter values

that maximize some objective function. The optimization can be by hill climbing, simulated annealing, genetic algorithm, etc.
5. Monte Carlo ensembles designed to measure parameters of various output distributions, measure correlations, determine the frequency of rare events, etc.
6. Uncertainty quantification studies designed to measure and apportion sources of uncertainty.

These patterns involve different ensemble control logic, run into different resource limitations, and involve different strategies for deciding what simulations to execute, and in what order. The outputs of ensemble studies have to be fed into other tools, e.g.

1. Data analysis tools
2. Visualization tools
3. The ensemble management tool itself, which may use results of early runs in the ensemble to decide what runs to schedule later in the same ensemble, with what command line inputs, configurations, model parameters, and random seeds to give them.

Managing ensembles of simulation runs is usually done essentially offline (i.e. not on the computer running the simulations), usually via simple scripts requiring a lot of detailed time, attention, and guidance from human analysts. Occasionally more automated tools are used, such as DAKOTA. Dakota is mature and excellent as far as it goes, but it generally does not run on the same platform as the simulations it launches. Instead, it submits simulation runs to the batch queue of another machine, and the batch queue scheduling policies and delays can make the full ensemble study far slower than necessary.

What are needed are ensemble management tools that allow ensemble management to be programmed and to run *concurrently*, *on the same platform* as the simulation runs. It must be able to *launch independent parallel simulations executions asynchronously*, *within the same job* as the ensemble management tool itself, and must be able can *recover from failures of some constituent simulations without the entire job being abnormally terminated*. Such a tool needs operating system or runtime system support, and needs to be *standardized*, *documented*, and *portable*.[19]

RISK: Not having sufficient simulation result management tools

Thread Rescoping Tool

Rank 1.9

Most large parallel scientific software applications are implemented using MPI. In MPI-parallelism, each MPI process is executed in serial with messages sent between serial processors to achieve parallelism. In order for these programs to exploit next-generation programming models, the first step required is to convert the serial code sections to threaded code sections. The first step to making code thread safe is to rescope variable declarations so that each variable is owned completely by a local thread instance. This can be a monumental, exhaustive, and error-prone task to do by hand for legacy codes that have nearly one million lines of source code. The ROSE team at LLNL has recently written a thread rescoping tool that automates this process, reducing the largest burden codes have in making their codes thread safe. These recent improvements to ROSE have been tested in two large ASC production codes, and the tool has eliminated a line-by-line examination of the code, specifically eliminating a decision-point for each and every line of code whether it is thread-safe or not.[20]

RISK: Manual threading process degrades code reliability through typographical errors and not recognizing thread-unsafe code

Fine-grained parallelism challenges

Rank 1.9

Presently, numerical kernels in most LLNL ASC codes are usually serial and operate on data associated with an entire domain. However, efficient parallelism is tied closely to memory-locality. One way to improve locality in a multithreaded environment is to use many small domains, allowing more threads to simultaneously share data caches without contention. Unfortunately, domain overhead measured in terms of additional memory needed for non-shared data, and domain management operations that are hard to amortize away, can lead to space or performance problems on current multicore systems. Thus, an alternative to traditional domain partitioning will be required to exploit massive on-processor parallelism.

A better option is to employ fine-grained data “chunking” within a domain where a chunk of data can be assigned to a work thread or passed to an algorithm kernel. Proper chunk size selection can balance both instruction and data cache usage so that neither cache becomes overly strained. For example, if an algorithm works on a single element at a time (typical for a complex material model), the amount of code executed may not fit into an instruction cache. So the *algorithm* is always streamed from main memory (as though there is no instruction cache), while the data may be perfectly cached, with room to spare. On the other hand, if the chunk size is larger than will fit into the highest level of processor data cache, then the *data* is always streamed from main memory (as though there is no data cache).

Careful ordering of array accesses is also important to improve cache reuse, which is critical for good performance; e.g., ensuring that all entries in a cache line are used before the cache line must be reloaded. In a multi-material hydrodynamics code, a material model may likely be the primary work unit on a domain. Ordering elements so that data for elements with the same material are adjacent in memory can provide an optimal cache mapping. When materials move between mesh elements due to advection, it may be wise to periodically permute mesh data to retain memory adjacency. Optimal cache reuse will likely occur when data layout mirrors the needs of dominant numerical operations. However, which particular loops dominate runtime for a code is often highly problem-dependent. Flexibility to permute data could save an application from using poor memory access patterns for a given architecture. Reordering can also enable "lock-free" computations in a multithreaded environment. Using traditional programming language constructs, such as C-style for-loops, all execution and data access details are hard coded in the application source code. Without some sort of abstraction layer, such as RAJA, altering implementation details is difficult and may require creating *and maintaining* multiple versions of individual loops.[21]

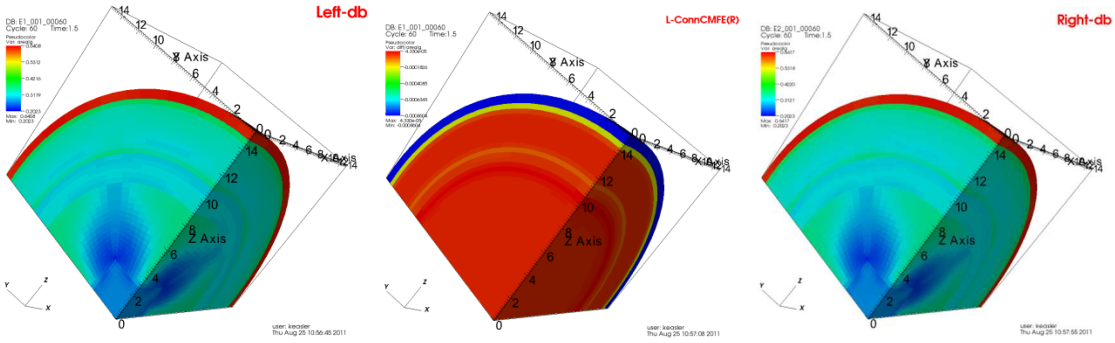
RISK: Code modification for porting causes decrease in code reliability

Visual Debugging

Rank 2.0

As we move to next-gen hardware, we expect the complexity of meshes used for simulations to grow dramatically. Parallel meshes will routinely contain terabytes of information, and some runs may even contain a petabyte of information. In this environment, how can we expect to quickly find errors

introduced into computations? One potential way to reduce time in finding errors is to introduce a visual debugging capability.



In the figure above, the visualization on the far left and far right show the "characteristic length" field of a physics simulation. The value on the far left is the "correct" answer, and the value on the right is the answer after a bug is introduced into a simulation. Looking at these arrays visually, there does not appear to be a large difference. Picking through the actual array used to contain the data for each field, every element of array data appears to be slightly different in the far left and far right case. How would a person hope to debug such a case when there is terabytes or even a petabyte of data to comb though? By taking a difference of the characteristic length field from two runs at the same time step, we get the picture in the center. Here, we can see that although essentially every value is different, there is a "front" that is clearly visible in the data. To the trained eye, there is an almost immediate recognition that there must have been a slight variation in the simulation time generated for the run on the left vs the run on the right, turning a very complicated analysis problem into a problem that suggests a clear place to look in the algorithm for the introduced error. Using the C++View capability in the TotalView debugger, and combining it with the VisIt simulation software, we have prototyped a system where a baseline simulation can be compared against a modified simulation, running lockstep in the debugger, to visually look for anomalies in the differences between runs. The prototype is not yet in production due to some manpower issues, but we feel will be a profitable debugging tool, once in place.[22]

RISK: Not having a quick way to identify anomalies at scale decreases productivity

HPC Resources For Testing

Rank 2.7

To achieve high software quality, application teams must periodically test large-scale codes on large-scale problems because some software defects only manifest at large scale. Currently, DOE projects write requests for allocations on DOE supercomputers, and they may neglect or underestimate the time required for automated testing. In the presence of limited resources, projects may be inclined to sacrifice quality for the sake of getting results.

There are several frameworks for automated testing (for example Jenkins or Bamboo). It can be difficult to set up automated testing on HPC systems because normal users may not be able to set up cronjobs, or there may be firewall restrictions that make it difficult to "checkout"
the latest version of the code from the software repository or to report back to the centralize test server.[23]

RISK: Testing treated as an afterthought

Data:

Data

Rank 2.2

The three things that matter most in quality engineering of scientific computing software are Data, Data, Data. This is counter-intuitive. Historically, we think about scientific computing applications and Data as distinct, nearly entirely independent products. For software, we started with machine languages, then assembly, then 3rd and 4th generation high level languages and now we are even working on domain-specific and 5th generation languages. Our treatment of Data, however, has not fared nearly so well. We're still in the machine-language era with respect to our means for describing Data. However, the manner in which we describe data governs entirely the community's ability to independently develop software components that operate upon it. The best way to enable community development of modular, portable, shareable, quality software components that act in useful and sophisticated ways upon Data is to foster the adoption of community-wide standards for the flexible storage, description and exchange, both in-situ and by files, of scientific data.[24]

Libraries

The next three things that matter most in quality engineering of scientific computing software are Libraries, Libraries, Libraries. As scientific computing applications grow in complexity, more and more functionality is packaged in independently developed libraries. Worse, as the computing environments in which these applications run grow in complexity, it gets easier to make mistakes in building, installing and using libraries as well as the applications that depend on them. Unfortunately, SQA standards so far developed focus primarily on applications, not libraries. SQA standards for libraries differ from applications in many respects. Libraries for next generation computing must be developed with a multitude of SQE practices aimed at minimizing the likelihood of making mistakes in using them and at maximizing users' ability to diagnose and correct them when they occur. Libraries must be "smart" enough to auto-detect appropriate defaults, enable users to access performance metrics (much like CPU hardware counters) and even inform users of poor performing configurations and use.[25]

Operating Systems

The last three things that matter most in quality engineering of scientific computing software are Windows, OS X , and Linux. Yes folks, believe it or not, a lot of scientific computing software that scales to 10^6 cores is developed on puny Windows and OS X laptops, even tablets using those platform's *native* development environments. Sure, we might *run* only small 4 and 8 processor runs on these platforms. But, the ability to *develop* next generation software on these platforms is invaluable for the community at large, particularly Open Source packages where users the world over may download, modify and submit patches to fix bugs and add features. This is possible only when applications *and* all of their support libraries are designed to support native development on these platforms. Towards this end, CMake (as opposed to Autoconf, Scons, etc.) and CMake-ified packages are a good step

forward. Maintaining proficiency in all three of these operating systems is essential for the success of next generation software development efforts.[26]

RISK: Development environment overly specialized or expensive

Data and algorithm organization in physics codes Rank 2.3

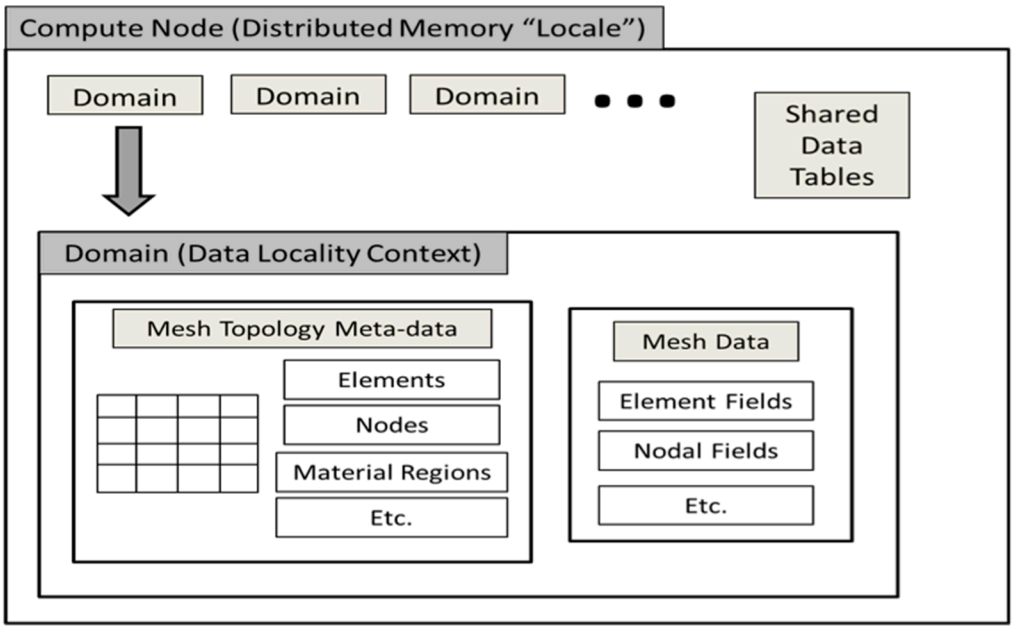


Figure 1. Basic organization of a typical domain structure in a mesh-based physics code.

A typical ASC code has clearly-defined mesh and data abstractions. Generally, a problem is decomposed and distributed across compute nodes on a partition of a parallel HPC platform. Each compute node is considered a distributed memory "locale" to which some number of MPI processes is assigned. A data structure, often called a "domain", owns a description of part of the mesh and the field data for that mesh part. Each domain is owned by exactly one MPI process and each process may own multiple domains. The basic elements of a domain structure are illustrated in Figure 1. Data on a domain is disjoint from data on other domains, and often maps to a "coherence domain" for caching purposes. A domain is also a "locality context", typically representing the finest level of data partitioning in a code. Each mesh field in a domain is associated with a fixed centering on the mesh, such as finite-element or vertex centered, and the data for each field is held in a distinct array. Field arrays are 1-dimensional computer science structures regardless of the underlying problem dimension. Fields are typically registered in a centralized data store, making it easy to map Fields to the peculiarities of the underlying memory subsystem from a single source code location. Also, there usually exists other metadata on a domain, for example to map materials to elements, as well as non-mesh data, such as tables of physical data (e.g., material properties, equation of state, etc.) shared by domains on a compute node.
The mesh topology defines the organization of elements and vertices on the mesh. Generally, there are two fundamental mesh configurations, structured and unstructured. A *structured* mesh uses an N-dimensional Cartesian index space, which defines uniform element-to-vertex connectivity and a single

element geometry. Structured mesh algorithms often use nested loops to traverse logically rectangular regions on a mesh. Such operations rely on zero-overhead implicit relationships between mesh entities, and allow a high level of compile-time optimization due to stride-1 data access patterns. An *unstructured* mesh is composed of arbitrarily connected vertex points that define the elements they surround; thus, an unstructured mesh admits arbitrary element geometries. Due to the irregular connectivity, relationships between unstructured mesh entities are defined using lists of array indices. For example, eight vertices define each element on a three-dimensional hexahedral mesh, so eight nodal-array indices are stored to access nodal field data for each element. Use of indirection arrays to manage relationships among mesh entities requires additional memory traffic, involves a much higher ratio of integer to floating point operations, and precludes many compiler optimizations. Regardless of the underlying mesh topology, most ASC physics codes employ algorithms involving regular, stride-1 memory accesses as well as those requiring indirection arrays. So, efficient implementations of both types of operations are important to every code.

Mesh data is often organized into a *hierarchy of contexts*, typically, where a context represents a relationship between the mesh and data on the mesh. There will be multiple *topological* contexts, one for vertex-centered quantities, one for element-centered quantities, face-centered quantities, etc. An element context will have child contexts that each enumerate the elements associated with a given material region. Often, material region contexts are further partitioned into clean elements (single material) and "mixed" elements (containing multiple materials). When contexts are nested, local indices are typically used within a child context to index into arrays associated with a parent context.

The context hierarchy in an ASC code is designed to map the conceptual organization of physics operations to the underlying data structures and memory subsystem. Most physics operations are encoded in loops; a large code will have tens of thousands of loops, typically. However, within a given code, there are relative few *loop patterns*. Common loop patterns involve:

- Simple traversal within a context (e.g., loop over all elements, vertices, etc.)
- "Parent-child" interactions within a topological context (e.g., loop over all elements containing material "A" and update values for some field defined over all elements)
- Relations between fields in different topological contexts (e.g., difference stencils involving vertex- and element-centered quantities)

Other operations may involve more elaborate data dependencies, but are less common.[27]

RISK: Architecture sprawls and is not optimized

---

[1] Tom McAbee LLNL

[2] Tom McAbee LLNL, Greg Pope LLNL, Ellen Hill LLNL, Stephanie Dempsey LLNL

[3] Burl Hall, LLNL, Derek Gaston, INL

[4] Patty Loo INL, Derek Gaston, INL

[5] Andy Salinger, SNL

[6] Rich Hornung LLNL

[7] Jeffery Carver, University of Alabama

[8] Rob Neely LLNL, Ellen Hill LLNL, Greg Pope LLNL

[9] Robert Blyth, DOE-ID

[10] David E. Bernholdt ORNL

[11] Roscoe A. Bartlett, ORNL, Michael Heroux, SNL, Jim Willenbring, SNL

[12] Tamara Dahlgren LLNL
[13] Greg Pope LLNL, Ellen Hill LLNL
[14] Burl Hall, LLNL
[15] Greg Pope LLNL, Stephanie Dempsey LLNL
[16] Greg Pope LLNL
[17] Jeff Keasler LLNL
[18] Burl Hall, LLNL, Derek Gaston, INL
[19] David Jefferson LLNL, Stephanie Dempsey LLNL
[20] Jeff Keasler LLNL
[21] Jeff Keasler LLNL
[22] Jeff Keasler LLNL
[23] Tom Epperly, LLNL
[24] Mark Miller LLNL
[25] Mark Miller LLNL
[26] Mark Miller LLNL
[27] Jeff Keasler LLNL