LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-781020

Diablo Test Suite: Code Coverage Analysis

R. K. Ganeriwala

July 11, 2019

Disclaimer

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

Lawrence Livermore National Laboratory

Diablo Test Suite: Code Coverage Analysis

Rishi Ganeriwala

July 3, 2019

Disclaimer

Table of Contents

Executive Summary

An analysis of the Diablo source code covered by its Software Quality Assurance test suite was performed. Intel's code coverage tool was used to determine the percentage of covered files, blocks, and functions. Excluding non-functional legacy files and single use developer-only files, the code coverage tool showed that 55% of blocks, the most applicable metric and closest to lines of code, are covered. For a large software code, 70-80% coverage is generally considered a reasonable target.

A significant percentage of the uncovered code can be categorized as features that are not practical to test, duplicate/unused legacy code, and niche features only used by select individuals. Additionally, large segments of code dedicated to debugging and error handling are untested, as the test problems do not error out. The remaining untested code can be considered general purpose for use by the analyst/user community. These untested general purpose features are identified in the document and should be prioritized when new testing is added. Suggestions for additional test problems that would have the biggest impact on increasing the code coverage metric are presented.

Finally, it is important to note that the single biggest way to increase the overall coverage metric is to delete old and unused code. While admittedly tedious to perform, future code development would also benefit markedly from this. Deleting unused code would likely push the coverage metric north of 60%, which is not too far off the 70-80% target. Overall, the current test suite covers the majority of commonly used features in Diablo, but there remains significant room for improvement. These areas for improvement have been identified.

Introduction

For Software Quality Assurance, it is important to test all aspects of an existing piece of software. These tests should be designed to run quickly and in parallel. Ideally these tests would cover 100% of the code to ensure every line was functioning as desired. However, this is often not practically feasible. Instead, code coverage of 70-80% is a reasonable goal for most system scale test suites [1]. Code coverage above this percentage offers diminishing returns and can hinder development if too much effort is being put into designing tests for every niche scenario. Additionally, it is important to realize that even code coverage of 100% does not mean the software is bug free, as sometimes a particular combination of features may cause an issue even if all those features function fine individually. Thus, the 70-80% metric is a reasonable goal to set for a relatively large finite element code such as Diablo [2].

This document describes the analysis performed to determine how much of the existing Diablo finite element code is covered by its test suite. First, a description of the methodology used to perform this analysis using Intel's Code Coverage Tool [3] is presented. After compiling Diablo with code coverage options enabled and running the test suite using this special executable, an overview of the code covered by the entire test suite is created. File-by-file analysis of the code coverage is performed and areas where the Diablo test suite is lacking are identified. Note that the Intel tool provides information at the block, function, and file level. A block is defined as a loop or logical statement within the code, i.e. an "if" statement or "do" loop. A function is a self-encompassing set of commands, i.e. a Fortran subroutine or function. A file is just that, an individual file within the code.

This purpose of this document is to:

1. Describe the methodology used to perform the code coverage analysis,
2. Perform an in-depth analysis and summary to identify areas that are not adequately tested within the existing test suite,
3. Recommend additional tests that will have the biggest impact on improving the current code coverage,
4. Place a historical time stamp on the coverage statistics of the Diablo code as of the date of this document.

Ideally the overall coverage of Diablo will increase as time goes on. Areas identified as not covered by the test suite because they are no longer used may be stripped out of the code in the future. Test problems that can have the most impact on increasing the overall code coverage percentage should be prioritized.

Methodology

The Intel Code Coverage Tool was used to identify how much of the Diablo source code is covered by the existing Diablo developers' test suite. This tool uses the profile guided optimization compilation options provided by the ifort (Fortran) and icc (C) Intel compilers. Use of this tool requires adding some compile flags when building the Diablo executable. While the vast majority of Diablo is written in Fortran, some high level files are written in C. In fact the main driver function is in C, which then calls Fortran. For this reason, the necessary compile

flags were added to both the C and Fortran compiler specifications. As Diablo uses Python's SCons format to build the executable, the following flags were added when compiling with code coverage enabled:

Fortran:
-prof-gen=srcpos -prof-dir=<build_directory>

C:
-prof-gen:srcpos -prof-dir=<build_directory>

Note the slight syntax difference in the Fortran vs. C flags. The "-prof-gen" flag creates the profile guided optimization data which is used to determine code coverage. The "-prof-dir" flag is used to specify the directory where all the profiling information is output to when the executable is run. This is explicitly specified to be the build directory so it is easy to combine them all later.

During compilation, a static profile information file "pgopt.spi" is created in the build directory. Next we execute a script that runs the entire test suite. After execution of each test problem, a dynamic profile information file, "*.dyn", is created. Two "*.dyn" files are created for each test problem since Diablo first partitions the problem, and then runs the simulation, requiring two separate executable calls. A third one will be generated for restart tests.

Running the test suite with these extra compilation flags slows down execution, but not overly so. The run-time increases by a factor of roughly 1.5x across the entire test suite, though this varies from problem to problem. After all the tests have completed, the test directory is now populated with hundreds of "*.dyn" files. These are combined into a single dynamic profile information file using the "profmerge" command. This command automatically searches the current working directory for all "*.dyn" files and combines them into a single file with the default name "pgopt.dpi", which contains all the dynamic profile information.

Finally, we run the code coverage tool using the command "codecov". This will automatically search the current working directory for the static profile information, "pgopt.spi", and dynamic profile information, "pgopt.dpi", files. Using this information it will output HTML files containing statistics and details about the code covered by the problems which were run. Included in these files are both (1) high level details regarding the overall coverage statistics of blocks, functions, and files in the entire code; and (2) information regarding the blocks and functions covered within each file. The user can click on an individual file to see a line by line coverage analysis, where uncovered sections of code are highlighted. The next section details specifics about the code coverage in Diablo using the existing test suite.

Remark: Certain files are not expected to be tested, but are still present in the code due to legacy reasons. These can be excluded via the use of the "-comp" flag when executing the "codecov" command. This requires the creation of a text file identifying which files to include/exclude. An example of this file, along with a summarized set of commands to run for Diablo developers, is included in Appendix A.

Results from Code Coverage Analysis

Code coverage statistics

Table 1 summarizes the code coverage statistics for the entire Diablo source code. However, the results in Table 1 include files in Diablo that are legacy and completely unused or unfunctional. Thus, for the purpose of identifying a more applicable current code coverage metric, a second coverage analysis was performed. In this second analysis (summarized in Table 2) any files solely dedicated to the following categories were excluded, as they are not currently used and there are no near-term plans to begin re-using these features:

- Electrostatics,
- Magnetics,
- Thermal ablation (includes smooth analysis/mesh smoothing),
- Corrosion and surface kinetics Neumann BCs for thermal and advection/diffusion.

A couple other features are currently unused but may eventually be re-enabled in the future. However, as these features are not considered functional at present, they were also excluded from the second analysis:

- Residual based error estimation for adaptive mesh refinement (AMR),
- AMR in conjunction with contact.

Finally, a few special purpose features only used by select developers and not intended for use by the general analyst community are also excluded from the second analysis:

- MatPro material models for Nuclear Energy Advanced Modeling and Simulation (NEAMS) project,
- ITAPS code coupling for NEAMS,
- Lysmer viscous boundary condition for earthquake simulations,
- Bielak stress calculation routines for earthquake simulations,
- Exascale computing project (ECP) related code coupling.

Excluding files associated with the above categories, the more applicable code coverage statistics are provided in Table 2. 85% of files within Diablo, 66% of functions/subroutines, and 55% of blocks are at least partially covered by the test suite.

Remark: This number is still skewed to be lower than the actual code coverage since there was no easy way to exclude blocks or functions dedicated to the above features within larger files. Rather, it was only possible to exclude files solely dedicated to the above features by searching over keywords in the filename. The excluded filename keywords are provided in Appendix A.

Table 1: Coverage statistics for entire Diablo source code

Files				**Functions**				**Blocks**			
Total	Cvrd	Uncvrd	Cvrg%	Total	Cvrd	Uncvrd	Cvrg%	Total	Cvrd	Uncvrd	Cvrg%
628	453	175	**72.13**	4,928	2,860	2,068	**58.04**	269,000	137,776	131,224	**51.22**

Table 2: Coverage statistics for reduced Diablo source code (excludes unused and single purpose developer-only files)

Files				Functions				Blocks			
Total	Cvrd	Uncvrd	Cvrg%	Total	Cvrd	Uncvrd	Cvrg%	Total	Cvrd	Uncvrd	Cvrg%
534	452	82	**84.64**	4,303	2,858	1,445	**66.42**	249,464	137,732	111,732	**55.21**

Regarding the remaining uncovered functions and blocks within the source code, we can categorize these broadly into three categories:

1. Features not expected to be covered,
2. Features that have niche use,
3. Features that may used by the general analyst community.

We will now go into more depth regarding each of these three categories.

Untested features not expected to be covered

A significant percentage of the remaining uncovered files, functions, and blocks relate to various features we would not expect to be covered by the test suite, namely:

1. Blocks that generate error messages,
2. Situations that the code should not be able to enter (barring a memory leak),
3. Duplicate routines,
4. Types that merely define variables/parameters,
5. Old/unused NIKE functions and files,
6. Legacy code no longer used,
7. Debug print code,
8. Library interface routines which are defined in Diablo, but never actually called.

It is not reasonable to test for every bit of error checking placed in the code, and it is not possible to test for error trapping put in place to ensure the code doesn't go into places that are not allowed (barring a memory leak). Additionally, in some files multiple subroutines were created which have the same functionality. However, the old subroutine was left in place and never deleted, presumably as a back-up in case there was a bug with the new subroutine. The largest incidence of this bit of "double code" has to do with memory destructor routines. In older sections of code, there is often one destructor routine that was manually put in place by the developer of that file, and a second that was later automatically generated by running the "DIABLO-RESTART" script. Obviously, only one of these two memory destructor routines will ever be called in those files where two are present.

The fourth category the code coverage tool flags as uncovered are user-defined Fortran type definitions which simply define a number of variables or parameters. As these bits of code never actually get executed within a block (rather they act as always being defined), the code coverage tool sees them as being uncovered. The fifth and sixth categories relate to unused functions and files never actually called within Diablo. This includes a number of interpolation routines and some stress material models ported over from NIKE, but never used as equivalent ones were created in Diablo. Additionally, there are a number of write subroutines in material models that never get called from within the code (presumably these are legacy of an old/outdated method of

handling restart or code coupling). These legacy write subroutines are still present in nearly every material model and contain a large number of unused blocks.

The seventh category is debug print code, which is placed in individual subroutines by developers when trying to debug a feature, but never intended to be seen by the general user community once that feature has been correctly implemented. The final category of untested features that we wouldn't expect to be covered relates to external libraries that Diablo links to (e.g. linear solvers, exodus reader, MPI, Lua, etc.). Interface functions are created within Diablo to map to the equivalent command in the library. However, some of these commands are advanced features of the library never actually called within Diablo, even though the interface function is defined.

Untested features that have niche use

Another category of features not covered by our current test suite may be classified as features which have niche use. Some examples of this include:

1. Error estimation routines in material models not currently functional with AMR,
2. METIS advanced partitioning options,
3. Exodus mesh advanced read-in options,
4. HYPRE, WSMP, and PASTIX advanced solver options,
5. AM specific features,
6. Eulerian body load (EBL) options,
7. Lua functions for PID feedback control,
8. AMR by geometric refinement or material indicator,
9. Advection/diffusion related features (currently under active development/modification by S. Castonguay),
10. LIS iterative linear solver (not commonly used).

As the above features are currently only used by developers and/or select other analysts, there is less priority on increasing code coverage for them. This is not meant to justify these features remaining untested, but rather is simply meant to state that increased testing of these features would not have as much benefit on ensuring proper Software Quality Assurance for the general analyst/user community.

Untested features that may be used by the general analyst community

The remainder of untested code in Diablo is applicable to the general analyst/user community; as such these features should be assigned the highest priority for increased testing. The following features in Diablo are either completely untested or only partially tested by the current test suite.

Outline 1: Untested features in use by the general analyst community

1. Material models
    a. Stress materials
        i. Stress 02 – orthotropic elastic
        ii. Stress 08 – thermo-elastic-creep
        iii. Stress 09 – power law plasticity
        iv. Stress 11 – transient thermal creep
        v. Stress 12 – Ramberg-Osgood elastic-plastic

vi. Nonlinear hardening options of Stress 27 (hyperelastic-plastic)
vii. Stress 63 – Ogden visco-hyperelastic
viii. Various viscosity options of Stress 67 (hyperelastic elastomeric foam with viscoelasticity)
ix. Stress 70 – elastomeric foam

b. Thermal materials
i. Various options of Thermal 02 (orthotropic conduction) and Thermal 04 (temperature dependent orthotropic)
(1) User frame of reference for specifying axes
(2) Temperature functions via Lua

2. Boundary conditions
a. Thermal Neumann BCs
i. Bulk node enclosure
ii. Sublimation
b. Stress point BCs
c. Constraint BCs
3. Overlink Silo database writing for code coupling
4. Select Lua options
5. Certain HYPRE solvers and precoditioners
a. Solvers not tested: BoomerAMG, LGMRES
b. Preconditioners not tested: PILUT, EUCLID
6. Contact
a. Thermal contact
i. Augmented Lagrange
ii. Many Thermal 01 Contact material properties
iii. Node on segment thermal contact
b. Various mortar contact options
7. Element types
a. Beams
i. Stiffness and/or mass damping
ii. Various beam parameters
iii. Thermal (Topaz) beams
b. Shells
i. Thermal shells
ii. Stress 03 shells
iii. Stress 04 shells
iv. Various shell parameters
c. Hex elements with single point Gaussian integration
d. Degenerate tet elements
8. NEFC and element print block options
9. Basemotion rotation options
10. Diablo-Paradyn coupling
11. Initial acceleration for thermal mechanics
12. AMR with restart (many test problems in AMR suite don't include restart)
13. Ragged (non-continuous) element and NEFC labeling

Conclusions and Recommendations

Intel's code coverage tool was used to determine the percentage of covered files, blocks, and functions in the Diablo source code by the developers' Software Quality Assurance test suite. Excluding non-functional legacy files and single use developer-only files, the code coverage tool showed that 55% of blocks are covered. For thorough system scale code coverage of a large FEA code such as Diablo, 70-80% coverage is seen as reasonable target.

However, even this 55% coverage metric is a bit misleading, as this includes many features that are not practical to test and niche features only used by select individuals/developers. It also includes a significant amount of duplicate and old/unused code that was not easily possible to remove from the analysis. While it is difficult to quantify the effect of all these areas on the exact code coverage metric, it is reasonable to assume that the overall coverage would increase to slightly above 60% of blocks if all of these above mentioned areas could be excluded. This is not too far away from the 70-80% target, and may be considered acceptable according to some sources [4].

On the other hand, there remain a number of general purpose features which do not have adequate coverage in the test suite. The highest priority for increasing code coverage should be assigned to these features, presented in Outline 1. A few suggestions for ways to most increase code coverage with the least effort are to either add or modify test problems to test for the following features (in approximate order):

1. Hex 8 elements with single point integration,
2. LIS linear solver,
3. Overlink,
4. Contact (especially various mortar and thermal options),
5. Stress material models (often completely untested, e.g. stress 70 has over 1000 uncovered blocks),
6. Shell elements for thermal, stress 03, and stress 04

Finally, it is important to mention that the single biggest way to increase the overall coverage metric is to delete old and unused code. For instance, the files "restart_IO_HDF.F90" and "restart_IO_Interface.F90" have over 10,000 (!!) blocks combined, but appear to have been completely replaced by "restart_IO_HDF_v2.F90". Another example is the file "pad_init_acc.F90" which has been completely replaced by new subroutines located in "diablo_init.F90". Since Diablo uses Git to track code changes, such legacy files and functions can safely be deleted as the complete version history is saved. Doing this would require some tedious editing throughout Diablo, but would significantly clean up the code and make future development easier. Obviously, this would have the added benefit of making it easier to identify the true code coverage metric for the Diablo test suite. Overall, it appears that the majority of commonly used features within Diablo have adequate coverage by the existing test suite. However, significant room for improvement remains.

References

[1] Cornett, Steve. *Minimum Acceptable Code Coverage*. Bullseye Testing Technology, www.bullseye.com/minimum.html. Accessed 16 June 2019.

[2] Solberg, J. M., Hodge, N. E., Puso, M. A., Castonguay, S. T., Ganeriwala, R. K., and Ferencz, R. M. *Diablo: A Parallel, Implicit Multi-physics Finite Element Code for Engineering Analysis User Manual*, LLNL-SM-757180, Lawrence Livermore National Laboratory, 2018.

[3] "Code Coverage Tool." *Intel® Fortran Compiler 19.0 Developer Guide and Reference*, Intel, 29 Apr. 2019, software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-code-coverage-tool. Accessed 16 June 2019.

[4] Oloso, Hamid, and Kouatchou, Jules. "Code Coverage Tools." *Modeling Guru*, NASA, 5 Mar. 2010, modelingguru.nasa.gov/docs/DOC-1828. Accessed 16 June 2019.

Appendix A: Developer Instructions for Determining Code Coverage

Steps for running code coverage analysis

The following steps summarize the commands a developer should use to perform a code coverage analysis of the Diablo test suite.

1. Run "Buildcodecov.sh" in the Build directory to compile with code coverage options enabled.
2. Run full test suite using "multi_test_driver.sh" in the "*_test_codecov" directory that is created.
3. Execute the command "profmerge" in your terminal window from within the "*_test_codecov" directory.
4. Execute the command "codecov -comp MyExcludeFiles.txt" in your terminal window from that same directory.
5. Open up the file "CODE_COVERAGE.HTML" in Firefox.

Exclude file keywords

The following keywords were added to a separate document called "MyExcludeFiles.txt" when running the code coverage analysis to generate the results shown in Table 2 of this report. The contents of this file are copied below. Note that the keywords preceded by a "~" are those that get excluded if contained within a filename.

Contents of "MyExcludeFiles.txt":

.f
.F
.f90
.F90
.c
.C
~ElecStat
~elecstat
~Magnetic
~magnetic
~Smooth
~smooth
~thermal_HW
~ablat
~matpro
~MATPRO
~ITAPS
~itaps
~lysmer
~Lysmer
~Bielak
~bielak

~_rb
~_RB
~ecp
~ECP
~Corro
~corro
~Kinet
~kinet
~EdgeOn