Technical Report UD-CIS-2011/02 January 21, 2011

A Functional Equivalence Verification Suite for High-Performance Scientific Computing

Stephen F. Siegel and Timothy K. Zirkel
Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716, USA

A Functional Equivalence Verification Suite for High-Performance Scientific Computing

Stephen F. Siegel and Timothy K. Zirkel

Abstract. Scientific computing poses many challenges to formal verification, including the facts that typical programs (1) are numerically-intensive, (2) are highly-optimized (often by hand), and (3) often employ parallelism in complex ways. Another challenge is specifying correctness. One approach is to provide a very simple, sequential version of an algorithm together with the optimized (possibly parallel) version. The goal is to show the two versions are functionally equivalent, or provide useful feedback when they are not. We present a new verification suite consisting of pairs of programs of this form. The suite can be used to evaluate and compare tools that verify functional equivalence. The programs are all in C and the parallel versions use the Message Passing Interface. They are simpler than codes used in practice, but are representative of real coding patterns (e.g., manager-worker parallelism, loop tiling) and present realistic challenges to current verification tools. The suite includes solvers for the 1-d and 2-d diffusion equations, Jacobi iteration schemes, Gaussian elimination, and N-body simulation.

1. Introduction

Computational techniques now play an integral role in virtually every scientific and engineering endeavor. This pervasive use of computing brings with it the challenges of creating accurate and stable software [13, 20]. Les Hatton's survey of computational software packages found that defects are ubiquitous "irrespective of the existence of any quality system, level of criticality, or application area" [10]. It is worth noting that only sequential software packages were used in the study. Many modern computational codes utilize parallelism, e.g. via the Message Passing Interface (MPI) [12], the *de facto* parallelization standard for high-performance scientific computation. It is doubtful the situation has improved.

One promising approach to addressing these problems is formal software verification. A number of verification tools targeting or applicable to scientific and mathematical software have been developed [2–5, 7, 8, 11, 14, 16, 18, 19]. What is needed to move the field forward are standard example suites for the testing, evaluation, and comparison of such tools.

In this paper we present the initial release of such a verification suite. The suite is targeted at tools for verifying the functional equivalence (i.e., "input-output" equivalence) of two programs. To this end, the examples are partitioned into groups consisting of a single specification and one or more implementations. The specification is a simple (non-optimized) sequential encoding of an algorithm. The implementations are generally more complex, optimized in various ways, and most are parallel

This material is based upon work supported by the National Science Foundation under Grants CCF-0733035 and CCF-0953210, and by the University of Delaware Research Foundation.

programs using MPI. Some of the implementations are erroneous (not functionally equivalent to the specification) and an effective verification tool should find and report the errors. At least one implementation in each group is (we believe) correct. All the examples are in the C99 dialect of C and use only standard C libraries, though some also use the GD graphics library for producing animated GIF files.

While the focus is on functional equivalence, the suite could also be used for verification of deadlock-freedom, absence of array indexing and pointer manipulation errors, numerical accuracy, and so on. The suite may also be useful for test-case generation tools, and for evaluating other testing techniques. The programs involving MPI might be useful for testing future tools that use parameterized model checking to prove correctness for any number of processes (cf. [6]).

The programs comprising our suite are much simpler than those actually used in modern state-of-the-art scientific and engineering research. Nevertheless, they reflect many common patterns employed in high performance scientific computing. In addition, the programs are sufficiently complex to challenge existing verification tools. As the tools become more effective, we plan on expanding the suite in future releases.

The programs in this release are written by us, in some cases inspired from examples in texts, which we have noted. They are all licensed under the GNU Public License v3 and can be found at `http://vsl.cis.udel.edu/fevs`. We will gladly consider contributions from others for future releases.

2. Notions of Equivalence

Often times, even a very simple pair of programs are not functionally equivalent under the semantics of floating-point arithmetic, i.e., they may give different answers when run on an actual computer. This is because floating-point operations are not associative. Two programs that, for example, add floating-point numbers in different orders are likely to get different results. Hence they are only "real-equivalent," i.e., equivalent if the values and operations are understood as taking place in the mathematical real numbers. This is in fact the case with most parallel numerical programs (cf. [15]). Other cases may arise where not even real equivalence is expected. See the numerical integration program below.

A stronger notion of equivalence is *Herbrand equivalence*. Two numerical programs are Herbrand equivalent if they produce the same result regardless of how the floating-point operations are defined (i.e., these operations can be treated as uninterpreted functions). A few of the examples in the suite satisfy this stronger condition.

3. The Programs

The complete list of programs together with several statistics for each are given in Figure 1. The first program listed in each group is the specification, the others are implementations. Those whose name includes "bad" are known to contain (often subtle) errors.

The columns in the table give (1) the program name, (2) the number of lines of code (excluding comments), (3) the McCabe cyclomatic complexity, and (4) the set of MPI primitives used by the program, excluding the functions `MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, and `MPI_Comm_size`, which are used in all of the MPI-based programs.

3.1. Adder

The adder programs read numbers from a file and return their sum. The specification uses a `for` loop to add the numbers in the order in which they occur in the file. The implementation is parallel code that distributes a portion of the input to each process. Each process adds its portion of the input,

Filename	LoC	CC	MPI
adder_spec.c	14	3	
adder_par.c	40	6	S,R
adder_bad.c	40	6	S,R
mean_spec.c	18	3	
mean_impl.c	18	3	
mean_bad.c	18	3	
factorial_spec.c	11	3	
factorial_iter.c	9	2	
factorial_spec.c	11	3	
fib_spec.c	14	3	
fib_impl.c	14	3	
fib_bad.c	13	2	
matmat_spec.c	32	12	
matmat_vec.c	38	12	
matmat_mw.c	67	20	S,R,BC
matmat_tile.c	46	18	
matmat_bad.c	67	20	S,R,BC
diffusion1d_spec.c	149	32	
diffusion1d_par.c	253	57	S,R,SR,BC
diffusion1d_nb.c	263	63	S,R,IS,IR,WA
diffusion1d_bad.c	243	57	S,R,SR,BC
diffusion2d_spec.c	172	24	
diffusion2d_par.c	252	42	S,R,SR
diffusion2d_bad.c	172	24	
laplace_spec.c	72	14	
laplace_rowdist.c	114	30	S,R,SR,AR
laplace_bad.c	114	30	S,R,SR,AR
gausselim_spec.c	101	27	
gausselim_rowdist.c	156	38	S,R,AR
gausselim_bad.c	101	27	
nbody_seq.c	289	46	
nbody_par.c	525	73	S,R,BC,IS,AR,B
nbody_bad.c	525	73	S,R,BC,IS,AR,B
integrate_spec.c	37	6	
integrate_mw.c	105	18	S,R
integrate_nb.c	122	17	IS,IR,W,BC,WN,WA
integrate_bad.c	105	18	S,R

FIGURE 1. The programs. The first in each group is the specification. Second column gives lines of (non-comment) code. Third column is McCabe's cyclomatic complexity. Fourth column lists the MPI functions used: S = `MPI_Send`, R = `MPI_Recv`, SR = `MPI_Sendrecv`, IS = `MPI_Isend`, IR = `MPI_Irecv`, W = `MPI_Wait`, WA = `MPI_Waitall`, WN = `MPI_Waitany`, BC = `MPI_Bcast`, BA = `MPI_Barrier`, AR = `MPI_Allreduce`.

and the resulting partial sums are combined to give the full sum. A single "root" process is used to read the file and send blocks of numbers to the other processes, each of which performs its local computation and sends the result back to the root. The only MPI functions used are `MPI_Send` and `MPI_Recv`.

The programs are real-equivalent. In the erroneous implementation, the root process neglects to add its own contribution to the partial sum.

3.2. Mean

These programs read numbers from a file and compute their mean. The specification first computes the sum of the numbers, then divides by the number of items. In the implementation, a running mean is computed each time a number is read. The programs are expected to be real-equivalent. The erroneous implementation uses the wrong data type to store the sum. It stores the sum as an integer rather than a floating-point number, so the result is truncated at each step.

3.3. Factorial

The factorial program computes $n!$. The specification performs the computation using recursion, while the implementation uses iteration. The erroneous version is recursive, but instead of stopping the recursion at 0 it stops at 1. Except when computing $0!$, this will be correct. On $0!$, the program will exhibit infinite recursion.

3.4. Fibonacci Phi Approximation

These programs approximate the "golden ratio" $\phi = (1+\sqrt{5})/2$ as the ratio of successive Fibonacci numbers. Since these ratios are alternately higher or lower than the actual value of ϕ, the error in the approximation can be estimated as the difference between two successive ratios. The program returns the first approximation for ϕ with an error less than the given tolerance. In each iteration, the specification computes the approximation, then uses that to obtain the error estimate. The implementation re-arranges the computations in several ways (e.g., using in-place exchanges instead of temporary variables, computing the error estimate first and using that to obtain the approximation). The programs are real-equivalent. In the erroneous implementation, the error is not forced to be non-negative.

3.5. Matrix Multiplication

These programs read two matrices from files and output their product. The specification computes the product in the standard way, which involves a triply-nested loop.

Implementation `matmat_vec.c` simply factors out the vector-matrix multiplication routine into a separate function.

Loop tiling is a common technique to improve performance by keeping small "chunks" of data in cache. Implementation `matmat_tile.c` accepts an additional parameter `TILE_SIZE` and tiles the three loops into chunks of that size. (The final chunk in each loop may have a smaller size.)

The manager-worker pattern is commonly used in parallel computing to obtain automatic load-balancing. The idea is that a problem is broken down into small tasks. A "manager" process distributes one task to each "worker" process. The worker computes and sends its result back to the manager. The manager, upon obtaining a result from some worker, processes the result and sends a new task to that worker. In `matmat_mw.c`, a task is the computation of one row of the product matrix. This requires multiplying one row vector from the first matrix with the entire second input matrix. This example is based on [9, Sec. 3.6]. Manger-worker programs are notoriously difficult to verify because of the combinatorial blow-up in the number of possible orders in which workers can return their results to the master.

The code used to compute one row of the product matrix is packaged in a matrix-vector multiplication function. This function is exactly the same in `matmat_mw.c` and in the sequential version

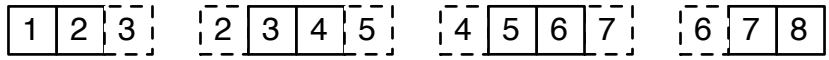

FIGURE 2. A typical distribution of points to 4 processes in 1-dimensional diffusion. The blocks with dotted lines represent ghost cells.

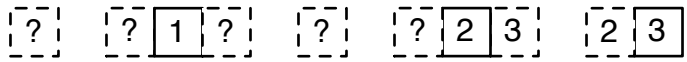

FIGURE 3. A diffusion configuration exhibiting the error in `diffusion1d_bad.c`. There are 5 processes, but only 3 cells. A process's immediate neighbor might not have any data.

`matmat_vec.c`. It provides an interesting opportunity to use abstraction: a tool for verifying functional equivalence of these two codes would not need to know what the matrix-vector multiplication function does, only that its output is a deterministic function of its input.

The erroneous implementation is based on `matmat_mw.c`, except that in the inner loop of the matrix-vector multiplication function, the output matrix is not initialized.

3.6. Diffusion1d

These programs simulate the evolution of the "diffusion" (or "heat") equation in one dimension. The initial temperature values and other parameters are read from a file. The two boundary values are kept constant. The program iterates for a given number of time steps, in each step, computing the new temperatures at each point in the domain. The output is in the form of an animated GIF file in which the color corresponds to temperature. The compile-time `DEBUG` option can also be used to send the output to a sequence of plain-text files.

Implementation `diffusion1d_par.c` is a parallel version in which the domain is block distributed: each process is assigned a (roughly) equal number of contiguous discrete points in the domain. Ghost cells are used to mirror the values of actual cells on the (at most) two neighboring processes. This is illustrated in Fig. 2. Communication is required at each time step to update ghost cells; this is accomplished using MPI's `MPI_Sendrecv` function.

Implementation `diffusion1d_nb.c` is another parallel version which uses MPI's *nonblocking* point-to-point function to improve performance. The nonblocking functions permit overlap of computation and communication. In this example, the ghost cell update is overlapped with the local computational update of the interior regions maintained by each process (cf. [17, Sec. 2.17]).

Implementation `diffusion1d_bad.c` is the same as `diffusion1d_par.c` but with a subtle bug introduced. In this version each process computes its left neighbor as process `rank-1` and its right neighbor as process `rank+1`. If, as in Fig. 3, the program is run with more processes than cells, some of the processes will not have any cells. The right and left neighbors should be the next and previous process that actually has cells. In the correct version, this is accomplished by using a function to compute the owner of the next or previous cell.

3.7. Diffusion2d

This is a two-dimensional version of the group above. In program `diffusion2d_par.c` the domain is distributed using a "checkerboard" pattern; this leads to a lower overall communication cost than a simple row-distribution scheme (as the total number of ghosts cells is reduced) but requires

a more complex communication pattern, as ghost cells must be updated in both dimensions. The erroneous version is the specification with an error in the update equation.

3.8. Laplace

These programs, based on the description of [1, Sec. 11.1.4], use a Jacobi iteration scheme for solving the 2-dimensional Laplace equation. These are similar to the row-distributed 2d diffusion solvers, but iterate until a convergence criterion is met instead of for a fixed number of time steps. Convergence is determined by measuring the L_2 norm between successive solutions and exiting when that falls below a specified tolerance. In the parallel version, this requires that each process compute its local contribution to the norm; the local contributions are summed using `MPI_Allreduce`. The erroneous implementation fails to make the neighbors of boundary cells `MPI_PROC_NULL`.

3.9. Gauss-Jordan Elimination

These programs read in a matrix of double precision floating point numbers, perform Gauss-Jordan elimination, and output the matrix in reduced row-echelon form. In the parallel version the matrix is row-distributed. Communication is required at several stages: finding the pivot row, exchanging the pivot row with the top row, and adding a suitable multiple of the pivot row to the other rows. In this case, the sequential and parallel versions are in fact Herbrand equivalent. The erroneous version is the specification but missing part of a loop update.

3.10. N-body Simulation

The 2-dimensional n-body simulator reads in the mass, initial position, and initial velocity of each body and simulates the motion of those bodies based on Newton's law of gravitation. This is an exact solution in the sense that for each body x, the force vectors resulting from each body y acting on x are summed to compute the total force on x. (More sophisticated schemes will approximate this computation to reduce the $O(n^2)$ complexity.) In the parallel version, each process is responsible for a fixed subset of the bodies; a mixture of non-blocking and blocking point-to-point communication is used to perform the force computation at each time step. The non-blocking communication prevents deadlock by allowing each process to send its vector of particle masses and positions to one neighboring process, and then receive from the other neighbor without waiting for the send to complete. The programs are real (but not Herbrand) equivalent. The erroneous program uses `dx` instead of `dy` in the y-direction acceleration computation.

3.11. Numerical Integration

These programs estimate the integral of a real-valued function f of one variable on a finite interval $[a, b]$ using the trapezoid rule with a 1-dimensional adaptive mesh refinement. See Fig. 4. An interval is cut in half until convergence between successive refinements is achieved. Refinement is non-uniform and unpredictable, so a manager-worker scheme is used in the parallel version. The programs should be real (but not Herbrand) equivalent. The erroneous implementation uses `MPI_ANY_SOURCE` rather than a variable for the PID in a `MPI_Recv`.

The integration examples expose a potential issue when checking functional equivalence. The most natural way to program a sequential algorithm is not necessarily functionally equivalent to the most natural way to program a parallel algorithm for the same problem. For sequential integration using the trapezoid rule, the natural approach is to start refining the entire domain. However, in the parallel version the domain must first be split and distributed across the processes. These splits might or might not line up with the divisions used for refining in the sequential version. Hence the specification and implementation would not actually be functionally equivalent, even though their results would be very close. The integration programs in FEVS work around this by having an additional input parameter. The domain is first split into a number of equal intervals based on the extra input. Refinement then proceeds as in the basic case. Since the intervals distributed to processes

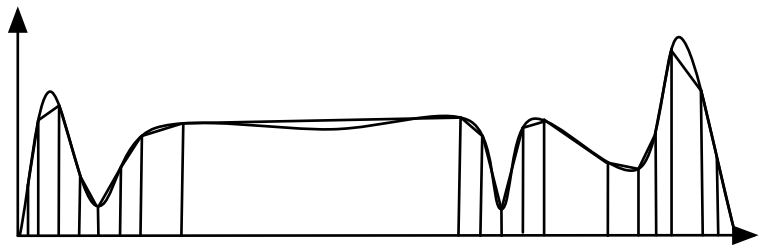

FIGURE 4. A trapezoid rule approximation of an integral using an adaptive mesh

in the parallel version are the same as the initial intervals for the sequential version, functional equivalence is preserved.

Verifying functional equivalence of the integration codes is very challenging. First, there is the explosion of the state space resulting from the use of `MPI_ANY_SOURCE`. Second, each recursive call might or might not meet the convergence condition. Finally, each recursive call includes a check that the tolerance does not fall below an ϵ based on the machine's minimum double.

It should be noted that the actual output of the integration programs will change depending on the value of the extra input parameter. Yet no matter the value, these programs are considered in some sense to be "correct." This suggests the need for specifications that are non-deterministic. A non-deterministic specification could define a family of possible executions. Any implementation could then be considered correct if it is equivalent to just one of the correct executions. Another example where this would be useful is a shortest path algorithm. An implementation should be correct if it defines *any* shortest path, so the specification must be able to define *all* possible shortest paths.

Another approach that might be useful for equivalence of numerical programs is to allow some small difference ϵ. In this case, the definition of functional equivalence would be relaxed to say that two programs are functionally equivalent if their output differs by no more than ϵ. For tools that perform analysis based on floating-point arithmetic rather than real arithmetic, this is probably the only viable notion of functional equivalence for most pairs of programs due to lack of associativity.

4. Future Work

FEVS provides a core set of test programs for functional equivalence verification tools. This set can be expanded to have broader applicability in a variety of ways. As tools improve in performance and accuracy, more complex examples and additional programs with known subtle errors will be added to the suite. Examples written in other languages, such as Fortran, will also be added.

The current version of FEVS is focused heavily on MPI. While MPI is the *de facto* standard, programs utilizing other parallelization implementations such as OpenMP, CUDA, or OpenCL would be welcome additions to FEVS. Diversifying in this way would allow FEVS to be used for performance comparison in addition to its primary use of evaluating tools for checking functional equivalence. For example, if a FEVS program has both MPI and CUDA versions, a scientist interested in performance on a particular hardware setup could run both versions (which are functionally equivalent) and compare the running times.

References

[1] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[2] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *LNCS*, pages 291–295. Springer-Verlag, 2005.

[3] Sylvie Boldo and Thi Minh Tuyen Nguyen. Hardware-independent proofs of numerical programs. In César Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium*, number NASA/CP-2010-216215 in NASA Conference Publication, pages 14–23, Washington D.C., USA, April 2010.

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[5] Patrick Cousot, Rashia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE Analyser. In Mooly Sagiv, editor, *Proc. European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 21–30, Edinburgh, April 2–10 2005. Springer.

[6] E. Allen Emerson and Vineet Kahlon. Parameterized model checking of ring-based message passing systems. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 325–339. Springer Berlin / Heidelberg, 2004.

[7] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV 2007*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[8] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. In *European Symposium on Programming, LNCS 2305*, pages 209–212. Springer Verlag, 2002.

[9] Willaim Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[10] Les Hatton. The T experiments: Errors in scientific software. *IEEE Computational Science & Engineering*, 4(2):27–38, April 1997.

[11] Bettina Krammer, Matthias S. Müller, and Michael M. Resch. MPI application development using the analysis tool MARMOT. In Marian Bubak, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Intl. Conference on Computational Science*, volume 3038 of *LNCS*, pages 464–471. Springer, 2004.

[12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, version 2.2, September 4, 2009. `http://www.mpi-forum.org/docs/`, 2009.

[13] Douglass E. Post and Lawrence G. Votta. Computational science demands a new paradigm. *Physics Today*, pages 35–41, January 2005.

[14] Stephen F. Siegel. Verifying parallel programs with MPI-SPIN. In Franck Cappello, Thomas Hérault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*, volume 4757 of *Lecture Notes in Computer Science*, pages 13–14. Springer, 2007.

[15] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM TOSEM*, 17(2):Article 10, 1–34, 2008.

[16] Stephen F. Siegel and Timothy K. Zirkel. Collective assertions. In Ranjit Jhala and David Schmidt, editors, *VMCAI 2011*, volume 6538 of *LNCS*, pages 387–402, 2011.

[17] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI—The Complete Reference, Volume 1: The MPI Core*. MIT Press, second edition, 1998.

[18] Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 285–286, 2008.

[19] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July , 2009, Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 599–613. Springer, 2009.

[20] Gregory Wilson. How do scientists really use computers? *American Scientist*, 97:8–10, September–October 2009.

Stephen F. Siegel
e-mail: siegel@cis.udel.edu
Verified Software Laboratory, Department of Computer & Information Sciences, University of Delaware, Newark DE 19716, USA

Timothy K. Zirkel
e-mail: zirkeltk@udel.edu
Verified Software Laboratory, Department of Computer & Information Sciences, University of Delaware, Newark DE 19716, USA