A Functional Equivalence Verification Suite for High-Performance Scientific Computing

Stephen F. Siegel, Timothy K. Zirkel, and Yi Wei *

Verified Software Laboratory, Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716, USA
{siegel|ywei|zirkeltk}@udel.edu http://vsl.cis.udel.edu

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. It is not suprising, then, that the computational science community has expressed increasing interest in formal software verification. A number of verification tools targeting or applicable to scientific software have been developed. 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 programs using the Message Passing Interface

* This material is based upon work supported by the National Science Foundation under Grant No. CCF-0733035.

(MPI), the de-facto parallelization standard for high-performance scientific computation. 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 text-case generation tools, and for evaluating other testing techniques.

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 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 (exluding comments), (3) the McCabe cyclomatic compexity, 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.

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, 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`.

Note that even in this simple example, the two programs are not functionally equivalent under the semantics of floating-point arithmetic, i.e., they may give different answers when run on an acutal computer. This is because floating-point addition is not associative, and the two programs add the numbers in different orders. Hence they are only "real-equivalent," i.e., equivalent if the values and

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

Fig. 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`.

operations are understood as taking place in the mathematical real numbers. This is in fact the case with most parallel numerical programs (cf. [1]).

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.

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.

Factorial. The factorial program computes $n!$. The specification performs the computation using recursion, while the implementation uses iteration.

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.

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 it 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 [2, Sec. 3.6]. Manger-worker programs are notorious difficult to verify because of the combinatorial blow-up in the numer 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 multipliation function. This function is exactly the same in `matmat_mw.c` and in the sequential version `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 multipliation function does, only that its output is a deterministic function of its input.

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 also send the output to a sequence of plain-text files.

Implementation `diffusion_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. 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. [3, Sec. 2.17]).

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.

Laplace. These programs, based on the description of [4, 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`.

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, 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.

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 programs are real (but not Herbrand) equivalent.

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. An interval is refined by cutting it 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.

References

1. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining symbolic execution with model checking to verify parallel numerical programs. ACM TOSEM **17**(2) (2008) Article 10, 1–34
2. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA (1999)
3. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI—The Complete Reference, Volume 1: The MPI Core. Second edn. MIT Press (1998)
4. Andrews, G.R.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley (2000)