The State of Software for Evolutionary Biology

Diego Darriba, Tomáš Flouri, and Alexandros Stamatakis

Supplementary material

This supplement has two sections. In Section 1 we provide additional technical details and an in-depth discussion regarding some of the codes we tested that is intended for code developers. In Section 2 we provide transcripts of all code analysis runs we executed.

Section 1 - technical details & discussion for code developers

Compiler Flags

In the following we list the compiler flags we used.
We executed `gcc` (version 4.9.1) with the `-Wall` flag enabled. Note that, there are substantial differences in the number *and* types of warnings that compilers issue. We have found that, the `clang` compiler (version 3.6.0) see http://clang.llvm.org/) generally issues considerably more warnings than `gcc`. For instance, when compiling the sequential SSE3 version of RAxML (v8.1.16) with the `gcc` warning flags used here, we obtain just three minor and irrelevant warnings about unused parameters in function calls. However, using `clang` with flags `-Weverything -Wno-padded` we obtain around 900 warnings for RAxML.
More importantly, `clang` also detects type errors in library function calls such as, for instance, the `malloc()` typecast issue (see below and main text).
Finally, for JAVA-based codes (i.e., Beast) we used the `-Xlint:all` argument, which enables all recommended warnings in the JAVA compiler. We then counted and categorized the warnings we obtained into major and minor as follows:

Comp	Type	*Warning*	Description
GCC	Minor	*-Wchar-subscripts*	array subscript has type ‘char’
		-Wformat-contains-nul	embedded ‘\0’ in format
		-Wreorder	‘x’ will be initialized after ‘y’
		-Wunused-but-set-variable	variable ‘x’ set but not used
		-Wunused-result	ignoring return value of ‘x’
		-Wunused-variable	unused variable ‘x’
	Major	*-Waggressive-loop-optimizations*	iteration 1u invokes undefined behavior
		-Warray-bounds	array subscript is above array bounds

		-Wformat=	format ‘x’ expects argument of type ‘y’, but argument n has type ‘z’
		-Wmaybe-uninitialized	‘x’ may be used uninitialized in this function
		-Wparentheses	suggest explicit braces to avoid ambiguous ‘else’
		-Wpointer-sign	pointer targets in passing argument X of foo differ in signedness
		-Wreturn-type	control reaches end of non-void function
		-Wsign-compare	comparison between signed and unsigned integer expressions
Java	Minor	*cast*	redundant cast
		deprecation	x in y has been deprecated
		fallthrough	possible fall-through into case
		options	bootstrap class path not set in conjunction with -source 1.6
		path	bad path element
		rawtypes	found raw type
		serial	serializable class x has no definition of serialVersionUID
		unchecked	unchecked conversion
	Major	*static*	static variable should be qualified by type name

Hoare logic

One classic and straightforward example for programming errors that can be prevented by using Hoare logic, that will also be detected by enabling compiler warnings, is the missing `default` statement in `C switch` clauses.

A *good* code would look as follows:

```
int c;
...
switch(c)
{
case 1:
  doSomething;
  break;
case 2:
  doSomethingElse;
  break;
default:
  assert(0);
}
```

In the above example, we state by Hoare logic (the assertion will always fail) that the variable `c`

can only assume values *1* or *2*.
Omission of the `default` clause can yield unexpected or unspecified program behavior in cases where the programmer simply forgot one or more cases or when a `break` statement is missing.
We advocate the explicit use of assertions instead of explicit conditional `if` statements that serve the same purpose because conditional statements may affect program performance. Thus, when a code is compiled with `-DNDEBUG` the assertions will be automatically removed from the code, thereby making it faster. We also suggest a clear distinction of program errors, that is, assertions should be used for assessing correctness during development and testing, whereas conditional statements should be used to inform users about incorrect input formats, incompatible command line flags etc.

Correct arguments for `malloc()`

The prototype function is void `*malloc(size_t size);`. Our emphasis here is on the fact that the argument is of `size_t` which, according to the C99 standard, is an unsigned integer data type of at least 16 bits. The actual size in bits depends on the target processor.

A frequent programming error is to use signed integers (with a typical range of *-2,147,483,648* to *+2,147,483,647* for *32* bit) in memory allocations, for instance:

```
int
  a = 5000000,
  b = 4000000;

double
  *v = (double *)malloc(a * b * sizeof(double));
```

Evidently, the above assignment will exceed the signed 32-bit integer number range and the `malloc()` call will fail. The correct way to do it is as follows:

Either, according to the implicit type conversion rules in C/C++, change the order of the operands to `sizeof(double) * a * b`, or typecast the first operand `(size_t)a * b * sizeof(double)`. However, we consider the following:

v = (double *)malloc((size_t)a * (size_t)b * sizeof(double));

as being a good and less implicit solution that does not require an understanding of C/C++ type conversion rules. Ideally though, both variables `a` and `b` should be defined as being of type `size_t` right away. This would be the cleanest solution.

Another common error is to typecast only the result of the operation:

For instance, what we denote as *MisCast* error is present in MrBayes where memory size is cast incorrectly as

```
malloc((size_t)((x+1)*sizeof(int)))
```

whereas a correct cast would be

```
malloc(((size_t)x+1)*sizeof(int))
```

In the ms population genetics code we observe a third (denoted as *WrongCast*) error type; here variables inside `malloc()` are cast to `unsigned int` instead of `size_t`. We therefore distinguish between *NoCast* (i.e., missing typecast as described before) and *MisCast* (misplaced cast) and *WrongCast* errors.

Selected Issues with tested codes

In the following we discuss some tool-specific issues that we consider worth reporting.

PAML

The entire package generates thousands of compiler warnings. The `malloc()` typecast issue is present, but all calls to `malloc()` are checked for success. Memory-wise some components do not free the memory at the end but we did not detect any leaks. In `mcmctree`, however a static array is accessed out of its bounds (see Section 2 of this supplement for further details).

PHYML

The code is generally clean with the exception of the `malloc()` issue (NoCast); potentially more assertions could be added to the code.

MrBayes

Apart from the `malloc()` *MisCast* error, MrBayes faces some memory issues that might be critical. Firstly, there is an out-of-bounds array access:

```
proposal.c:17192:37: warning: array subscript is below array bounds
[-Warray-bounds]
proposal.c:17188:24: warning: array subscript is above array bounds
[-Warray-bounds]
```

Secondly, on a small test alignment (see supplement for details) `valgrind` reports several invalid read accesses. The above two memory management errors might induce incorrect program behavior and/or crashes.

T-Coffee

T-Coffee generated a comparatively large number of 345 major warnings. Apart from that, it cannot read alignment files with upper-case letters in the file name and will exit with a segmentation fault.

Also, the following macro definition is problematic:

```
#define ACTION(x) ((n_actions>=(x+1))?action_list[x]:NULL)
```

Calling `ACTION(x)` as follows: `ACTION(cond ? a : b)` may cause problems. When extending large codes, programmers usually cannot afford the luxury to revise potential mistakes. Based on unit testing, programmers assume that the code is correct and build upon the existing functionality. One potential pitfall with this approach is the usage of macros.
Calling the above macro in the following way

```
ACTION(cond ? a : b)
```

will expand the call to

```
((n_actions>=(cond ? a : b + 1))?action_list[cond ? a : b]:NULL)
```

which was probably not the original intention. What one would expect to happen is that the condition is evaluated, and the result (`a` or `b`) is incremented by one. However, in the current execution the condition is evaluated, and the result is either `a` or `b+1`.

A correct macro definition would be

```
#define ACTION(x) ((n_actions>=((x)+1))?action_list[(x)]:NULL)
```

which expands to

```
((n_actions>=((cond ? a : b) + 1))?action_list[(cond ? a : b)]:NULL)
```

This is probably what the programmer expected to happen.

BEAST

BEAST is the only code written in Java and will therefore generate different types of warnings.

For BEAST we obtained the largest number (3800) of warnings among all codes. These include calls to deprecated functions and 13, potentially deliberate, fallthroughs in `switch` statements (analogous to missing `break` statements in the C `switch` construct). Note that, while JAVA warning flags do exist, based on our experience, they are rarely being used to improve codes.

An analysis with `valgrind` was not feasible. While it supports JAVA, we obtained over 10,000,000 errors. We assume that these are caused by the Java virtual machine in conjunction with garbage collection. To further analyze BEAST (v1.8.1) we deployed the dedicated FindBugs (http://findbugs.sourceforge.net) tool for JAVA codes. The tool found and categorized a total of 6193 potential bugs/issues.

The bug categories are distributed as follows: Bad practice (1843), Malicious code vulnerability (1453), Dodgy code (1451), Performance (863), Internationalization (290), Correctness (200), Multithreaded correctness (71), Experimental (22). For details on the categories, please refer to the FindBugs documentation.

SOAP

SOAP also generated a large number of 145 major warnings that we consider to be potentially critical in the sense that the program may not work according to its specification. It does not use any assertions and also shows the `classic' `malloc()` *NoCast* error.

Further Numerical Issues

We first provide an additional example to illustrate how the mere addition order of real values can affect the log likelihood score.

As input we used the values of 29,149 per-site log likelihoods at the root of a phylogeny for a DNA dataset of 125 taxa as calculated by RAxML (code and data available at: https://github.com/stamatak/softwareQuality). Then, we generated 1000 random permutations of these values to generate distinct addition orders and recorded the overall log likelihood scores. The maximum was *-826079.30333342* and the minimum was *-826079.30333344* with a deviation of *0.000000021* **log** likelihood units.
Evidently, for larger phylogenomic datasets with more sites, these roundoff deviations will increase. Keep in mind that, this deviation is due to reordering computations at the root only. For branch length optimization in Maximum Likelihood (ML) programs using the Newton-Raphson method, two such cumulative sums (first and second derivative of the likelihood) are computed at each branch of the tree. This explains why different ML programs can return different ML scores for the same tree.

An interesting general experiment highlighting the magnitude of round-off errors can be found at the following link http://i.stack.imgur.com/UVWuE.png

We reproduced the results and found that with *10^8* iterations which is not a very large number using single precision floating point we get an error of approximately *10^7*. The result of the operation should be zero, but it was 10 millions instead! Of course this is constructed worst-case example. However, it does illustrate why round-off errors are something we should be aware of.

In the following we discuss an issue that is related to computational performance when using floating point arithmetics.

With respect to floating point performance, the effect of so-called de-normalized floating point numbers on execution times is widely unknown. For instance, the exact same number of floating point operations can require execution times that differ by up to *50%* for different input values (see http://dl.acm.org/citation.cfm?id=1775087).
A micro-benchmark (available https://github.com/stamatak/denormalizedFloatingPointNumbers) illustrating this is available. We extracted it from RAxML and it exactly shows this unexpected run-time variance for the likelihood function. While developing the Evolutionary Placement Algorithm (EPA, http://sysbio.oxfordjournals.org/content/60/3/291) we observed that some reads almost required twice the time to be placed into the reference tree than others because of de-normalized floating point values.

Section 2 - code analysis transcripts

Tools for phylogenetic inference

Standard procedure:

Make: compile with gcc and clang and warning flags enabled.

Valgrind: run with valgrind using the example datasets and suggested parameters included with each tool.

Malloc: Checked for one of the three error types (NoCast, MisCast, WrongCast)

Assertions: grep for assertions in source code

PAML 4.8

Make: Compiles correctly, but with thousands of warnings; reports a large number of unused variables.

Malloc: No typecasts, but the success of the allocation is always checked

Assertions: No assertions

baseml

Make: 45 Warnings: Using `char` data type as array subscript, ignoring return values of `fscanf()` and `fgets()`, 8 warnings of type: "`pointer targets in assignment differ in signedness`"

Valgrind: A few errors detected, and memory leaks at exit

```
valgrind --tool=memcheck baseml examples/baseml.ctl

==12945== Conditional jump or move depends on uninitialised value(s)
==12945==    at 0x40B3DA: OutSubTreeN (treesub.c:2938)
==12945==    by 0x41054F: StepwiseAddition (treesub.c:2954)
==12945==    by 0x428889: main (baseml.c:338)
…
==12945== HEAP SUMMARY:
==12945==    in use at exit: 20,484 bytes in 19 blocks
==12945==   total heap usage: 53 allocs, 34 frees, 8,069,923 bytes
allocated
==12945==
==12945== LEAK SUMMARY:
==12945==    definitely lost: 0 bytes in 0 blocks
==12945==    indirectly lost: 0 bytes in 0 blocks
==12945==    possibly lost: 0 bytes in 0 blocks
==12945==    still reachable: 20,484 bytes in 19 blocks
==12945==         suppressed: 0 bytes in 0 blocks
```

basemlg

Make: 47 Warnings: Same as above

Valgrind: Memory leaks at exit

```
valgrind --tool=memcheck basemlg examples/basemlg.ctl

==12942==    in use at exit: 10,640 bytes in 20 blocks
==12942==   total heap usage: 37 allocs, 17 frees, 42,209 bytes
allocated
==12942==
==12942== LEAK SUMMARY:
==12942==    definitely lost: 1,360 bytes in 1 blocks
==12942==    indirectly lost: 0 bytes in 0 blocks
==12942==    possibly lost: 0 bytes in 0 blocks
==12942==    still reachable: 9,280 bytes in 19 blocks
==12942==          suppressed: 0 bytes in 0 blocks
```

chi2

Make: 2 warnings: Printing a double as integer, ignoring return value from `scanf()`.

Valgrind: No errors or leaks

```
valgrind --tool=memcheck chi2 1 3.84

==12927== HEAP SUMMARY:
==12927==  in use at exit: 0 bytes in 0 blocks
==12927==   total heap usage: 0 allocs, 0 frees, 0 bytes
allocated
==12927==
==12927== All heap blocks were freed -- no leaks are
possible
```

codeml

Make: 83 warnings. Same as `baseml`, some additional type mismatches in `printf()`

Valgrind: Does not free the memory at the end, no errors.

```
valgrind --tool=memcheck codeml
examples/Technical/Simulation/Codon/codeml.ctl

==12905== HEAP SUMMARY:
==12905==     in use at exit: 5,063,272 bytes in 13 blocks
==12905==   total heap usage: 54 allocs, 41 frees, 5,515,947 bytes
allocated
==12905==
==12905== LEAK SUMMARY:
==12905==    definitely lost: 0 bytes in 0 blocks
==12905==    indirectly lost: 0 bytes in 0 blocks
==12905==      possibly lost: 0 bytes in 0 blocks
==12905==    still reachable: 5,063,272 bytes in 13 blocks
==12905==         suppressed: 0 bytes in 0 blocks
```

evolver

Make: 83 warnings. 3 `char` data types as array subscripts, 3 pointers with different sign and 1 '\0' encoded in the format string of `scanf()`. The rest are ignored return values.

Valgrind: Memory is leaked after each iteration.

Evolver is an interactive tool, so we did 2 iterations for computing randomly an unrooted and an rooted tree with 100 taxa, second one computing branch lengths from a birth-death process.

```
valgrind --tool=memcheck ./evolver

==12870==     in use at exit: 205,864 bytes in 5 blocks
==12870==   total heap usage: 5 allocs, 0 frees, 205,864 bytes
allocated
==12870==
==12870== LEAK SUMMARY:
==12870==    definitely lost: 110,248 bytes in 2 blocks
==12870==    indirectly lost: 0 bytes in 0 blocks
==12870==      possibly lost: 0 bytes in 0 blocks
==12870==    still reachable: 95,616 bytes in 3 blocks
==12870==         suppressed: 0 bytes in 0 blocks
```

```
==12870== Rerun with --leak-check=full to see details of leaked
memory
```

mcmctree/infinitesites

Make: 92 warnings. Most of them as before, there is a more interesting one:
`“lgene”` is declared in `mcmctree.c` as
int `lgene[1]`,
but there is a loop accessing positions 1 and 2:

`treesub.c:292 -> “for(i=0;i<3;i++) com.lgene[i]=com.ls/3;”`.

This produces the following warnings:

```
warning: iteration 1u invokes undefined behavior
[-Waggressive-loop-optimizations]
warning: array subscript is above array bounds
   [-Warray-bounds]
```

Valgrind: There are some still reachable memory blocks at the end, but no errors.
However, the warning above might produce a segmentation fault.

```
valgrind --tool=memcheck ./mcmctree DatingSoftBound/mcmctree.ctl

==12588==    in use at exit: 13,324 bytes in 1 blocks
==12588==   total heap usage: 129 allocs, 128 frees, 10,479,421 bytes
allocated
==12588==
==12588== LEAK SUMMARY:
==12588==    definitely lost: 0 bytes in 0 blocks
==12588==    indirectly lost: 0 bytes in 0 blocks
==12588==    possibly lost: 0 bytes in 0 blocks
==12588==    still reachable: 13,324 bytes in 1 blocks
==12588==         suppressed: 0 bytes in 0 blocks

valgrind --tool=memcheck ./mcmctree
DatingSoftBound/mcmctree.Infinitesites.ctl

==12821==    in use at exit: 139,401 bytes in 33 blocks
==12821==   total heap usage: 135 allocs, 102 frees, 10,485,865 bytes
allocated
==12821==
==12821== LEAK SUMMARY:
```

```
==12821==    definitely lost: 0 bytes in 0 blocks
==12821==    indirectly lost: 0 bytes in 0 blocks
==12821==    possibly lost: 0 bytes in 0 blocks
==12821==    still reachable: 139,401 bytes in 33 blocks
==12821==          suppressed: 0 bytes in 0 blocks
```

pamp

Make: 29 warnings. Type `char` used as array subscript, ignored function return values, 2 pointers with different sign.

Valgrind: Small leaks and no `free()` calls at all. No errors.

```
valgrind --tool=memcheck ./pamp examples/pamp.ctl

==12383== HEAP SUMMARY:
==12383==    in use at exit: 8,020,060 bytes in 28 blocks
==12383==   total heap usage: 52 allocs, 24 frees, 8,081,686 bytes
allocated
==12383==
==12383== LEAK SUMMARY:
==12383==    definitely lost: 7,456 bytes in 1 blocks
==12383==    indirectly lost: 0 bytes in 0 blocks
==12383==    possibly lost: 0 bytes in 0 blocks
==12383==    still reachable: 8,012,604 bytes in 27 blocks
==12383==          suppressed: 0 bytes in 0 blocks
```

yn00

Make: 31 warnings. Same as before

Valgrind: Almost the entire memory is leaked. No errors

```
valgrind --tool=memcheck ./yn00 examples/yn00.ctl

==12367== HEAP SUMMARY:
==12367==    in use at exit: 207,880 bytes in 15 blocks
==12367==   total heap usage: 24 allocs, 9 frees, 228,894 bytes
allocated
==12367==
==12367== LEAK SUMMARY:
==12367==    definitely lost: 200,000 bytes in 1 blocks
==12367==    indirectly lost: 0 bytes in 0 blocks
```

```
==12367==    possibly lost: 0 bytes in 0 blocks
==12367==    still reachable: 7,880 bytes in 14 blocks
==12367==          suppressed: 0 bytes in 0 blocks
```

PHYML 20141009

Make: No major warnings

Valgrind: No errors or leaks

```
valgrind --tool=memcheck ./phyml -i examples/nucleic

==12976==    in use at exit: 0 bytes in 0 blocks
==12976==   total heap usage: 262,708 allocs, 262,708 frees,
66,232,882 bytes allocated
==12976==
==12976== All heap blocks were freed -- no leaks are possible
```

Malloc: No typecasts (Error type: NoCast)

Assertions: Only a few assertions

MrBayes 3.2.4-svn(r926)

Downloaded latest development version from sourceforge (28.10.2014). MrBayes is written in C with a total of 94,432 lines of code (LoC). Loc determined using

```
cloc `find $HOME/mrbayes -type f -iname "*.c" -o -iname
"*.h" -o -name "*.cpp" -o -name "*.cc" -o -name "*.hpp"`
```

which ignores comments and empty lines.

When compiling with `-Wall`, we get two warnings only:

```
proposal.c:17192:37: warning: array subscript is below array
   bounds [-Warray-bounds]
proposal.c:17188:24: warning: array subscript is above array
   bounds [-Warray-bounds]
```

When typing:

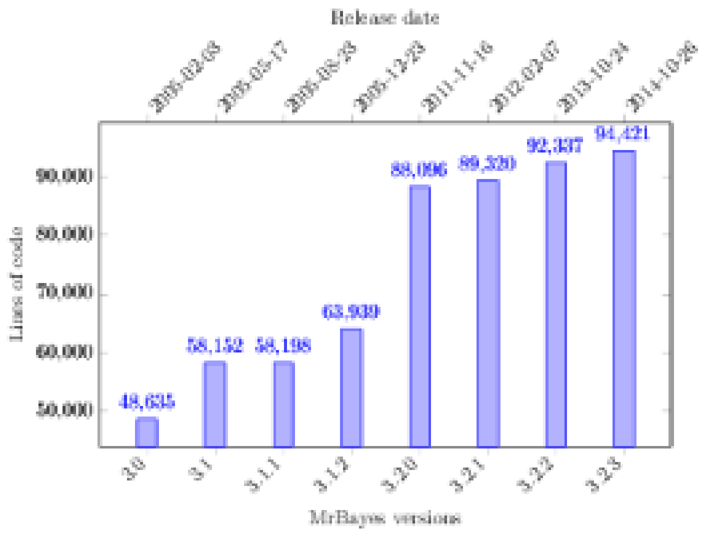
```
valgrind ./mb
execute co1_1.nex
mcmc
```

`valgrind` reports several invalid reads on uninitialized memory!

Some memory is also leaked.

Test datasets were downloaded from:
http://www.molecularevolution.org/software/phylogenetics/mrbayes

MrBayes code growth over time is illustrated in the plot below:



Except `bayes.c`, all files contains assertions (223 in total).

Typecasts in `malloc()` are used incorrectly. Typically, in MrBayes, typecasts within have the following format:

```
malloc((size_t)(x*sizeof(type)));
```

This is incorrect, see the following simple example:

```
int x = INT_MAX;

void * p = malloc((size_t)((x+1)*sizeof(int)));
```

The result of the above line will be a `NULL` pointer since `x` will overflow and assume a value of 0.
Typecasting needs to be applied to variables that will be used in operations and not to the result of operations that may already have overflown/underflown, e.g.:

```
void * p = malloc(((size_t)x+1)*sizeof(int));
```

RAxML 8.2.11

Make: GCC compilation does not report major warnings, but sporadic unused parameters or functions. Nonetheless, these warnings show up because of the different Makefiles provided (e.g., 5 for SSE, 9 for AVX, 11 for AVX+PTHREADS). Several functions and parameters are used or not depending on the selected flavor. Clang, however, reported 964 warnings when used with -Weverything -Wno-padded flags.

Valgrind: No errors nor hard leaks detected, but several soft leaks (i.e., reachable blocks at the end of the execution).

Malloc: the typecast issue is generally tackled by placing the sizeof(x) expression in first place, which casts the following operands to size_t. Hence, it is not necessary to explicitly use the size_t cast.

Assertions: Assertions are thoroughly used throughout the code.

Population genetics software

ms

No version information available for ms. Last update of source files was done in 8 Sep 2014 (rand1.c).

Written in C. Total of 2,063 lines of code. Counted using

```
cloc `find $HOME/ms -type f -iname "*.c" -o -iname "*.h" -o
-name "*.cpp" -o -name "*.cc" -o -name "*.hpp"`
```

which ignores comments and empty lines.

When compiling with `gcc -Wall`, we obtain 33 warnings of the following categories:

33	**Total number of warnings**
22	unused or set but not used variables
1	control reaches end of non-void function
9	variables may be used uninitialized

Ms incorrectly typecasts most variables from `int` to `unsigned int` when calling `malloc()`.

No assertions are being used.

SweepFinder

No version provided for SweepFinder. Downloaded on February 23, 2015 from http://http://people.binf.ku.dk/rasmus/webpage/sf.html

Make: A few warnings but copying command line arguments into variables is done as follows:

```
char snpfn[1000];
sprintf(snpfn, argv[2]);
```

Apart from the missing format in the `sprintf()` function instruction, this may lead to a stack buffer overflow. This can be risky if the kernel or the compiler do not check for this. This should not be a problem on modern systems, but it is not good practice; see http://insecure.org/stf/smashstack.html

When a custom frequency spectrum is specified, the tool assumes that the file covers the whole sample size. This constraint causes an assertion to fail,

```
freq.c:328: loadfreq: Assertion `p[i] >=0.0' failed.
```

Normally, the tool should check this beforehand and produce a more informative error message.

Valgrind: No errors or leaks, some memory still reachable at the end.

```
valgrind --tool=memcheck ./SweepFinder -m 100 infile out

==13598== HEAP SUMMARY:
==13598==    in use at exit: 220,532 bytes in 674 blocks
```

```
==13598==   total heap usage: 2,692 allocs, 2,018 frees, 1,036,396
bytes allocated
==13598==
==13598== LEAK SUMMARY:
==13598==    definitely lost: 0 bytes in 0 blocks
==13598==    indirectly lost: 0 bytes in 0 blocks
==13598==    possibly lost: 0 bytes in 0 blocks
==13598==    still reachable: 220,532 bytes in 674 blocks
==13598==          suppressed: 0 bytes in 0 blocks
```

Sequence alignment software

Mafft v7.205

Downloaded latest mafft v7.205 without extensions.
Written in C. Total of 57,688 LoC. Counted using

```
cloc `find $HOME/mafft -type f -iname "*.c" -o -iname "*.h"
-o -name "*.cpp" -o -name "*.cc" -o -name "*.hpp"`
```

which ignores comments and empty lines.

When compiling with `gcc -Wall`, we get 134 warnings of the following types:

134	**Total number of warnings**
73	unused or set but not used variables
60	variables may be used uninitialized
1	operation may be undefined

The undefined operation is the following:

```
blosum.c:290:16: warning: operation on '*(*(matrix +
   (sizetype)((long unsigned int)i * 8ul)) + (sizetype)((long
   unsigned int)i * 8ul))' may be undefined [-Wsequence-point]
   matrix[i][i] = matrix[i][i] = (double)1.0;
```

line 290 in `blosum.c` looks like this:

```
matrix[i][i] = matrix[i][i] = (double)1.0;
```

Some simple executions of the program of the program fail:

```
$ ./mafft-profile --help
file --help
illegal option -
Segmentation fault
```

```
$ ./mafft-profile --a
file --a
illegal option -
options: Check source file !  ?
```

```
$ ./mafft-profile --b
file --b
illegal option -
Error in myatoi()
```

```
$ ./mafft-profile -h
Segmentation fault
```

Valgrind execution:

All memory leaks are blocks of allocated memory that are still reachable program termination. Valgrind reports invalid file descriptors at `close()`.

There are several memeory allocations of the form: `alloc(a+b)`. For instance, line 2013 of `tbfast.c`

```
calloc(nogaplen+1, sizeof (RNApair *));
```

where `nogaplen` is of type (`signed`) `int` and was obtained by:
`nogaplen = stlren(bseq[i]);`

When feeding the program with sequences of length `MAX_INT` we may expect problems/errors.

No assertions are used.

T-Coffee v20141026_23:18

Downloaded latest tcoffee version 20141026_23:18.
Written in C. Total of 124,282 LoC. Counted using

```
cloc `find $HOME/tcoffee -type f -iname “*.c” -o -iname
“*.h” -o -name “*.cpp” -o -name “*.cc” -o -name “*.hpp”`
```

which ignores comments and empty lines.

When compiling with `gcc -Wall`, we get 964 warnings of the following categories:

964	**Total number of warnings**
551	unused or set but not used variables/functions
2	wrong format specifiers
124	comparisons between signed and unsigned data types
11	unknown pragmas
117	variables may be used uninitialized
8	suggestions for parenthesis in logic operations
68	/* within comment
32	null argument where non-null required
5	control reaches end of non-void function
17	no return statement in function returning non-void
12	array subscript has type char
4	too many arguments for format
6	operations may be undefined
4	dereferencing type-punned pointer will break strict-aliasing rules example from tcoffee leading to this warning: `int n;`

	`double x;`
	`...`
	`n = (*(long long *)&x);`

1	second parameter of ‘va_start’ not last named argument
1	converting to non-pointer type int from NULL
1	use of tmpnam function is dangerous, better use mkstemp

Bugs in the program:

tcoffee has a command-line parameter `-infile` to specify the input alignment file. If the filename contains a capital letter (which in our case it did), for example ‘Alignment.fa’, tcoffee magically ignores the capital letter, tries to read ‘lignment.fa’ and then exits with a segmentation fault.

Another critical mistake is the macro definition in `lib/util_lib/reformat.c` lines 15-17.
The macro `ACTION(x)` is defined as

```
#define ACTION(x) ((n_actions>=(x+1))?action_list[x]:NULL)
```

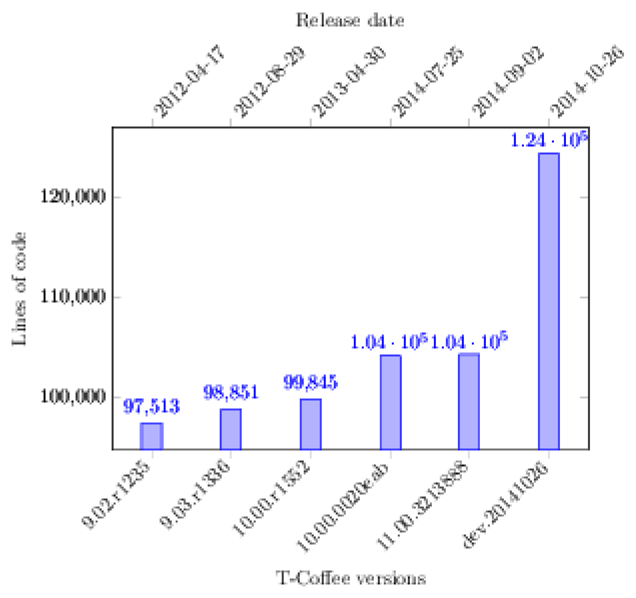
However, calling `ACTION(x)` in the form:

```
ACTION(cond ? a : b)
```

will lead to problems.

Reachable memory blocks at the end, except around ~ 30 bytes (constant amount) which are always leaked.

Code complexity increase (LoC) of T-Coffee over time:



Prank v.140603

Downloaded latest prank v.140603.
Written in C++. Total of 23,947 LOC. Counted using

```
cloc `find $HOME/prank -type f -iname "*.c" -o -iname "*.h"
-o -name "*.cpp" -o -name "*.cc" -o -name "*.hpp"`
```

which ignores comments and empty lines.

When compiling with `gcc -Wall`, we get 170 warnings of the following categories:

170	**Total number of warnings**
6	unused or set but not used variables/functions
83	comparisons between signed and unsigned data types
2	variables may be used uninitialized
26	suggestions for parenthesis in logic operations
1	control reaches end of non-void function
23	no return statement in function returning non-void

4	passing pointer instead of bool value (will always evaluate as true)
25	deleting object of polymorphic class which has non-virtual destructor might cause undefined behaviour

Valgrind does not report memory leaks, however it reports that several conditional jumps depend on uninitialized memory.

Software for divergence time estimation

BEAST v1.8.0

By default warnings are not displayed in JAVA and it compiles without problems. Ant enforces to use JDK 1.6 and it produces a record (for the codes analyzed in our paper) of approximately 3800 Warnings, including a large number of calls to deprecated functions. Setting the source to 1.7 BEAST compiles as well, with the same warnings.

Warning list: 87 deprecations, 2356 rawtypes, 478 unchecked, 94 redundant casts, 750 [serial] (serializable classes without ID), 13 fall-throughs in switch statements, 22 static attributes without qualifier.

For readers not familiar with Java, here are some explanations:

Deprecated: Use of methods that are explicitly marked as deprecated. Probably they will not be supported in future versions. However, these warnings come from deprecated annotations in the BEAGLE code.

Serial: Serializable classes should declare a static version identifier such that when an instance is loaded one can compare it with the current identifier and prevent errors from loading data stored in a different format.

Static attributes without qualifier: When a static attribute is called from a non-static member in the source code, it should be addressed as part of the class and not the instance (i.e., `Classname.staticAttribute` instead of just `staticAttribute`). Note that, `static` in java denotes a class member as in C++/C#. It does not have the same meaning as in C.
However, sometimes the authors of BEAST are updating static attributes inside the class constructor. For example:

```
public AbstractPCARateMatrix(String name, DataType dataType,
String dir) {
```

```
                this.name = name;
                this.dataType = dataType;
                this.dataDir = dir;
        }
        private static String name;
        protected static String dataDir;
        protected DataType dataType;
```

In the example above, `name` and `dataDir` are accessed as members of the instance and **not** the class (`this.name`, ...). They are updated with every instantiation, but since `name` and `dataDir` are static variables, this will also affect every already existing instance. In particular, `dataDir` is also updated for every subclass. Maybe this class is instantiated only once, or it does not have any effect at all during the execution, but this is bad practice.

Rawtypes and unchecked: These 2 warnings are related to each other. A rawtype means that a generic class/interface is declared without specifying the type argument. For example. `Set` instead of `Set<int>`. If one attempts to operate with a rawtype on a generic class with a defined type, one will obtain an `unchecked` warning. For example, `a.add(b)`, where `a` is a `Set<int>` and `b` is a rawtype. This is like working with `void *` pointers in C despite the fact that one already knows that they are integer pointers.

Valgrind: Although Valgrind supports java, it was impossible to use. We obtained the following output:

```
==19039== More than 10000000 total errors detected.  I'm not reporting any more.
==19039== Final error counts will be inaccurate.  Go fix your program!
==19039== Rerun with --error-limit=no to disable this cutoff.  Note
==19039== that errors may occur in your program without prior warning from
==19039== Valgrind, because errors are no longer being displayed.
```

We assume this is a Valgrind issue related to the Java Virtual Machine.

FDPPDIV v1.3

Compiling produces a large number of warnings, but no major warnings. Most of them are of type: `comparison between signed and unsigned integer` in loop headers `[-Wsign-compare]`, and attribute initializations in the wrong order `[-Wreorder]` in class instantiations. Also some `maybeUninitialized` warnings were issued.

Error compiling AVX-vectorized version. The header file `immintrin.h` was commented in `cpuspec.h`. Just needed to remove the comment, or change all the intrinsics includes for `x86intrin`.

Valgrind: It does not report errors during run-time, but there is a small memory leak directly proportional to the alignment size

```
valgrind --tool=memcheck ./dppdiv-seq-sse -in test_seq.dat
-tre test_tre.phy -cal test_fos.cal -tga -clok -n 1000 -sf
100 -pf 100 -out test_fbd

==15811== HEAP SUMMARY:
==15811==  in use at exit: 36,701 bytes in 309 blocks
==15811==   total heap usage: 89,037 allocs, 88,728 frees,
12,963,276 bytes allocated
==15811==
==15811== LEAK SUMMARY:
==15811==  definitely lost: 1,888 bytes in 15 blocks
==15811==  indirectly lost: 34,813 bytes in 294 blocks
==15811==       possibly lost: 0 bytes in 0 blocks
==15811==  still reachable: 0 bytes in 0 blocks
==15811==       suppressed: 0 bytes in 0 blocks

valgrind --tool=memcheck ./dppdiv-seq-sse -in test_seq.dat
-tre test_tre.phy -cal test_fos.cal -tga -clok -n 1000 -sf
100 -pf 100 -out test_fbd.pr -rnp

==15937== HEAP SUMMARY:
==15937==  in use at exit: 36,701 bytes in 309 blocks
==15937==   total heap usage: 18,532 allocs, 18,223 frees,
10,814,520 bytes allocated
==15937==
==15937== LEAK SUMMARY:
==15937==  definitely lost: 1,888 bytes in 15 blocks
==15937==  indirectly lost: 34,813 bytes in 294 blocks
==15937==       possibly lost: 0 bytes in 0 blocks
==15937==  still reachable: 0 bytes in 0 blocks
==15937==       suppressed: 0 bytes in 0 blocks
```

Comment: The input data is not checked for consistency. For example, if the alignment and tree do not match, the program exits with a segmentation fault and no additional information.

We used a 6 taxa random alignment with 20 sites, and a 5 taxa rooted tree, setting random seeds to 1 for reproducibility:

```
test.phy
       T1 atcgcgtcgacgtatctgct
       T2 attgcgtcgacgtatctgct
       T3 attgggtcgacgtagctgat
       T4 atcgggtagacgtagctgca
       T5 atcgcgtagacgtatctgat
       T6 atcgcgtagacgtatctgat

test.tre
       (T1,((T2,T3),(T4,T5)));

test.cal
       3
       -t    root    241.81
       -t    T2    T5    15.7573
```

Test command:

```
dppdiv-seq-sse -in test.phy -tre test.tre -cal test.cal
-tga -clok -pf 10 -s1 1 -s2 1
```

Output:

```
…
Contamination model fossil calibration lambda parameters: lambda1 =
    5.38996, lambda2 = 32.3397
Strict clock, initial substitution rate = 1.1191
Rate Group Elements: (1.1191) -> 0 1 2 3 4 5 7 8 9 10
6 --- 241.81
6 --- 241.81
7 --- 55.7573
Segmentation fault (core dumped)
```

Coalescence

BP&P v3.0

There is no Makefile available. The instructions for compiling are part of the README file:

```
Notes by Ziheng Yang
30 May 2010

(1) To compile, try one of the following

   UNIX gcc/icc:
        cc -o bpp -ansi -O3 -funroll-loops -fomit-frame-pointer
-finline-functions bpp.c tools.c -lm

        icc -o bpp -fast bpp.c tools.c -lm

   MAC OSX intel:
        cc -o bpp -funroll-loops -fomit-frame-pointer -finline-functions
bpp.c tools.c -lm
        cc -o MCcoal -DSIMULATION -O4 -funroll-loops -fomit-frame-pointer
-finline-functions bpp.c tools.c -lm

   Windows MSC++ 6.0/2008:
        cl -O2 bpp.c tools.c
        cl -O2 -FeMCcoal.exe -DSIMULATION bpp.c tools.c
        cl -O2 -Fesummarize.exe -DSUMMARIZE bpp.c tools.c

(2) To run an example analysis, try

cd examples

../bpp yu2001.bpp.ctl
../bpp ChenLi2001.bpp.ctl
../bpp lizard.bpp.ctl

Good luck.
```

Make: 102 warnings. 79 unused variables, including 35 ignored return values from `fscanf()` or fget(), 10 `char` data types used as array subscripts, 5 maybe uninitialised variables.

Valgrind: Does not report errors during execution, but there are reachable blocks at program termination.

```
valgrind --tool=memcheck ./bpp examples/ChenLi2001.bpp.ctl

==13655== HEAP SUMMARY:
==13655==    in use at exit: 803,528 bytes in 204 blocks
==13655==   total heap usage: 1,180 allocs, 976 frees, 3,975,101
bytes allocated
==13655== LEAK SUMMARY:
==13655==    definitely lost: 240 bytes in 1 blocks
```

```
==13655==    indirectly lost: 0 bytes in 0 blocks
==13655==    possibly lost: 0 bytes in 0 blocks
==13655==    still reachable: 803,288 bytes in 203 blocks
==13655==          suppressed: 0 bytes in 0 blocks

valgrind --tool=memcheck ./bpp examples/yu2001.bpp.ctl

==14207== HEAP SUMMARY:
==14207==    in use at exit: 1,066,749 bytes in 270 blocks
==14207==   total heap usage: 400 allocs, 130 frees, 2,533,018 bytes
allocated
==14207== LEAK SUMMARY:
==14207==    definitely lost: 0 bytes in 0 blocks
==14207==    indirectly lost: 0 bytes in 0 blocks
==14207==    possibly lost: 0 bytes in 0 blocks
==14207==    still reachable: 1,066,749 bytes in 270 blocks
==14207==          suppressed: 0 bytes in 0 blocks
```

Sequence Simulation Tools

Seq-gen v1.3.3

Make: Some warnings on unused variables and 1 warning for redefining the function `cexp()` as `complex cexp(complex a);`.

Valgrind: There are reachable blocks at program termination and small leaks. The size of the memory leak is directly proportional to the number and length of the sequences and the number of trees on which the user wants to simulate data. However, the leak size does not depend on the number of alignment replicates that are simulated per tree.

```
valgrind --tool=memcheck ./seq-gen -mHKY -t3.0 -f0.3,0.2,0.2,0.3 -l40
-n3 < examples/example.tree

==15755== HEAP SUMMARY:
==15755==    in use at exit: 171,807 bytes in 58 blocks
==15755==   total heap usage: 66 allocs, 8 frees, 235,807 bytes
allocated
==15755==
==15755== LEAK SUMMARY:
==15755==    definitely lost: 8 bytes in 1 blocks
==15755==    indirectly lost: 17,092 bytes in 7 blocks
```

```
==15755==    possibly lost: 0 bytes in 0 blocks
==15755==    still reachable: 154,707 bytes in 50 blocks
==15755==          suppressed: 0 bytes in 0 blocks
```

Malloc: No typecast (NoCast error), but the results of the allocations are checked for success.

Assertions: No assertions in the code. However, some preconditions are enforced. For example, if a set of frequencies that do not sum to 1.0 is specified, the tool automatically modifies the highest frequency.

For example

`f:={0.01, 0.01, 0.02, 0.01}`

will produce an output with frequencies

`f’={0.01, 0.01, 0.97, 0.01}`

without issuing any sort of warning.

INDELible v1.0.3

Make: 260 warnings. 193 of them a comparison between signed and unsigned integers in for loops, e.g.:

```
for (int i = 1; i < ratevec.size(); i++)
```

25 unused variables, 12 set but unused variables.

Valgrind: No errors or leaks

```
valgrind --tool=memcheck ./indelible

==15764== HEAP SUMMARY:
==15764==    in use at exit: 0 bytes in 0 blocks
==15764==   total heap usage: 15,943 allocs, 15,943 frees, 1,595,946
bytes allocated
==15764==
==15764== All heap blocks were freed -- no leaks are possible
```

We used the following control file:

```
[TYPE] NUCLEOTIDE 2

[MODEL] model1
[submodel] GTR 3.695 2.025 1 2.025 1 9.753
```

```
[rates] 0 0.8529 4
[statefreq] 0.3126 0.1613 0.1021 0.424

[MODEL] model2
[submodel] GTR 3.695 2.025 1 2.025 1 9.753
[rates] 0 0.8529 4
[statefreq] 0 0 0 1

[TREE] tree1
((t1:0.45,t2:0.8):0.4,(t3:0.12,t4:0.7):0.3,(t5:0.1,t6:0.2):0.31);
[TREE] tree2
((t1:0.0,t2:0.8):0.4,(t3:0.12,t4:0.7):0.3,(t5:0.1,t6:0.2):0.31);

[PARTITIONS] partitions
[tree1 model1 849]
[tree2 model1 849]
[tree1 model2 849]

[EVOLVE] partitions 1 alignment1
```

Malloc: There are no `malloc()` invocations. It uses the the `C++ std::vector` (resizable arrays) approach without specifying an initial size.

Assertions: None

de-Novo sequence assembly

Soap 2 r240

Downloaded latest SOAPdenovo2 r240.
Written in C/C++. Total of 37,020 lines of code. Counted using

```
cloc `find $HOME/soap -type f -iname “*.c” -o -iname “*.h”
-o -name “*.cpp” -o -name “*.cc” -o -name “*.hpp”`
```

which ignores comments and empty lines.

Compiling fails with 8 errors, which are due to the following 3 problems repeated in several files:

1. cannot find function `usleep()`
2. cannot find function `getopt()`
3. undeclared variable `optind`;

After fixing the problems by including the necessary header files and declarations, we obtain 773 warnings of the following categories (when compiling with `gcc -Wall`):

773	**Total number of warnings**
422	unused or set but not used variables/functions
144	wrong format specifiers
28	comparisons between signed and unsigned data types
54	unknown conversion type character (0xa or 0x9) in format. This was caused because of using “%\n” and/or “%\t” as a format specifier, which is wrong. The percentage character must be correctly displayed as %% (double percentage).
51	variables may be used uninitialized
2	suggestions for parenthesis in logic operations
50	implicit function declarations
4	control reaches end of non-void function
4	no return statement in function returning non-void
2	array subscript has type char
4	too many arguments for format
8	pointer targets in passing arguments differ in signedness In our case, caller passes int *, callee accepts unsigned int *.

Only Reachable memory leaks.

No assertions are used, and no typecasting (error type NoCast) when calling `malloc()`.

Abyss 1.5.2

The perfect program. No compiler warnings. There are assertions everywhere. No memory leaks. Correct `malloc()` typecasts.

Written in C/C++. 43,189 LOC.