Received: 20 April 2020 | Revised: 16 September 2020 | Accepted: 3 November 2020

DOI: 10.1002/cpe.6094

WILEY

SPECIAL ISSUE PAPER

# Memory allocation anomalies in high-performance computing applications: A study with numerical simulations

Antônio Tadeu A. Gomes[1] | Enzo Molion[2] | Roberto P. Souto[1] | Jean-François Méhaut[3]

[1]IPÊS Research Group, Laboratório Nacional de Computação Científica (LNCC), Petrópolis, Brazil

[2]École Polytechnique (Polytech Grenoble), Université Grenoble Alpes, Grenoble, France

[3]Université Grenoble Alpes, CNRS, Grenoble INP, Laboratoire Informatique Grenoble (LIG), Grenoble, France

**Correspondence**
Antônio Tadeu A. Gomes, IPÊS Research Group, Laboratório Nacional de Computação Científica (LNCC), Av. Getúlio Vargas 333, Quitandinha, Petrópolis-RJ 25651-075, Brazil.
Email: atagomes@lncc.br

**Funding information**
Conselho Nacional de Desenvolvimento Científico e Tecnológico: Grant/Award Number: 314816/2018-6; Petrobras: Grant/Award Number: 2017/00751-7

**Summary**

A memory allocation anomaly occurs when the allocation of a set of heap blocks imposes an unnecessary overhead on the execution of an application. This overhead is particularly disturbing for high-performance computing (HPC) applications running on shared resources—for example, numerical simulations running on clusters or clouds—because it may increase either the execution time of the application (contributing to a reduction on the overall efficiency of the shared resource) or its memory consumption (eventually inhibiting its capacity to handle larger problems). In this article, we propose a method for identifying, locating, characterizing and fixing allocation anomalies, and a tool for developers to apply the method. We experiment our method and tool with a numerical simulator aimed at approximating the solutions to partial differential equations using a finite element method. We show that taming allocation anomalies in this simulator reduces both its execution time and the memory footprint of its processes, irrespective of the specific heap allocator being employed with it. We conclude that the developer of HPC applications can benefit from the method and tool during the software development cycle.

KEYWORDS

high-performance computing, memory allocation, methodology

## 1 | INTRODUCTION

Researchers and practitioners have long worked on the performance analysis and optimization of high-performance computing (HPC) applications.[1-4] These specialists, however, have marginally considered the subject of *memory allocation anomalies*. An allocation anomaly occurs when the allocation of a set of heap blocks imposes an unnecessary overhead on the execution of an application. This overhead may increase the number of CPU cycles the application uses because of the heap management (*time* overhead), or increase the memory space the heap blocks actually occupy (*space* overhead). The developer of the application should therefore try to minimize this overhead in order for HPC applications to run faster and with lower memory footprint. This is usually not only in the interest of the application's user; when the application runs on a shared resource such as an HPC cluster or a cloud provider, reducing the execution time and the memory footprint of the application also means allowing for an increase in the utilization of the resource.

Memory allocation anomalies may take two forms: (i) unnecessary allocation of temporary memory; or (ii) unnecessary breakdown of large allocations into several, interlinked small allocations.

The use of programming languages and software libraries with high-level abstractions as first-class constructs—for example, classes in object-oriented languages and functions in functional languages—improve productivity and software quality, but may incur in allocation anomalies if the developer is not acquainted with the way binary code is generated from these constructs.

In this article we focus on memory allocation anomalies in the context of numerical simulators aimed at approximating solutions to partial differential equations (PDEs). The use of higher level compiled languages like C++[5,6] or dynamic languages such as Python[7,8] is increasingly common in software libraries that support the development of these simulators. Different numerical methods may be available in these libraries, and of different categories (finite elements, finite differences, finite volumes). Moreover, as mathematicians create innovative numerical methods, new libraries are made available for these methods. These libraries build on a set of fundamental linear algebra operations and data structures: matrices, vectors, matrix-matrix and matrix-vector operations, solvers of systems of linear equations, to name the most important ones. In many cases, these structures and the algorithms running atop them are irregular (e.g., polytopal meshes, heterogeneous degrees of polynomials) and of unknown *a priori* sizes. Therefore, dynamic memory allocation on the heap is a pre-requisite for this type of software. Support libraries such as Eigen,[1] Boost uBlas,[2] and NumPy[3]offer these operations and data structures in higher level languages, but allocation anomalies may arise if the developer does not properly use them.

Developers with different backgrounds aggravate the problem: modules that require more math background (e.g., in upscaling procedures that arise in multiscale methods) are usually developed by mathematicians, who may not be aware of the consequences at the level of memory allocation of choosing one operation for another. As a simple example, consider the "multiply-add" statement $a = a + a * b$. In C++ we can use the add-assignment operator and rewrite this statement as $a += a * b$. If $a$ and $b$ are of basic types (e.g., a floating point number), there is no substantial difference between the two choices. Nevertheless, if $a$ and $b$ are of vector or matrix types defined by a library such as Eigen or Boost uBlas, the first choice may incur in the creation of at least one additional temporary object (the one storing the result of $a + a * b$), irrespective of the optimization opportunities that the compiler, or the libraries themselves, or both may offer.

As a first contribution of this article, we present a method for identifying, locating, characterizing, and fixing memory allocation anomalies. The method is iterative—at each iteration, the developer chooses and tackles a region of the application code and a specific allocation size, and then measures the impact of this iteration on the performance of the target application. To experiment with the method, we chose a set of numerical simulation libraries developed at LNCC.[9] The so-called MSL (Multiscale hybrid-mixed Set of Libraries) supports the implementation of numerical simulators based on classical or multiscale finite element methods. It is developed in C++, and has 28,260 lines of non-commented code.[4] It supports hybrid parallelism with OpenMP and MPI, and integrates with many third-party libraries. Among them, Eigen and the Standard Template Library (STL) are the main sources of dynamic memory allocations—and also of allocation anomalies—in MSL.

As a second contribution of this article, we present a tool we have developed to support our method. Notice that some available tools (e.g., the Google Heap Profiler[10] and Valgrind/Massif[11]) provide, each of them, a different subset of the features that our method needs. Yet the developer cannot use these tools as an integrated toolset that provides these features in an efficient and effective way. This does not mean that the developer cannot use other tools with our method; as we show in this article, some third-party tools have been used to assess the efficiency of our method.

A shorter version of this article appeared in the Brazilian Symposium on High-Performance Computing Systems (WSCAD2019).[12] In the present article, we provide further details about our tool and about some tools already available in the literature. We also analyze more closely the impact of our method on changes in the simulator code. Finally, we extend our experiments with a more thorough statistical analysis of the results. We also include in these experiments the high-performance Google TCMalloc heap allocation library,[13] and we show that the application of our method and tool complements the performance gains this library offers to the simulator. More specifically, we show that taming allocation anomalies in a MSL simulator, when using the GLIBC heap allocator, reduces its memory footprint by 40.19% and its execution time by 17.00% on average; when using TCMalloc, these reductions are by 36.77% and 4.80% on average, respectively.

The remainder of this article is structured as follows. In Section 2, we give the context of our work and discuss related work on allocation anomalies. Section 3 presents our method and the associated tool. Section 4 validates the method experimentally. Finally, we present some concluding remarks and possibilities for future work in Section 5.

## 2 | BACKGROUND AND RELATED WORK ON MEMORY ANOMALIES

Heap allocators reserve memory chunks—so-called *heap blocks*—that applications request at runtime. The programming interface (API) of these allocators is always based on a dozen functions such as `malloc()/new()`, `free()/delete()`, `calloc()`, and `realloc()/resize()`. Most Linux distributions use GNU Libc (GLIBC)[14] as their standard C runtime library. The GNU Libc is derived from ptmalloc[15] which in turn is derived from dlmalloc.[16] Other allocators are also available such as Hoard,[17] TCMalloc,[13] and TBB Malloc.[18] The performance of these allocators may vary in execution time and in consumed memory space. Nevertheless, all of them impose on the application an overhead that increases with the amount of allocation anomalies.

[1] http://eigen.tuxfamily.org
[2] https://www.boost.org/doc/libs/1_65_1/libs/numeric/ublas
[3] https://www.numpy.org
[4] Computed with the sloccount tool.

Researchers and practitioners have already studied the issues of memory consumption by applications.

The study on the Belady anomaly[19] was the first to analyze the performance behavior of applications taking into account memory access patterns. This study focused on memory paging, but showed that a method for such analysis was needed for assessing counter-intuitive behaviors. Since then, *memory leaks*[20,21] have been the main memory issue studied by the scientific community.[22,23] Memory leaks are areas of dynamically allocated memory that the application can no longer reach or free. Memory leaks pose problems that are different from the ones the memory allocation anomalies introduce, but some profiling tools used to analyze the former can be also used to deal with the latter, as discussed in Section 2.1.

*Space leaks* are less studied than memory leaks. They occur when the application uses more memory than needed. The term was first coined in Reference 24 in the context of functional programming to refer to applications that do release the allocated memory, but later than the developer expects. Since then, researchers have revisited space leaks for different languages (e.g., Haskell in Reference 25, or Java for embedded applications in Reference 26). Space leaks are a type of memory allocation anomaly, and our method can detect and fix them, together with other anomalies.

To the best of our knowledge, there is no other work in the area of HPC applications that deals with the identification and characterization of diverse types of memory allocation anomalies, as our work does.

## 2.1 Tools

The heap allocators already provide some global statistics on the memory space the applications allocate (e.g., `malloc_stats()`). In the case of TCMalloc, statistics on the number of memory allocations per heap block size are also available. These statistics are useful to identify some potential allocation anomalies, but lack information that allows locating and characterizing these anomalies.

Some memory profiling tools can help in locating and characterizing allocation anomalies. mtrace[5]is a feature of the GNU Libc that allows detection of memory leaks caused by unbalanced `malloc`/`free`calls. It is implemented as a function call, `mtrace()`, which turns on tracing and creates a log file of addresses allocated and freed. A Perl script, also called `mtrace`, displays the log file, listing only the unbalanced combinations and—if the source file is available—the line number of the source where the allocation occurred. The tool can be used to check both C and C++ programs under Linux. One of the features that makes `mtrace` desirable is the fact that it is scalable. It can be used to do overall program debugging but can be scaled to work on a module basis as well. Similar features are also provided with the heap profiler developed for the TCMalloc heap allocator.[13]

Valgrind[11] is a framework for building analysis tools. The software contains several tools such as memory checker, call graph profiler, and heap profiler. Massif is a tool within Valgrind that measures heap and stack usage. Each heap allocation is a band drawn in a graph. Massif provides some global measurements on the memory dynamically allocated by the program functions. It also gives information on the extra memory used by allocations. (see Section 4.3 for an example.) With these nice and powerful features come also a great overhead: Massif may profile an application tens of times slower than during its regular execution.

Intel Vtune Amplifier[27] is a commercial profiler for software performance analysis. It has both a graphical user interface (GUI) and command line in versions for Linux and Microsoft Windows. Intel VTune Amplifier provides several features to analyze computing hotpots, memory accesses and memory consumption. In this article, we focused on the memory consumption feature of Intel Vtune Amplifier. This feature targets to explore memory consumption over time and identify memory objects allocated and released during the analysis run. VTune Amplfier helps to identify peaks of the memory consumption and analyze allocation stacks for the hotspot functions. The Intel VTune GUI gives a very useful view on the source lines allocating a high amount of memory.

Google developed a Heap Profiler[10] to analyze how C++ programs manage memory. The Google Heap profiler helps to detect the memory leaks of programs. It also helps to find places where the application does a lot of memory allocations. The Google Heap Profiler allows the developer to analyze limited code regions by calling the `HeapProfilerStart()` and `HeapProfilerStop()` functions.

IBM/Rational/PurifyPlus[20] is a commercial memory debugger to detect memory access errors in programs and applications. If a memory error occurs during the execution, the program will print out the exact location of error, the memory address involved, and other relevant information. PurifyPlus also discovers memory leaks, extra array bounds reads and writes, and access to unallocated memory. IBM also provide others tools such as PureCoverage for code coverage analysis.

Intel Inspector[28] is also a commercial framework for checking and debugging multithreaded programs in order to increase reliability, security of applications. Memory checking includes memory leaks, dangling pointers, and mismatched memory allocation/deallocation. Intel Inspector mainly targets to analyze the correctness of applications.

These tools can be divided into two groups: (i) tools that allow instrumenting parts of the application code; (ii) tools that offer allocation statistics. The Google Heap Profiler and the GNU Libc mtrace pertain to the first group. They allow the developer to reduce the cost of profiling, but offer no means by which the developer can select specific allocation sizes for a detailed analysis. The Intel Vtune Amplifier and Valgrind/Massif pertain to the second group. They offer the developer information regarding the location and size of the allocations, but does not allow the developer to instrument code regions. This limitation results in a high cost in terms of analysis time because of the significant overhead during the profiling. To

[5]http://man7.org/linux/man-pages/man3/mtrace.3.html

sum up, the tools presented herein are inappropriate for our method, because they cannot provide a view of allocation measurements that is at the same time precise and selective. That is the reason we have developed a tool to support our method.

## 3 METHOD AND TOOL FOR TAMING ALLOCATION ANOMALIES

The main contribution of this article is to design a new method for the identification and characterization of memory allocation anomalies in HPC applications. Allocation anomalies can be numerous and involve a wide spectrum of allocation sizes. To tackle them, we propose an iterative method and an associated tool. We depict an overview of its steps in Figure 1, which we discuss below.

First, the developer performs a *global profiling* of the number and sizes of memory allocations. From this profiling, the developer chooses an allocation size to analyze in more detail. The choice of such size may depend on the developer's knowledge of the source code and data structures. Nevertheless, if the developer does not understand why there are so many allocations of specific sizes, he or she can choose as a general rule the smallest sizes first, because they are the ones more likely to impose the highest overheads.

Second, the developer performs a *detailed profiling* of memory allocations of the chosen size. This detailed profiling allows the developer to first locate the places in the source code where these allocations happen, and then characterize their importance.

Third, the developer *refactors* the source code to reduce the amount of allocations of the chosen size. The developer may then measure the impact of the refactoring on the performance of the application, and get back to the first step for a new iteration, if needed.

**Example**: We illustrate the use of the method with a simulator implemented with MSL. More specifically, we explore the multiscale hybrid-mixed (MHM) finite-element method available in this set of libraries.[29] The MHM method is interesting for presenting our method because it is composed of different, clearly separable phases of resolution, each one with a distinct allocation pattern:

- **split**: This phase has a work distribution process. The MHM method departs from a coarse mesh defined over the physical domain of interest, and then defines a local problem for each element of this mesh. It also defines a global problem that glues together the upscaled solutions of the local problems;
- **local**: This phase has a loosely coupled process. Each local problem is solved independently from other local problems;
- **reduce**: This phase has a gather process, followed by a tightly coupled process. The solutions to the local problems are loaded as inputs to the global problem, which is then solved;
- **post**: This phase has a work distribution process followed by a loosely coupled process. The solution to the global problem is combined with the solution to each local problem, again independently from other local problems, thus rendering the final approximating solution.

In Figure 2, we show the number and size of dynamic memory allocations during the simulation of a two-dimensional diffusion process with MHM. The allocations are divided into three main groups: small allocations (up to 256 KiB), medium-sized allocations (from 256 KiB to 1 MiB), and large allocations (more than 1 MiB). Notice in Figure 2(A) that the number of allocations of the first group is much larger than the other two. Besides, as we show in Figure 2(B), within the group of small allocations the number of 4- to 512-byte allocations is much larger than that of 512-byte to 256-KiB allocations. The number of allocations of larger sizes—not shown in the figure—is even lower. We can infer from the figure that there is a huge amount of 4- to 512-byte allocations. Immediate questions may arise with regard to these allocations: (i) what is their purpose? (ii) if they might be unnecessary, where in the source code (e.g., in which phase of the MHM simulation) one could find them so that they could be replaced by an equivalent structure?

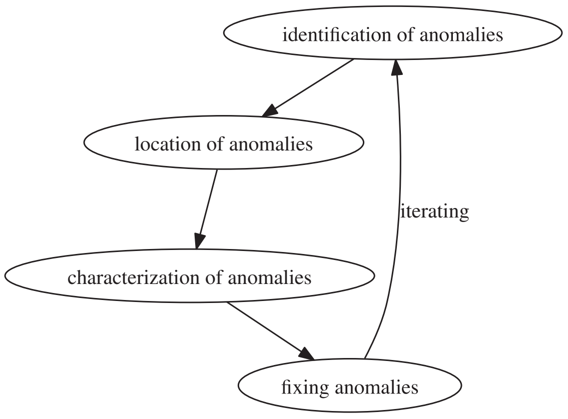
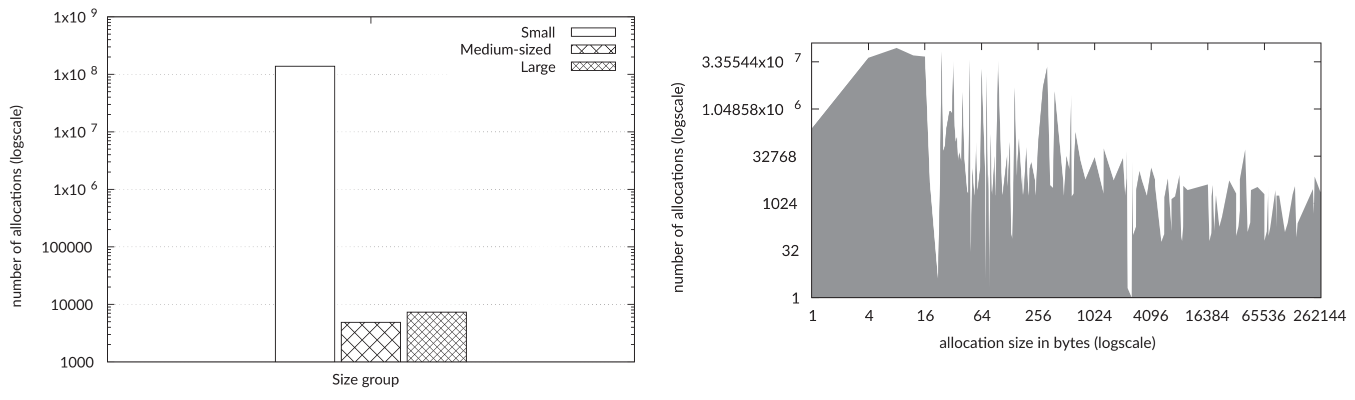


**FIGURE 1** Overview of the method steps



**(A)** Number of allocations per size group.

**(B)** Number of small-size allocations.

**FIGURE 2** Number and size of dynamic memory allocations. (A) Number of allocations per size group. (A) Number of small-size allocations

In the following, we detail the steps of our method that try to answer these questions.

## 3.1 | Identification of anomalies

First, the developer needs a global view of dynamic memory allocations in the application, to identify which parts of its code need further attention in the analyses that will follow. The main memory allocators (GNU libc, TCMalloc, TBB Malloc) provide runtime functions such as `malloc_stats` to print statistics about memory allocated with the `malloc` system call. These statistics give to the developer a global view of the usage of dynamic memory. They are not enough for the developer, though, as he or she needs to discover which parts of the application are the ones consuming the largest amounts of small-sized memory allocations.

The developer can use our tool to identify these parts more efficiently by instrumenting specific code regions (see Section 3.6). With our method and tool the developer may instrument regions at the application's main function only, without delving into lower-level pieces of code. Taking for example the MHM simulation presented at the beginning of this section, the developer can instrument each of the four phases of the simulation in the main function: split, local, reduce, post. In Table 1, we show the results of such an instrumentation—taking only allocation sizes with more than $10^6$ allocations in at least one of the four phases of the simulation.

**TABLE 1** Number of allocations per allocation size (in bytes)

| Allocation size | split | local | reduce | post |
|---|---|---|---|---|
| 8 | $289.9 \times 10^3$ | $92.43 \times 10^6$ | $19.7 \times 10^3$ | $2.6 \times 10^3$ |
| 12 | $135.2 \times 10^3$ | $53.88 \times 10^6$ | 0 | 0 |
| 16 | 0 | $49.41 \times 10^6$ | 2 | $2.7 \times 10^3$ |
| 24 | $577.5 \times 10^3$ | $72.36 \times 10^6$ | 1 | 0 |
| 32 | $12.3 \times 10^3$ | $35.86 \times 10^6$ | 3 | 0 |
| 40 | $5.0 \times 10^3$ | $3.9 \times 10^6$ | 0 | 0 |
| 48 | 7 | $39.53 \times 10^6$ | 2 | 0 |
| 64 | $12.3 \times 10^3$ | $20.3 \times 10^6$ | 1 | 0 |
| 72 | $24.6 \times 10^3$ | $14.98 \times 10^6$ | 0 | 0 |
| 96 | 5 | $36.82 \times 10^6$ | 0 | 0 |
| 144 | 3 | $5.18 \times 10^6$ | 0 | 0 |
| 288 | 0 | $5.44 \times 10^6$ | 1 | 0 |
| 320 | $36.9 \times 10^3$ | $24.7 \times 10^6$ | 0 | 0 |
| 384 | 0 | $3.88 \times 10^6$ | $4.0 \times 10^3$ | $2.7 \times 10^6$ |
| 576 | 0 | $2.95 \times 10^6$ | 0 | 0 |

After collecting overall statistics, the developer chooses a specific code region and a specific allocation size for the following steps of the method. The choice of such a size may take into account not only the total number of allocations for each size, but also the knowledge and level of openness of the code (e.g., if a library is developed by a third-party) and of the data structures involved.

## 3.2 | Location of anomalies

In this step, the developer employs our tool to collect data about *stack trace types* that match the code region and allocation size selected in the previous step of our method. A stack trace type is a set of stack traces gathered during the application execution that have exactly the same function signature at each level of the stack. A stack trace type thus represents a specific execution path the application took one or more times throughout its execution. Each function signature includes the name of the function, its parameters, the source code file, and the line number in which the function is defined.

In Listing 1, we show an extract of output from our tool when we instrumented the local phase of a MHM simulation to trace 12-byte allocations. The first line of the output shows the number of different stack trace types. The following lines give further detail for each stack trace type. We only show two of these stack trace types in the listing—notice that they have different signatures at some of the levels of their stacks. The listing also shows the number of occurrences of each stack trace type within the instrumented region of the executed code. The developer may use this number to select the most pertinent stack trace types for the next step of the method.

Listing 1: Stack trace types

```
Number of stack trace types: 14
Stack trace type 1/14 : 256 occurrences
[0]operator new(...) @ /usr/lib/x86_64-linux-gnu/libstdc++.so.6
[1]__gnu_cxx::new_allocator<int>::allocate(...) @ /usr/include/c++/7/ext/new_allocator.h:101
[2]std::vector<...>::max_size() const @ /usr/include/c++/7/bits/stl_vector.h:676
[3]std::_Vector_base<...>::_M_allocate(...) @ /usr/include/c++/7/bits/stl_vector.h:169
[4]std::vector<...>::_M_fill_insert(...) @ /usr/include/c++/7/bits/vector.tcc:504
[5]std::vector<...>::resize(...) @ /usr/include/c++/7/bits/stl_vector.h:712
[6]Element::allocNodes(...) @ .../include/element.h:288
[7]Element::alloc(...) @ .../include/element.h:266
[8]Mesh::getElem(...) const @ .../src/mesh.cpp:5007
[9]Mesh::operator[](...) const @ .../include/mesh.h:1297
[10]StdFiniteElementSpace::create() @ .../src/space_stdfiniteelem.cpp:134
[11]StdFiniteElementSpace::create(...) @ .../src/space_stdfiniteelem.cpp:187
[11]DiffusionCGProblem::configureSpacesImpl() @ .../examples/src/problem_cgdiffusion.cpp:356
[13]Problem<...>::getDataFiles[abi:cxx11]() @ .../include/problem.h:489
[14]MHMLocalProblem<...>::readDataFiles(...) @ .../include/problem_mhmlocal.h:320
[15]main @ .../src/main_mhm_diffusion_memalloc.cpp:81
[16]__libc_start_main @ /lib/x86_64-linux-gnu/libc.so.6
[17]_start @ ??:?
----------
... //other stack trace types
----------
Stack trace type 12/14 : 256 occurrences
[0]void* std::malloc(...) @ /usr/lib/x86_64-linux-gnu/libc.so
[1]void* Eigen::conditional_aligned_malloc<true>(...) @ .../Eigen3/src/Core/util/Memory.h:212
[2]int* Eigen::internal::conditional_aligned_new_auto<...>(unsigned long) @ .../Eigen3/src/Core/util/Memory.h:374
[3]Eigen::DenseStorage<...>::resize(long, long, long) @ .../Eigen3/src/Core/DenseStorage.h:555
[4]Eigen::PlainObjectBase<...>::resize(long, long) @ .../Eigen3/src/Core/PlainObjectBase.h:47
[5]void Eigen::PlainObjectBase<...>::resizeLike<...>(...) @ .../Eigen3/src/Core/PlainObjectBase.h:374
[6]Eigen::PlainObjectBase<...>::PlainObjectBase<...>(...) @ .../Eigen3/src/Core/PlainObjectBase.h:533
[7]Eigen::Matrix<...>::Matrix<...>(...) @ .../Eigen3/src/Core/Matrix.h:376
[8]Mesh::getElem(...) const @ .../src/mesh.cpp:5007
[9]Mesh::operator[](...) const @ .../include/mesh.h:1297
[10]FiniteElementSpace::setEssentialBC(...) @ .../src/space_finiteelem.cpp:36
[11]DiffusionCGProblem::configureSpacesImpl() @ .../examples/src/problem_cgdiffusion.cpp:356
[12]Problem<...>::getDataFiles[abi:cxx11]() @ .../include/problem.h:489
[13]MHMLocalProblem<...>::readDataFiles(...) @ .../include/problem_mhmlocal.h:320
[14]main @ .../main_mhm_diffusion_memalloc.cpp:81
[15]__libc_start_main @ /lib/x86_64-linux-gnu/libc.so.6
[16]_start @ ??:?
----------
... //other stack trace types
-----------
```

## 3.3 | Characterization of anomalies

To characterize an anomaly, the developer takes all the stack trace types selected in the previous step, collected for a specific allocation size and instrumented region of code, and builds from them a *call graph*indicating the different execution paths each stack trace type represents. Each entry of a stack trace type is a vertex in this call graph; the vertices shared by distinct execution paths in the call graph are functions called within distinct stack trace types. Currently, the developer must manually build this graph (c.f. Section 5).

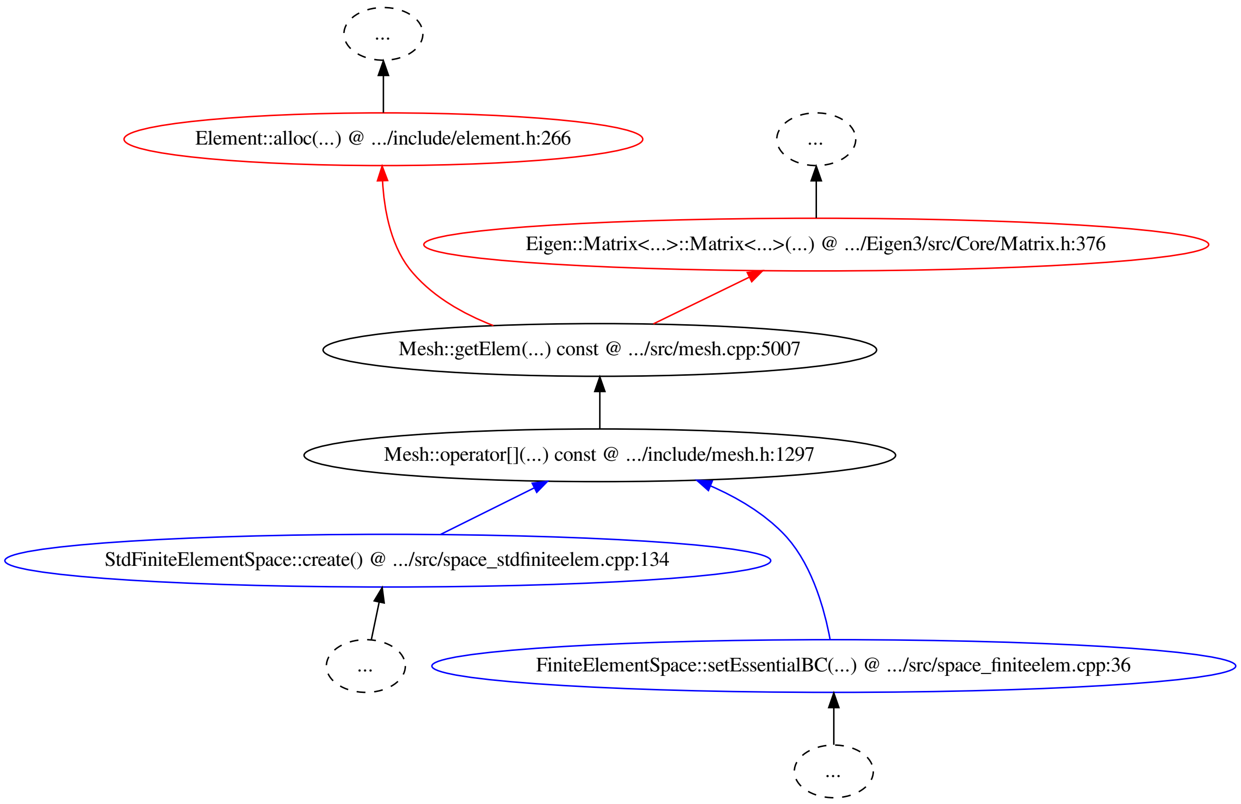

...
Element::alloc(...) @ .../include/element.h:266
...
Eigen::Matrix<...>::Matrix<...>(...) @ .../Eigen3/src/Core/Matrix.h:376
Mesh::getElem(...) const @ .../src/mesh.cpp:5007
Mesh::operator[](...) const @ .../include/mesh.h:1297
StdFiniteElementSpace::create() @ .../src/space_stdfiniteelem.cpp:134
...
FiniteElementSpace::setEssentialBC(...) @ .../src/space_finiteelem.cpp:36
...

FIGURE 3 Call graph derived from the stack trace types in Listing 1

In Figure 3, we show a snip of the call graph obtained from Listing 1. The entries in the stack trace types that Figure 3 illustrates are in blue and red in Listing 1. The vertices shared by the largest amount of execution paths—functions `Mesh::getElem()` and `Mesh::operator[]()` in the example—are the targets of the next step of our method.

## 3.4 | Fixing anomalies

In the last step of our method, the developer refactors the code so that the functionality is preserved while minimizing the undesired allocations. In the example used throughout this section, the creation of temporary objects of class `Element` caused the identified anomaly. These objects represent geometric elements in a mesh—they allocate memory for storing information such as node coordinates and node-edge connectivity. Most of the time, these objects are used for a simple processing (e.g., computing the number of degrees of freedom related to the element) and destroyed just afterwards. We illustrate this anomaly in Listing 2. In these cases, the cost of memory allocation and deallocation surpasses that of the actual computation.

Listing 2: Detected anomaly

```
class Element { ... };

class StdFiniteElementSpace : public FiniteElementSpace {
    ...
    Mesh *mMesh;                          // pointer to the submesh associated with
                                          // the local problem

    std::vector<std::vector<int> mDoF; // mapping from local indexing to
                                          // global indexing of degrees of freedom
    ...
    void create() {
        ...
        for (int L = 0; L < mMesh->mMeshInfo.mNoElems; ++L) {
            Element K = (*mMesh)[L];    // allocation of temporary structures by K
            mLocalSpace->setDoF(K, mMesh->mMeshInfo, mDoF[L]);
        }                                 // implicit deallocation of K structures
        ...
    }
    ...
};
```

Listing 3: Fixed anomaly

```
class Element : public ElementBase<Element> {
    ...
};

class SurrogateElement : public ElementBase<SurrogateElement> {
    ...
};

class StdFiniteElementSpace : public FiniteElementSpace {
    ...
    Mesh *mMesh;                        // pointer to the submesh associated with
                                       // the local problem

    std::vector<std::vector<int> mDoF; // mapping from local indexing to
                                       // global indexing of degrees of freedom
    ...
    void create() {
        ...
        for (int L = 0; L < mMesh->mMeshInfo.mNoElems; ++L) {
            SurrogateElement K = mMesh->genSurrogateElement(L);
            mLocalSpace->setDoF(K, mMesh->mMeshInfo, mDoF[L]);
        }
        ...
    }
    ...
}
```

To fix the anomaly above, we implemented a new class `SurrogateElement` with the same interface as `Element`, and a template superclass `ElementBase` to guarantee an efficient interface compatibility. Instances of `SurrogateElement`, however, only store a reference to the original mesh and an index of the element in that mesh. As a result, `SurrogateElement`operations are computationally less efficient than the equivalent ones from `Element`, but without the added cost of heap management. Listing 3 shows the new version of the code presented in Listing 2. Mind that `SurrogateElement` does not *replace* `Element` in MSL; in some situations the developer needs the representation of a geometric element dissociated from a mesh. Distinguishing these situations emphasizes the importance of characterizing anomalies.

## 3.5 | Iterating

At the end of the last step, the developer assesses the results of the iteration and chooses another allocation size for a next iteration, if needed. In Section 4.3, we illustrate the use of iterations in our method with a complete case study.

## 3.6 | Tool

The tool we have developed to support our method collects two types of measurements: (i) number of allocations per allocation size; and (ii) number of instances of each stack trace type per allocation size.

To reduce the cost of profiling, our tool allows the developer to select code regions for detailed profiling. We achieve this via a "code sectioning" system (like Google Heap Profiler). Our tool is also precise with regard to the location of allocations, because it records the full stack trace for each stack trace type. "Allocations" is used herein as a general term that designates every allocation-related low-level function.

The tool obtains both types of measurements in the same way: by linking the profiling library that accompanies our tool against the program to profile. The next two subsections give further details about our tool. Our tool is open source and available at: https://gitlab.com/EnzoMolion/profiling-library.

### 3.6.1 | Number of allocations count

To count the number of allocations, the profiling library is launched by using a profiler `init_size_profiling()` function and stopped using a profiler `end_size_profiling()`function. The tool then records any allocation between these two calls. We can then print count details via a dedicated `dump_size_profiling()` function or reset the counting via the `reset_size_profiling()` function.

Our tool works by hooking on using low-level allocation functions (`malloc()`, `realloc()`, and `calloc()`). The profiling library intercepts every call to these functions (via their redefinition) and stores the call information within an internal data structure (namely by increasing an allocation counter as value in a map having allocation sizes as keys).

### 3.6.2 | Recording of stack trace types

The profiling functions and mechanism for this functionality are the same as the ones previously described; the developer states which allocation sizes he wishes to record the call stack trace as parameters of the `init_size_profiling()` function. In the case of the stack traces, however, the information is much more complex both in its format and in its fetching. It consists in a map composed of allocation sizes as keys and lists of stack trace types as values. Thus, for every allocation size, the tool can display the different “paths” that lead to such allocation. The tool fetches the stack trace symbols using the `backtrace_symbols()` function. This may be followed by a system call (via the `popen()` function) to `c++filt` or `addr2line` command, if symbol name *demangling* is needed—this is typically the case for C++ applications, specially when templates are used. Notice that for this functionality to be available, the application must be compiled with debug options turned on.

## 4 | RESULTS

### 4.1 | Experimental setup

We conducted experiments with our method in two setups of MHM simulations. In all the experiments, the simulator code was compiled with GNU C++ compiler version 7.4, and used the standard GLIBC heap allocator.

First, we applied the method over a small use case, with the simulator compiled in debug mode so we could collect the stack trace types. For this case, we used a single-node machine configuration, with 8 cores in a single socket using OpenMP as the only parallelism technique. This case simulates a diffusion process in stationary regime over a two-dimensional domain, using quadratic approximating functions. We use a mesh of 4096 triangles at the global level. Within each element of the mesh, we solve a local problem composed of 1137 linear equations. These local problems feed a global problem composed of 99,328 linear equations. This amounts to 4,756,480 linear equations to be solved in total in the simulation, of which 4,657,152 are related to local problems.

The numbers above reflect on the amount of memory allocated in each phase of the simulation, as we show in Table 2. We use in this table the same size groups as those of Figure 2(A). The phase of local problems is the one most likely to impose the largest memory-related overheads. Hence, it has been the focus of the case study.

After each iteration, we ran a larger use case, with the simulator compiled in release mode, to measure the overall execution time and maximum memory consumption throughout the simulation. For this case, we used a two-node configuration in a cluster, with 2 sockets of 12 cores each, using OpenMP within each socket and 4 MPI ranks in total (one per socket). With this configuration, the memory footprint of this experiment fits completely in the physical memory of the nodes (no swapping to disk). This case also simulates a diffusion process in stationary regime, but over a three-dimensional domain, using cubic approximating functions. We use a mesh of 1536 tetrahedra at the global level. Within each element of the mesh, we solve a local problem composed of 12,405 linear equations on average. These local problems feed a global problem composed of 158,208 linear equations. This amounts to 19,212,288 linear equations to be solved, of which 19,054,080 are related to local problems.

We also used the larger use case for evaluating the impact of our method when we link the simulator against a high-performance heap allocator. For this evaluation, we employed TCMalloc since it is the heap allocator with the best average response time and memory usage among all heap allocators optimized for multithreaded applications.[30]

The cluster in which we ran the larger case comprises 756 nodes in total, interconnected by an Infiniband FDR (56Gbps) network with a fat-tree topology. Each node has 2 sockets Xeon E5-2695v2 Ivy Bridge 2.4GHz and 64Gb of DDR3 RAM. Since this is a shared cluster, we need to consider non-negligible fluctuations in the results due to the allocation of different nodes and the placement in different parts of the interconnection topology each time we run the simulation. Therefore, for each different configuration of the larger case (with or without the method’s iterations; with or without TCMalloc), we ran the simulator ten times, as ten independent job submissions, each time allocating a different set of nodes. We then created 1000 samples out of these ten observations by resampling with replacement using the empirical bootstrap technique.[31] Each one of the bootstrap samples has the same amount of observations (10) as the original sample. We used the bootstrap samples to obtain mean estimates of performance indicators for the simulator with and without TCMalloc, as well as 95% confidence intervals for these estimates using the standard error method.

**TABLE 2** Number of allocations and amount of allocated memory per phase

| Phase | Small allocations | Mid-sized allocations | Large allocations |
|---|---|---|---|
| split | $1.26 \times 10^6$ – 87.18 MiB | 21 – 13.27 MiB | 1 – 1.66 MiB |
| local | $136.67 \times 10^6$ – 7.376 GiB | $4.8 \times 10^3$ – 2.666 GiB | $7.3 \times 10^3$ – 7.483 GiB |
| reduce | $44.0 \times 10^3$ – 3.45 MiB | 45 – 20.47 MiB | 24 – 4.05 GiB |
| post | $36.2 \times 10^3$ – 84.88 MiB | 0 | 0 |

## 4.2 | TCMalloc primer

In this section, we provide a quick introduction to the essential notions of the TCMalloc heap allocator. TCMalloc is mainly based on local memory caches. TCMalloc proposes two types of local caches: (i) per-thread caches; and (ii) per-core caches. TCMalloc uses different allocation size classes and store the per-thread/per-core cache as a linked list for each of the size classes. The main advantage of local caches for dynamic memory allocation is to reduce the synchronization cost if objects can be allocated in the local cache, since concurrent allocations can be performed with few or no interactions between computing threads. TCMalloc distinguishes small objects (<256KB) from large ones, and small objects are preferably allocated in the local caches. This choice is particularly interesting for multithreaded applications that allocate many small objects. The programmer has the possibility to configure the size of the local caches in TCMalloc by calling the functions `SetMaxTotalThreadCacheBytes`(for per-thread caches) or `SetMaxPerCpuCacheSize` (for per-core caches). The larger the local cache, the higher the memory consumption and the smaller the execution time of the application. By default, these local caches are of 32MB. We use this size for our experiments with TCMalloc.

## 4.3 | Identification and characterization of anomalies in the small use case

We applied the method in the smaller use case described above, iterating 3 times to tackle different memory allocation anomalies found in the MHM simulator:

- Use of temporary objects that dynamically allocate memory for short periods of time (the example depicted in Section 3.4): characterized when we did a detailed profiling over allocations of 12 bytes;
- Use of STL C++ class `std::vector<>` to store matrices as vectors-of-vectors (c.f., member `StdFiniteElementSpace::mDoF` in Listing 2): characterized when we did a detailed profiling over allocations of 24 bytes;
- Use of statements like `objData = objData*otherData` (when `objData` internally allocate heap blocks) instead of `objData *= otherData`: characterized when we did a detailed profiling over allocations of 16 bytes.

In Figure 4, we illustrate the overheads of the heap management during the execution of the small use case before the application of the method ("Not optimized" in the figure), and after each of the three iterations of our method.

In Figure 4(A), we show the cumulative number of allocations of small size made throughout the simulations, as collected by our tool. This number decreases monotonically after each iteration of our method. This reduction—of about 30% at the end of the last iteration—contributes to decrease the time overhead imposed by the heap management. Nevertheless, the actual reduction in the overall execution time of the simulation is difficult to measure because of the size of the small use case. We therefore postpone to the following subsection the demonstration of this reduction with the larger use case.

In Figure 4(B), we show the maximum number of *extra heap bytes* allocated throughout the simulations, as collected by the Massif tool.[11] This number represents the amount of bytes allocated in excess of what the application asked for, and can be caused: (i) by the administrative bytes associated with each heap block; or (ii) by allocators rounding up the number of bytes asked for, to ensure suitable alignment within the heap block.

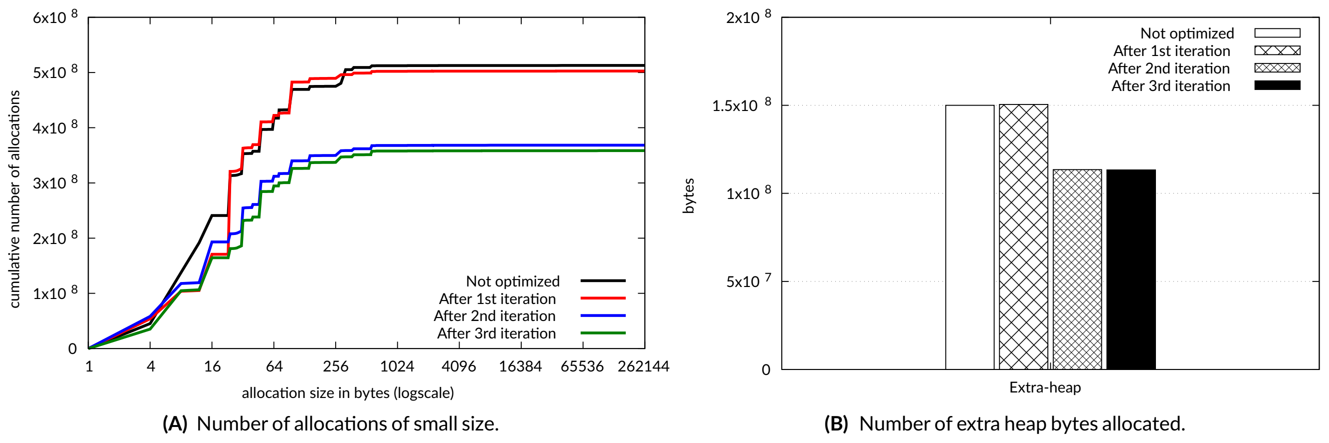
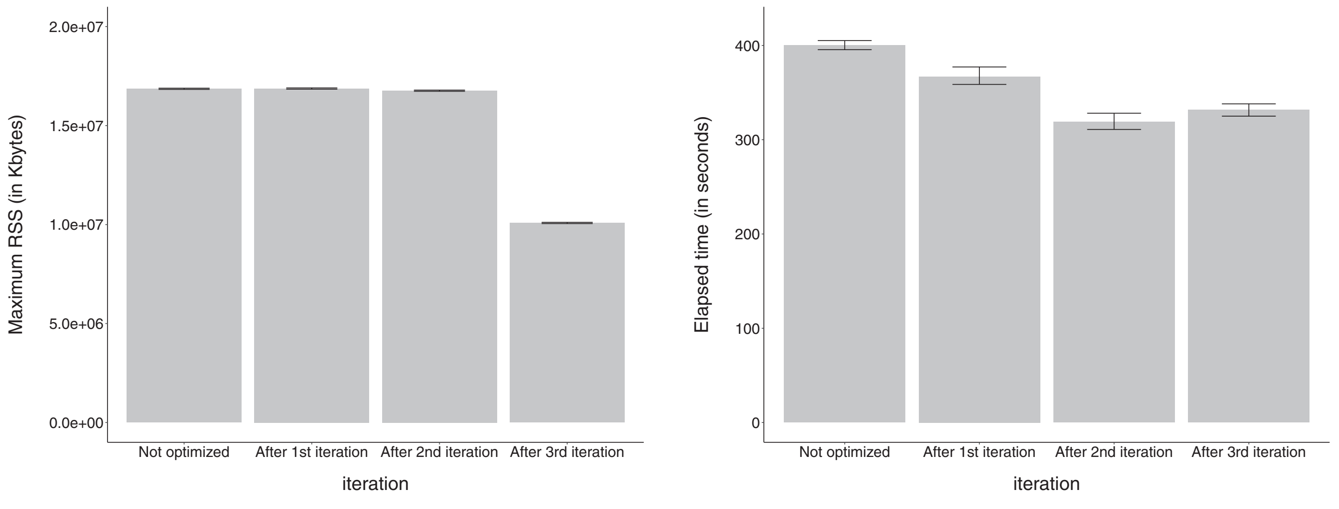


FIGURE 4 Overhead of the heap management. (A) Number of allocations of small size. (B) Number of extra heap bytes allocated



**(A)** Maximum resident set size (RSS).

**(B)** Total execution time.

**FIGURE 5** Performance of simulator. (A) Maximum resident set size (RSS). (B) Total execution time

The difference of about 15% at the end of the last iteration demonstrates that our method can also reduce the space overhead imposed by the heap management, allowing larger simulations to take place in a same computational resource.

The reader may observe that the first and third iterations had no direct effect on the small use case. In the first iteration, there was a "migration" of anomalies from 12-byte to 24-byte allocations in the phase of local problems: the amount of 12-byte allocations in that phase felt from $53.88 \times 10^6$ to $1.36 \times 10^6$, but the amount of 24-byte allocations in the same phase increased from $72.36 \times 10^6$ to $150.10 \times 10^6$. The second iteration then slashed the amount of 24-byte allocations in that phase to $14.64 \times 10^6$, without significant migrations to other allocation sizes, thus contributing to the overall reduction in the measured overheads. In the third iteration, it appears not to have been the best of allocation size to tackle for this specific use case: it reduced the amount of 16-byte allocations from $73.82 \times 10^6$ to $57.95 \times 10^6$. Nevertheless, we observe that these two iterations did have an impact in the large use case described in the following subsection.

## 4.4 | Impact of anomaly fixing in the large use case

In Figure 5, we show the performance of the MHM simulator for the large use case, using the GBLIC heap allocator, before the application of the method ("Not optimized" in the figure), and after each of the three iterations of our method. We use the following performance indicators: (i) resident set size (RSS); and (ii) overall execution time.

We collected these indicators with the accounting tool of the cluster's resource manager (Slurm[6]).

The maximum RSS represents the maximum memory footprint over all the four MPI ranks in the simulation. Figure 5(A) shows a reduction in the maximum RSS by 40.19% on average, after the three iterations of our method. Figure 5(B) shows a reduction in the total execution time by 17.00% on average, after these same iterations.

Notice that the last iteration of the method considerably reduced the total amount of memory demanded during the simulation, even if without changing in a statistically significant way its overall execution time in comparison with the second iteration. Explaining the reasons why some iterations do not improve some of the results is much harder for the large use case because of the cost of profiling, but we believe these reasons are related with the ones described in Section 4.3.

For each of the four configurations, Figure 6 (respectively, Figure 7) shows the plots for the maximum RSS (respectively, total execution time) estimate using empirical bootstrap resampling. They approximate a normal distribution, with the mean of the original sample close to the center of the histogram (the dotted vertical line in the left panel) and the quantiles of the empirical distribution mostly coinciding with the quantiles of the standard normal distribution (the dotted diagonal line in the right panel). We can then regard the 95% confidence intervals in Figure 5 obtained with the standard error method as statistically meaningful.

[6] `sacct` tool. https://slurm.schedmd.com

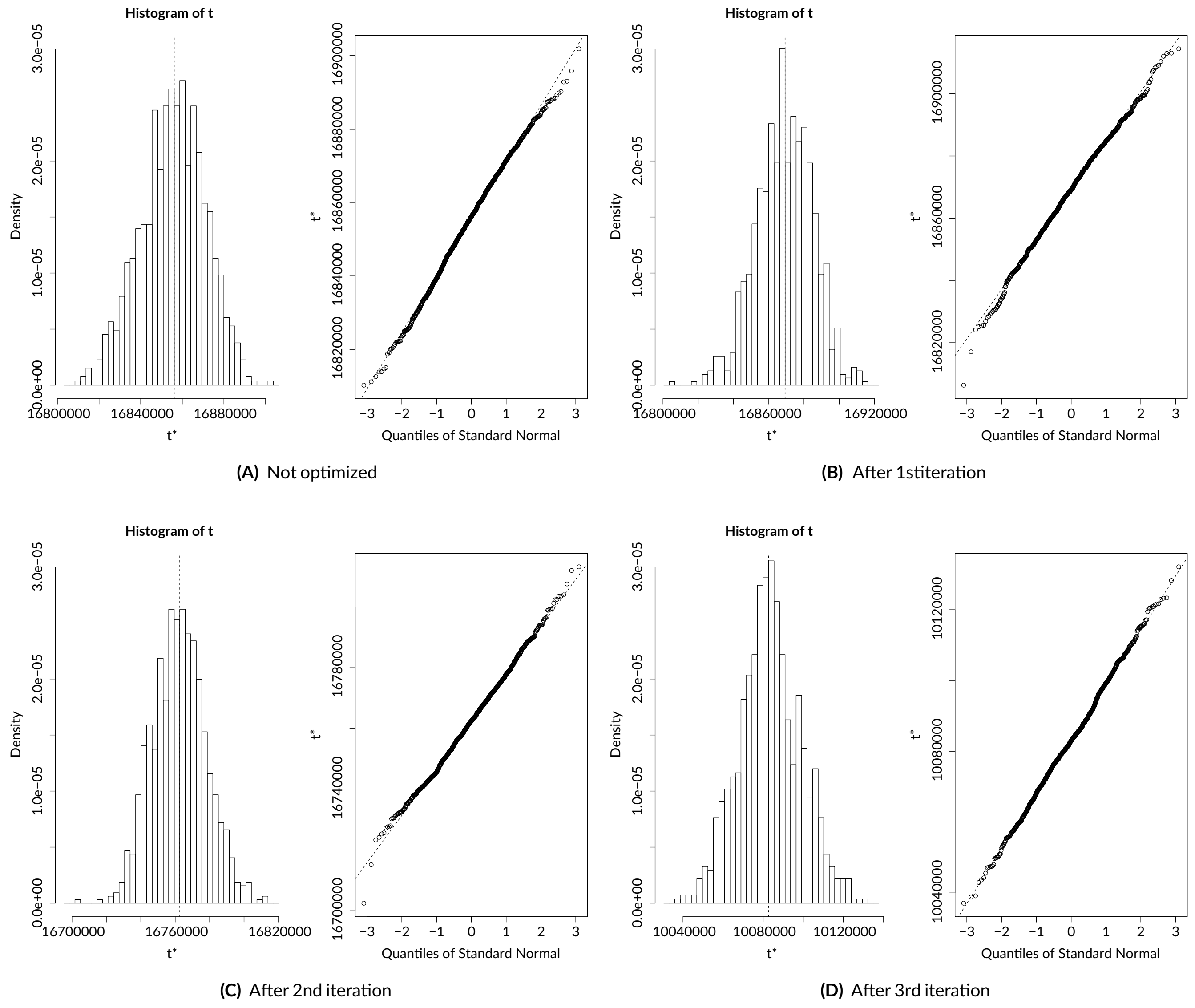


**FIGURE 6** Maximum RSS estimates using case resampling. (A) Note optimized. (B) After first iteration. (C) After second iteration. (D)After third iteration

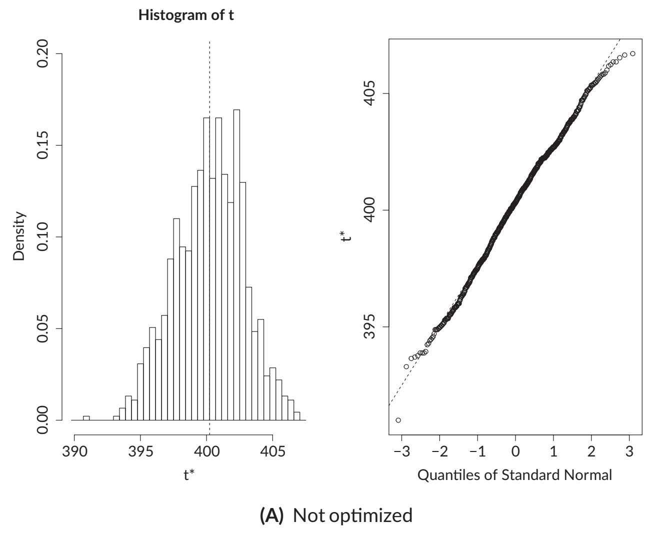
## 4.5 | Impact of anomaly fixing together with a high-performance allocator

In Figure 8, we show the performance of the MHM simulator for the large use case, using the TCMalloc heap allocator, before the application of the method ("Not optimized" in the figure), and after each of the three iterations of our method. We use the same performance indicators as in Section 4.4.

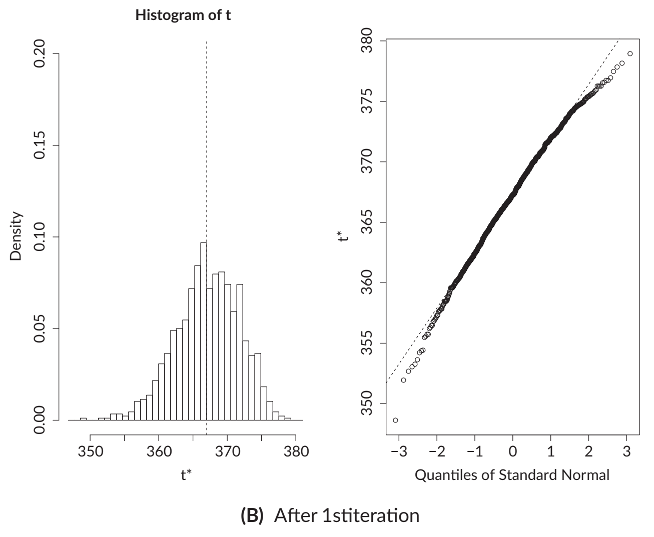
Figure 8(A) shows a reduction in the maximum RSS by 36.77% on average, after the three iterations of our method. Nevertheless, Figure 8(B) shows a reduction in the total execution time by only 4.80% on average, after these same iterations.

Again, notice that the last iteration of the method considerably reduced the total amount of memory demanded during the simulation, even if without changing in a statistically significant way its overall execution time in comparison with the second iteration. This is an interesting result, because it shows how our method may be complementary to the gains we can obtain in an HPC application by using a high-performance heap allocator such as TCMalloc. Nevertheless, like in Section 4.4, it is hard to explain in this experiment why some iterations do not improve some of the results because of the cost of profiling, but we believe the reasons have to do with the explanation in the last paragraph of Section 4.3.

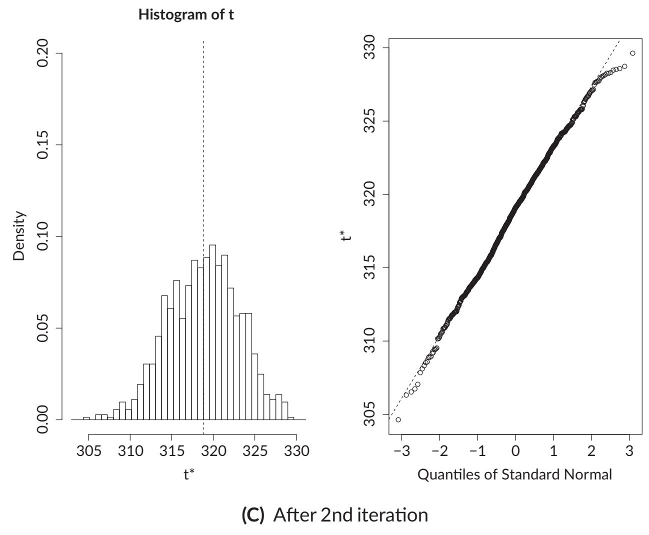
For each of the four configurations, Figure 9 (respectively, Figure 10) shows the plots for the maximum RSS (respectively, total execution time) estimate with TCMalloc using empirical bootstrap resampling. Again, they approximate a normal distribution, by the same reasons as those stated in Section 4.4. We can then regard the 95% confidence intervals in Figure 8 obtained with the standard error method as also statistically meaningful.



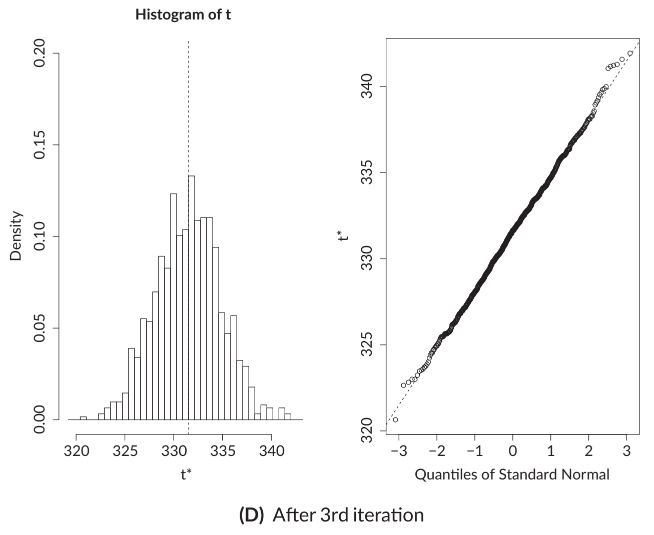
(A) Not optimized



(B) After 1stiteration

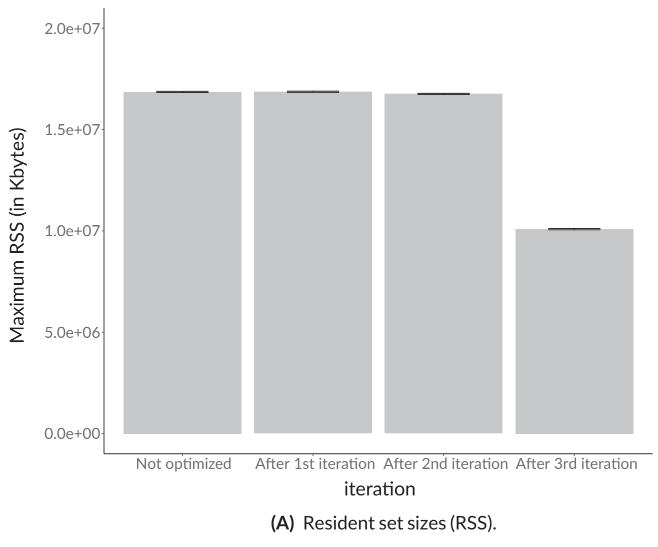


(C) After 2nd iteration

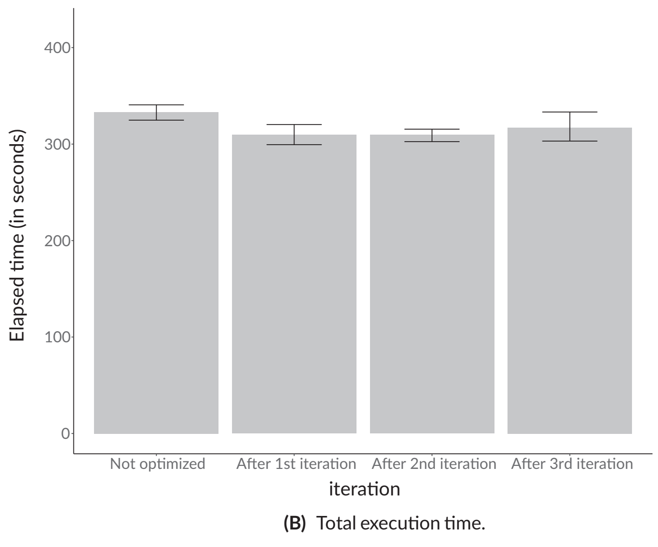


(D) After 3rd iteration

**FIGURE 7** Total execution time estimates using case resampling. (A) Note optimized. (B) After first iteration. (C) After second iteration. (D)After third iteration

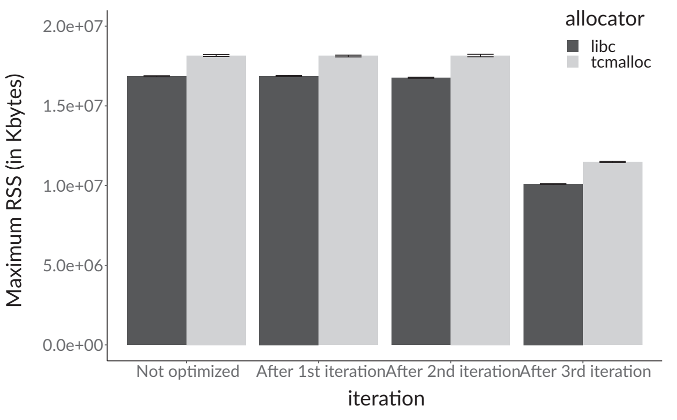


(A) Resident set sizes (RSS).

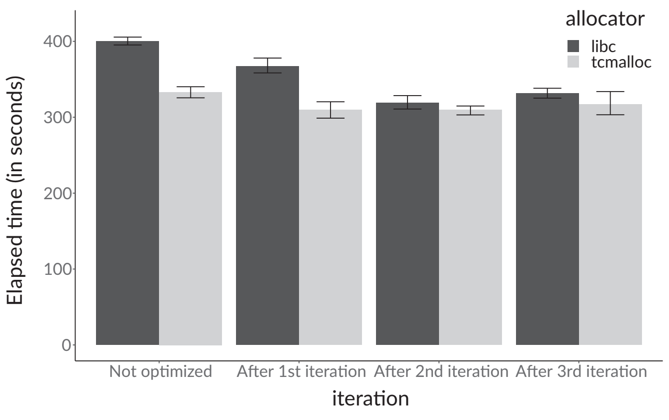


(B) Total execution time.

**FIGURE 8** Performance of simulator when using TCMalloc. (A) Resident set sizes (RSS). (B) Total execution time

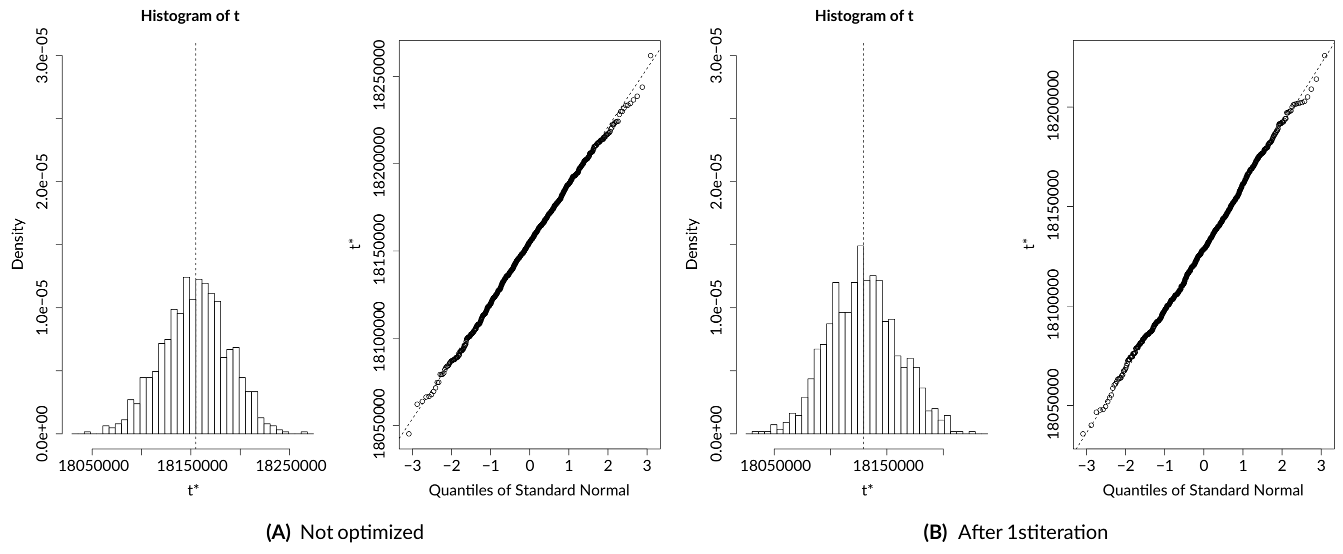


(A) Resident set sizes (RSS).



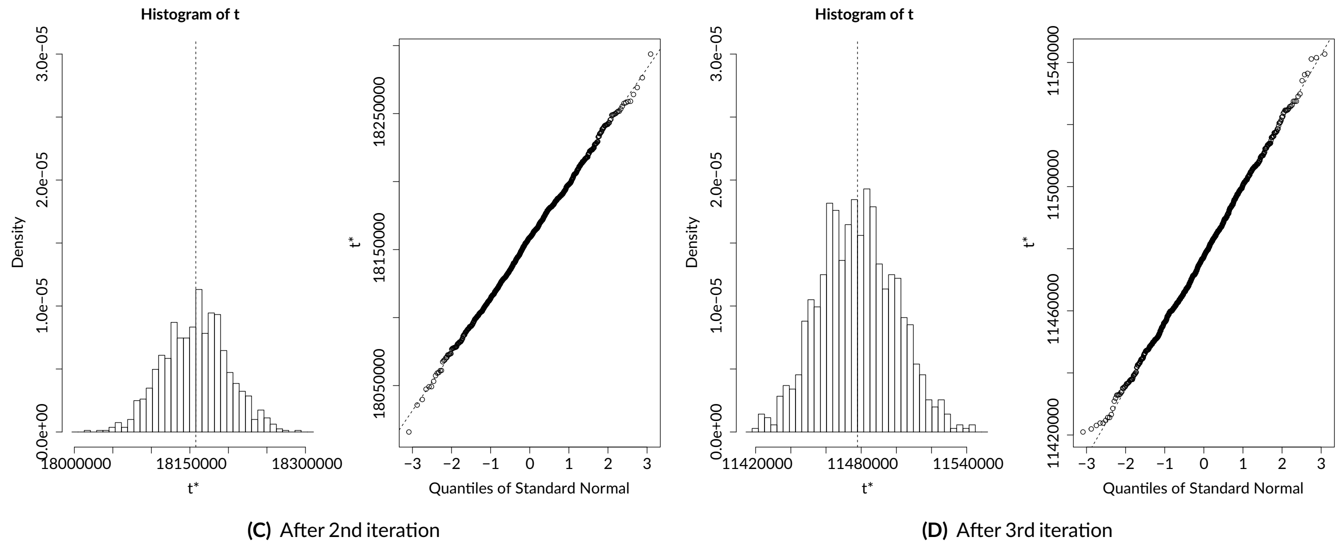
(B) Total execution time.

**FIGURE 9** Maximum RSS estimates with TCMalloc using case resampling. (A) Note optimized. (B) After first iteration. (C) After second iteration. (D) After third iteration



(A) Not optimized

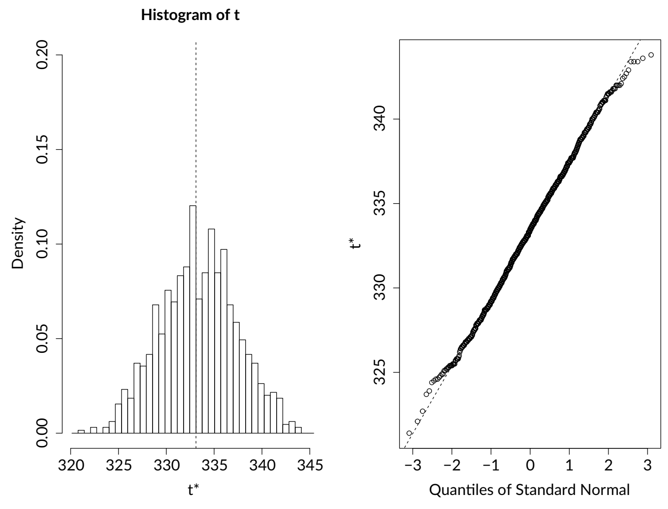
(B) After 1stiteration



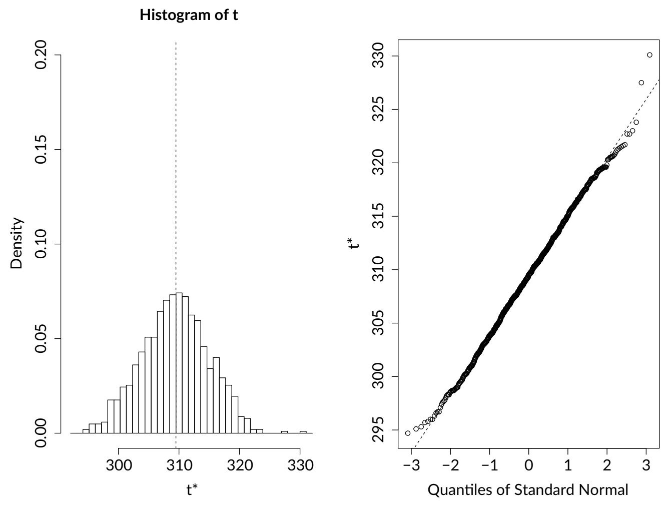
(C) After 2nd iteration

(D) After 3rd iteration

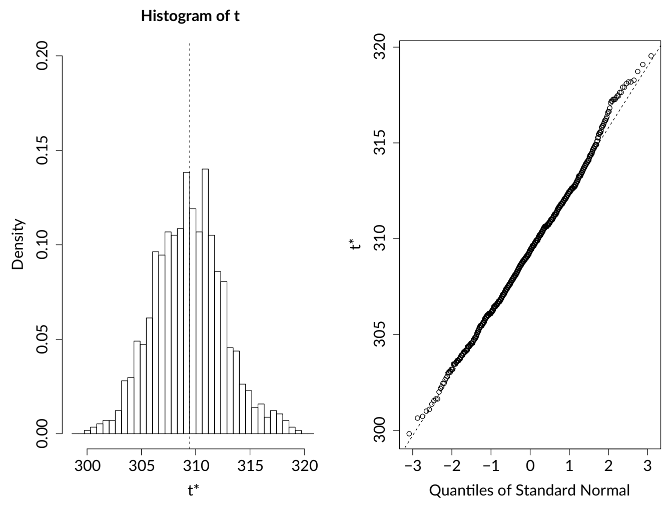
**FIGURE 10** Total execution time estimates with TCMalloc using case resampling. (A) Note optimized. (B) After first iteration. (C) After second iteration. (D) After third iteration



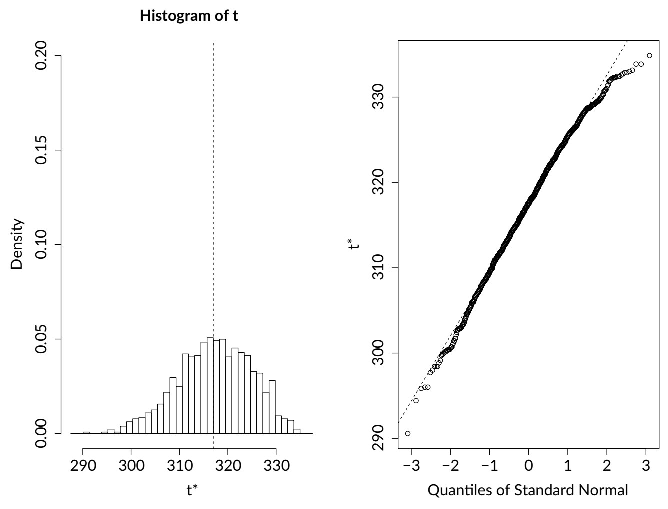
**(A)** Not optimized



**(B)** After 1stiteration



**(C)** After 2nd iteration



**(D)** After 3rd iteration

**FIGURE 11** Side-by-side performance comparison of simulator when using GLIBC and TCMalloc allocators. (A) Resident set sizes (RSS). (B) Total execution time

We finish this section by showing in Figure 11 a side-by-side comparison of the effects of our method on both GLIBC and TCMalloc heap allocators. Notice in the right panel of the figure that, as expected, the TCMalloc heap allocator helps with reducing the execution time of the application in comparison with the GLIBC heap allocator, even though the difference between the two allocators as regards this metric reduces as we employ the second iteration of our method. We believe this to be due to the impact of the second iteration of our method on the amount of small size allocations (c.f., Figure 4). In the left panel of the figure, however, we observe that the TCMalloc heap allocator consistently consumes more memory than the GLIBC heap allocator. This goes against the results Ferreira et al.[30] present. A possible reason for our observation is that TCMalloc reserves fixed-size local caches. A local cache allows TCMalloc to provide memory to an application faster, at the cost of left-over memory when rounding requests to the local cache size, specially for large allocations. Small allocations raise a different issue in TCMalloc; because it fits multiple allocations of a same size onto a single local cache, it may be the case that such cache will never become completely empty. Thus, TCMalloc may never return the memory consumed by the local cache to the system. A consequence of the above is that defining a proper local cache size in TCMalloc may be highly dependent on the application memory consumption pattern. We leave for future work a more thorough analysis of local cache sizes, including the impact of the per-thread/core cache types on the execution time and memory consumption of the MHM simulator.

## 5 | CONCLUSIONS AND FUTURE WORK

In this article, we have studied anomalies related to dynamic memory allocations. We have proposed an iterative method that allows a developer of HPC applications to detect, locate, and correct these allocation anomalies. We have successfully applied this method on a multiscale numerical simulator that allocated huge amounts of small chunks of memory. The results after three iterations of the method (taming allocations of 12, 24, and 16 bytes) show impressive gains in the number of calls to heap allocators—making the simulator run faster—and in the memory footprint—making the simulator capable of solving larger problems. Besides, we obtain these gains even when using a high-performance heap allocator such as TCMalloc. Our promising results may be regarded as indicators of a large potential impact of this method to larger application scenarios. In this sense, we are currently in the process of applying this method to other C/C++ applications from Petrobras (the Brazilian oil company), which partially funded this research.

The method we have proposed requires some profiling features to help the developer quickly detect and correct anomalies. We have developed a tool that offers a trade-off between obtaining precise information with a manageable cost of instrumentation and collection of profiling traces. As future work, we plan to study the integration of these features to existing memory profiling tools that are planned to be extensible, such as Valgrind/Massif.[11] We also plan to develop a post-processing tool that takes the stack trace types and generates call graphs like the one in Figure 3automatically.

In this article, the size of the allocations was the main criterion for optimization. It aimed at minimizing the unnecessary allocation of temporary objects, or aggregating the small allocations in bigger ones, or both. As future work, we plan to study other optimization criteria, such as the lifetime of dynamically allocated memory chunks. We believe a memory chunk with an extremely short lifetime should be allocated on the stack rather than on the heap. Another criterion, complementary to the lifetime, is the number of times a heap block is actually accessed and used. Assessing these new types of anomalies may require new approaches to the anomaly fixing. Higher level languages such as C++ have operator overloading capabilities that may help to correct these anomalies.

As a final remark, we have evaluated our method in a somewhat limited scenario, involving a single programming language and two heap allocators. As future work, we plan to analyze other heap allocators commonly used with multithreaded C/C++ applications, such as jemalloc,[32] and also extend our method to applications developed in other higher-level, VM-based languages. In particular, Python is beginning to be used widely in the context of HPC. In this respect, Python offers a `memory_profiler` module[33] that finely analyzes the allocation and release of memory; the tools we have developed for our method would therefore have to be adapted to the Python runtime.

### ACKNOWLEDGEMENTS

This work has been partially funded by CNPq, LNCC/MCTIC, and Petrobras (the Brazilian oil company). The authors acknowledge LNCC for providing HPC resources of the SDumont supercomputer, which have contributed to the research results reported within this article. http://sdumont.lncc.br. The authors also thank Yliès Falcone (UGA, LIG) for his comments that greatly improved the article.

### DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon reasonable request.

### ORCID

*Antônio Tadeu A. Gomes* https://orcid.org/0000-0002-0746-4014
*Roberto P. Souto* https://orcid.org/0000-0002-2081-0751
*Jean-François Méhaut* https://orcid.org/0000-0003-1047-7462

### REFERENCES

1. Appelbe B, Bergmark D. Software tools for high performance computing: survey and recommendations. *Sci Program*. 1996;5:239-249. https://doi.org/10.1155/1996/468929.
2. Gropp WD, Lumsdaine A. Parallel tools and environments: a survey. In: Heroux MA, Raghavan P, Simon HD, eds. *Parall Process Sci Comput*. Philadelphia, PA: SIAM; 2006:223-232.
3. Servat H, Llort G, Huck K, Gimenez J, Labarta J. Framework for a productive performance optimization. *Parall Comput*. 2013;39:336-353. https://doi.org/10.1016/j.parco.2013.05.004.
4. Supalov A, Semin A, Klemm M, DahnKen C. *Optimizing HPC Applications with Intel Cluster Tools: Hunting Petaflops*. New York, NY: Apress; 2014.
5. Kirk BS, Peterson JW, Stogner RH, Carey GF. libMesh: a C++ library for parallel adaptive mesh refinement/coarsening simulations. *Eng Comput*. 2006;22(3–4):237-254. https://doi.org/10.1007/s00366-006-0049-3.
6. Arndt D, Bangerth W, Clevenger TC, et al. The deal.II library, version 9.1. *J Numer Math*. 2019;27(4):203–213. https://doi.org/10.1515/jnma-2019-0064.
7. Logg A, Wells GN, Hake J. DOLFIN: a C++/Python finite element library. In: Logg A, Mardal KA, Wells G, eds. *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Berlin, Heidelberg/Germany: Springer; 2012:173-225.
8. Rathgeber F, Ham DA, Mitchell L, et al. Firedrake: automating the finite element method by composing abstractions. *ACM Trans Math Softw*. 2016;43(3):24:1-24:27. https://doi.org/10.1145/2998441.

9. Gomes ATA, Pereira WS, Valentin F, Paredes D. On the implementation of a scalable simulator for multiscale hybrid-mixed methods. *CoRR*. 2017;abs/1703.10435. http://arxiv.org/abs/1703.10435.
10. Ghemawat S. Gperftools heap profiler. 2019. https://gperftools.github.io/gperftools/heapprofile.html.
11. Seward J, Nethercote N, Weidendorfer J. *Valgrind 3.11 Reference Manual*. Surrey, UK: Samurai Media Limited; 2015.
12. Gomes ATA, Molion E, Souto RP, Méhaut JF. Identification and characterization of memory allocation anomalies in high-performance computing applications. Paper presented at: Proceedings of 20th Simpósio em Sistemas Computacionais de Alto Desempenho. UFMS. SBC; 2019:1-12; Porto Alegre, RS, Brazil.
13. Ghemawat S, Menage P. TCMalloc: thread caching Malloc; 2007. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.
14. GNU Developer community The GNU C Library (glibc); 2019. https://www.gnu.org/software/libc.
15. Gloger W. ptmalloc: Pthreads Malloc; 2006. http://www.malloc.de/en/.
16. Lea D. dlmalloc: a memory allocator; 1996. https://github.com/ennorehling/dlmalloc.
17. Berger ED, McKinley KS, Blumofe RD, Wilson PR. Hoard: a scalable memory allocator for multithreaded applications. Paper presented at: Proceedings of the 9th International Conferences on Architectural Support for Programming Languages and Operating Systems; 2000:117-128; New York, NY, MIT, ACM.
18. Kukanov A, Voss MJ. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technol J*. 2007;11(04):309-322. https://doi.org/10.1535/itj.1104.05.
19. Belady LA, Nelson RA, Shedler GS. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun ACM*. 1969;12(6):349-353. https://doi.org/10.1145/363011.363155.
20. Hastings R, Joyce B. Purify: fast detection of memory leaks and access errors. Paper presented at: Proceedings of the Winter USENIX Technical Conference. University of California. USENIX; 1992:125-136; Berkeley, CA.
21. Boehm H. Dynamic memory allocation and garbage collection. *Comput Phys*. 1995;9:297-393.
22. Novark G, Berger ED, Zorn BG. Efficiently and precisely locating memory leaks and bloat. Paper presented at: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. Trinity College Dublin; 2009:397-407; New York, NY, ACM.
23. Andrzejak A, Eichler F, Ghanavati M. Detection of memory leaks in C/C++ code via machine learning. Paper presented at: Proceedings of the 9th International Workshop on Software Aging and Rejuvenation. Hiroshima University. IEEE Computer Society; 2017:252-258; Washington, DC.
24. Wadler P. Fixing some space leaks with a garbage collector. *Softw Pract Exper*. 1987;17(9):595-608. https://doi.org/10.1002/spe.4380170904.
25. Mitchell N. Leaking space. *Queue*. 2013;11(9):10:10-10:23. https://doi.org/10.1145/2538031.2538488.
26. Guo C, Zhang J, Zhang Z, Zhang Y. Characterizing and detecting resource leaks in android applications. Paper presented at: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. University of California; 2013:389-398; New York, NY, IEEE/ACM.
27. Kukunas J. Intel VTune amplifier. In: Kukunas J, ed. *Power and Performance: Software Analysis and Optimization*. Amsterdam, The Netherlands: Elsevier; 2015.
28. Intel Corporation Intel Inspector XE; 2019. http://software.intel.com/en-us/intel-inspector-xe/.
29. Araya R, Harder C, Paredes D, Valentin F. Multiscale hybrid-mixed method. *SIAM J Numer Anal*. 2013;51(6):3505-3531. https://doi.org/10.1137/120888223.
30. Ferreira TB, Matias R, Macedo A, Araujo LB. An experimental study on memory allocators in multicore and multithreaded applications. Paper presented at: 12th International Conference on Parallel and Distributed Computing, Applications and Technologies. Seoul National University of Science and Technology. IEEE Computer Society; 2011:92-98; Washington, DC.
31. Davison AC, Hinkley DV. *Bootstrap Methods and Their Application*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge, UK: Cambridge University Press; 1997.
32. Evans JA. Scalable concurrent Malloc(3) implementation for FreeBSD. Paper presented at: Proceedings of the Technical BSD Conference. University of Ottawa. BSDCan; 2006; Ottawa, Ontario, Canada.
33. Pedregosa F, Gervais P. Python memory profiler; 2019. https://pypi.org/project/memory-profiler/.

**How to cite this article:** Gomes ATA, Molion E, Souto RP, Méhaut J-F. Memory allocation anomalies in high-performance computing applications: A study with numerical simulations. *Concurrency Computat Pract Exper*. 2021;33:e6094. https://doi.org/10.1002/cpe.6094