Identification and Characterization of Memory Allocation Anomalies in High-Performance Computing Applications *

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

[1] Laboratório Nacional de Computação Científica (LNCC)
Petrópolis-RJ, 25651-075, Brazil

[2]École Polytechnique (Polytech Grenoble)
Université Grenoble Alpes, 38000 Grenoble, France

[3]Université Grenoble Alpes, CNRS, Grenoble INP
LIG, 38000 Grenoble, France

{atagomes,rpsouto}@lncc.br

enzo.molion.0@gmail.com

jean-francois.mehaut@univ-grenoble-alpes.fr

Abstract. *A memory allocation anomaly occurs when the allocation of a set of heap blocks imposes an unnecessary overhead on the execution of an application. In this paper, 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 the memory footprint of its processes by* 37.27% *and the execution time by* 16.52%*. We conclude that the developer of high-performance computing applications can benefit from the method and tool during the software development cycle.*

1. Introduction

Researchers and practitioners have long worked on the performance analysis and optimization of high-performance computing applications [Appelbe and Bergmark 1996, Gropp and Lumsdaine 2006, Servat et al. 2013, Supalov et al. 2014]. 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).

In this paper 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++ [Kirk et al. 2006, Arndt et al. 2019] or dynamic languages such as Python [Logg et al. 2012, Rathgeber et al. 2016] is increasingly common in software libraries that support the development of these simulators.

*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 paper. URL: http://sdumont.lncc.br. The authors also thank Yliès Falcone (UGA, LIG) for his comments that greatly improved the paper.

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.

As a first contribution of this paper, 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 [Gomes et al. 2017]. 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. We show that taming these anomalies in a MSL simulator reduces its memory footprint by 37.27% and its execution time by 16.52%.

As a second contribution of this paper, we present a tool we have developed to support our method. Notice that some available tools (e.g., the Google Heap Profiler [Ghemawat 2019] and Valgrind/Massif [Seward et al. 2015]) 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.

The remainder of this paper 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) [GNU Developer community 2019] as their standard C runtime library. Other allocators are also available such as Hoard [Berger et al. 2000], TCMalloc [Ghemawat and Menage 2007], and TBB Malloc [Kukanov and Voss 2007]. The performance of these allocators may

[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.

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.

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

The study on the Belady anomaly [Belady et al. 1969] 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* [Hastings and Joyce 1992, Boehm 1995] have been the main memory issue studied by the scientific community [Novark et al. 2009, Andrzejak et al. 2017]. 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 Subsection 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 [Wadler 1987] 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 [Mitchell 2013], or Java for embedded applications in [Guo et al. 2013]). 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 high-performance computing 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. 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 [Ghemawat 2019] and the GNU Libc mtrace[5] 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 [Kukunas 2015] and Valgrind/Massif [Seward et al. 2015] 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 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.

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

3. Method and Tool for Taming Allocation Anomalies

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 summarize the steps of our method 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.

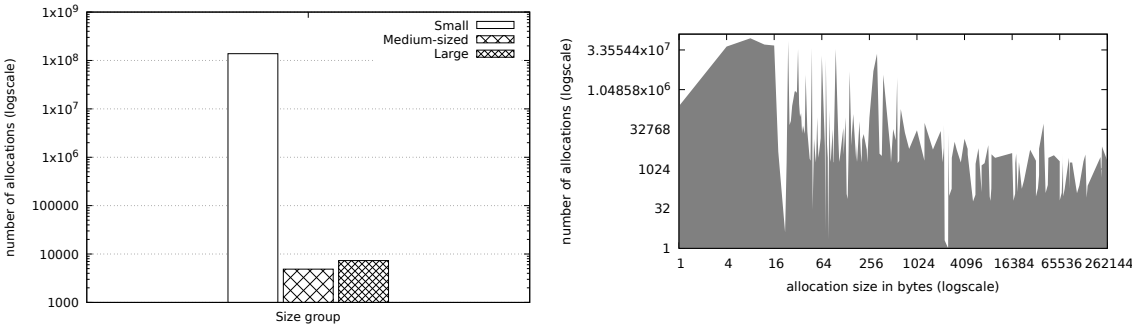
Example: we illustrate the use of the method with a simulator implemented with MSL. More specifically, we explore the Multiscale Hybrid-Mixed (MHM) finite-element method available in this set of libraries [Araya et al. 2013]. The MHM method is interesting for presenting our method because it is composed 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 Fig. 1 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 3 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 Fig. 1a that the number of allocations of the first group is much larger than the other two. Besides, as we show in Fig 1b, 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. In the following we detail the steps of our method that tackle the anomalies related to these small allocations.

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.



(a) Number of allocations per size group.

(b) Number of small-size allocations.

Figure 1. Number and size of dynamic memory allocations.

The developer can use our tool to identify these parts more efficiently by instrumenting specific code regions (see Subsection 3.6). With our method and tool the developer may instrument regions at the application's main function only. Taking for example the MHM simulation presented at the beginning of this section, the developer can instrument each of the 4 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 4 phases of the simulation.

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 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

Table 1. Number of allocations per allocation size (in bytes).

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

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. The developer may use this number to select the most pertinent stack trace types for the next step of the method.

3.3. Characterization of anomalies

To characterize an anomaly, the developer takes all the stack trace types selected in the previous step, 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.

In Fig. 2 we show a snip of the call graph obtained from Listing 1. The entries in the stack trace types that Fig. 2 illustrates are in boldface 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.

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
-----------
```

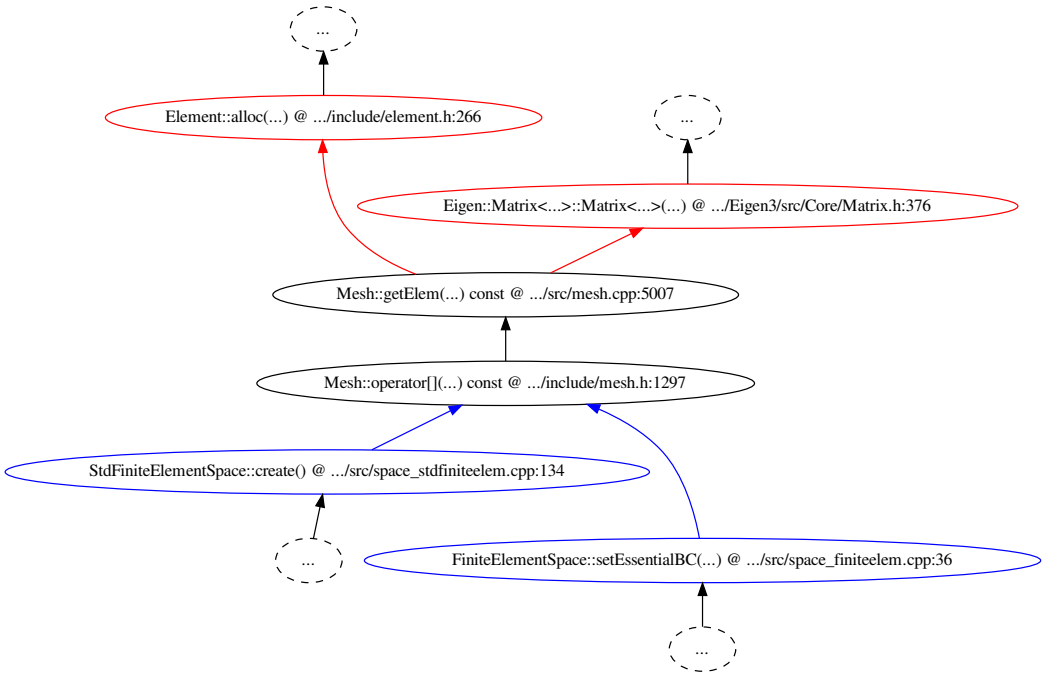


Figure 2. Call graph derived from the stack trace types in Listing 1.

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. In these cases, the cost of memory allocation and deallocation surpasses that of the actual computation.

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. 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 Subsection 4.2 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. Our tool is open source and available at: `https://gitlab.com/EnzoMolion/profiling-library`.

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 $4,096$ triangles at the global level. Within each element of the mesh, we solve a local problem composed of $1,137$ 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 Fig. 1a. 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). 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 $1,536$ 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.

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. 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:

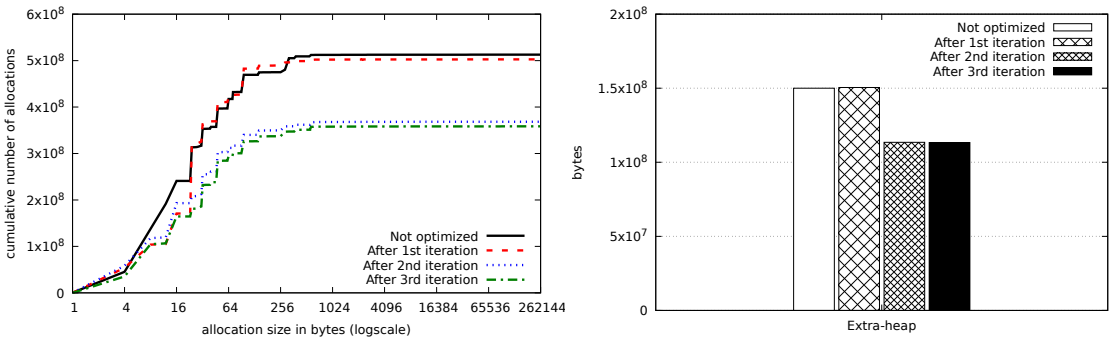
- Use of temporary objects that dynamically allocate memory for short periods of time (the example depicted in Subsection 3.4): characterized when we did a detailed profiling over allocations of 12 bytes;
- Use of STL C++ class `std::vector<>` to store matrices as vectors-of-vectors: characterized when 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 Fig. 3 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 3 iterations of our method.

In Fig. 3a 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 Fig. 3b we show the maximum number of *extra heap bytes* allocated throughout the simulations, as collected by the Massif tool [Seward et al. 2015]. 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. 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



(a) Number of allocations of small size. **(b)** Number of extra heap bytes allocated.

Figure 3. Overhead of the heap management.

in that phase felt from 53.88×10^6 to 1.36×10^6, but the amount of 24-byte allocations in the same phase increased from 72.36×10^6 to 150.10×10^6. The second iteration then slashed the amount of 24-byte allocations in that phase to 14.64×10^6, 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 73.82×10^6 to 57.95×10^6. Nevertheless, we observe that these two iterations did have an impact in the large use case described in the following subsection.

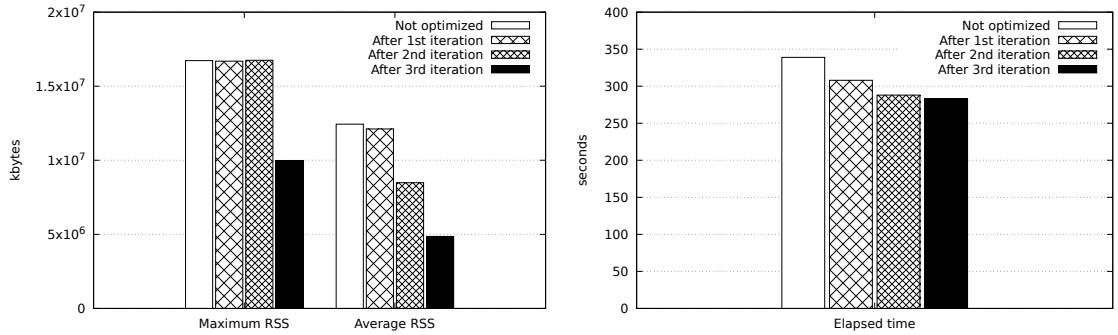
4.3. Impact of anomaly fixing in the large use case

In Fig. 4 we show the performance of the MHM simulator for the large use case before the application of the method ("Not optimized" in the figure), and after each of the 3 iterations of our method.

We use the following performance indicators: (i) resident set size; and (ii) overall execution time. We collected these indicators with the accounting tool of the cluster's resource manager. The maximum (resp., average) resident set size represents the maximum (resp., average) memory footprint over all the 4 MPI ranks in the simulation. Figure 4a shows a reduction in the maximum resident size by 37.27%, and in the average resident size by 58.18%. Figure 4b shows a reduction in the execution time by 16.52%. Notice that the last iteration of the method considerably reduced the total amount of memory demanded during the simulation, even if with only a slight decrease in its overall execution time. Explaining the reasons why some iterations do not improve some of the results is much harder for the large use case because of cost of profiling, but we believe these reasons are related with the the ones described in Subsection 4.2.

5. Conclusions and future work

In this paper, we have studied anomalies related to dynamic memory allocations. We have proposed an iterative method that allows a developer of high-performance computing 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 3 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



(a) Resident set sizes (RSS). **(b)** Total execution time.

Figure 4. Performance of simulator.

capable of solving larger problems. We intend to apply this method on other applications, especially from Petrobras (the Brazilian oil company).

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 [Seward et al. 2015]. Besides, we intend to evaluate our method against other, more scalable heap allocators such as TCMalloc and TBB Malloc. The aim is to assess the extent to which applications linked to these allocators may also benefit from the application of our method.

In this paper, 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.

References

Andrzejak, A., Eichler, F., and Ghanavati, M. (2017). Detection of memory leaks in C/C++ code via machine learning. In *9th International Workshop on Software Aging and Rejuvenation (WoSAR 2017)*, pages 252–258, Toulouse, France.

Appelbe, B. and Bergmark, D. (1996). Software tools for high performance computing: Survey and recommendations. *Scientific Programming*, 5:239–249.

Araya, R., Harder, C., Paredes, D., and Valentin, F. (2013). Multiscale hybrid-mixed method. *SIAM Journal on Numerical Analysis*, 51(6):3505–3531.

Arndt, D., Bangerth, W., Clevenger, T. C., Davydov, D., Fehling, M., Garcia-Sanchez, D., Harper, G., Heister, T., Heltai, L., Kronbichler, M., Kynch, R. M., Maier, M., Pelteret, J.-P., Turcksin, B., and Wells, D. (2019). The `deal.II` library, version 9.1. *Journal of Numerical Mathematics*.

Belady, L. A., Nelson, R. A., and Shedler, G. S. (1969). An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353.

Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. (2000). Hoard: A scalable memory allocator for mumtithreaded applications. In *9th International Conferences on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, Cambridge, MA, USA.

Boehm, H. (1995). Dynamic memory allocation and garbage collection. *Computers in Physics*, 9:297–393.

Ghemawat, S. (2019). Gperftools Heap Profiler. `https://gperftools.github.io/gperftools/heapprofile.html`.

Ghemawat, S. and Menage, P. (2007). TCMalloc: Thread caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

GNU Developer community (2019). The GNU C library (glibc). `https://www.gnu.org/software/libc`.

Gomes, A. T. A., Pereira, W. S., Valentin, F., and Paredes, D. (2017). On the implementation of a scalable simulator for multiscale hybrid-mixed methods. *CoRR*, abs/1703.10435.

Gropp, W. D. and Lumsdaine, A. (2006). *Parallel Tools and Environments: A Survey*, chapter 12, pages 223–232. SIAM.

Guo, C., Zhang, J., Zhang, Z., and Zhang, Y. (2013). Characterizing and detecting resource leaks in Android applications. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE'2013)*, pages 389–398, Palo Alto, CA, USA.

Hastings, R. and Joyce, B. (1992). Purify: Fast detection of memory leaks and access errors. In *Winter USENIX Conference*, pages 125–136, San Francisco, CA, USA.

Kirk, B. S., Peterson, J. W., Stogner, R. H., and Carey, G. F. (2006). `libMesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3–4):237–254.

Kukanov, A. and Voss, M. J. (2007). The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(04):309–322.

Kukunas, J. (2015). Intel VTune Amplifier. In Kukunas, J., editor, *Power and Performance: Software Analysis and Optimization*. Elsevier.

Logg, A., Wells, G. N., and Hake, J. (2012). DOLFIN: a C++/Python finite element library. In Logg, A., Mardal, K.-A., and Wells, G., editors, *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*, pages 173–225. Springer, Berlin, Heidelberg.

Mitchell, N. (2013). Leaking space. *Queue*, 11(9):10:10–10:23.

Novark, G., Berger, E. D., and Zorn, B. G. (2009). Efficiently and precisely locating memory leaks and bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, pages 397–407, Dublin, Ireland.

Rathgeber, F., Ham, D. A., Mitchell, L., Lange, M., Luporini, F., Mcrae, A. T. T., Bercea, G.-T., Markall, G. R., and Kelly, P. H. J. (2016). Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3):24:1–24:27.

Servat, H., Llort, G., Huck, K., Gimenez, J., and Labarta, J. (2013). Framework for a productive performance optimization. *Parallel Computing*, 39:336–353.

Seward, J., Nethercote, N., and Weidendorfer, J. (2015). *Valgrind 3.11 Reference Manual*. Samurai Media Limited.

Supalov, A., Semin, A., Klemm, M., and DahnKen, C. (2014). *Optimizing HPC Applications with Intel Cluster Tools: Hunting Petaflops*. Apress.

Wadler, P. (1987). Fixing some space leaks with a garbage collector. *Software: Practice and Experience*, 17(9):595–608.