Performance Analysis and Optimizations Techniques for Legacy Code Numerical Simulations

Federico J. Díaz and Fernando G. Tinetti[1]

III-LIDI, Fac. de Informática, Universidad Nacional de La Plata, Argentina
[1]Also with Comisión de Inv. Científicas de la Provincia de Buenos Aires
fernando@info.unlp.edu.ar

Abstract. Numerical simulations used today by scientists in various disciplines, are frequently based on implementations created when the predominant computing hardware was sequential by design. In this simulations, new features are added or updated, when new discoveries are made, but the computational implementation remains unchanged, not taking advantage of modern hardware architectures. This "legacy code" study cases, presents the opportunity to create a set of techniques and tools, oriented to perform optimizations from a computational and software engineering points of view. As an example, in conjunction with an astrophysics research group, a real-world case numerical integrator optimization is presented, were these techniques were applied, showing the results obtained.

Keywords: High Performance Computing, Optimization, Numerical integrators Legacy Numerical Software.

1 Introduction

Scientific disciplines often require complex numeric simulations to compute the models that describe real world/physical processes. The natural complexity of the real world requires that the simulations process large volumes of data, and perform computational-heavy calculations, in order to obtain the desired results.

The software used to compute these numerical simulations was often created in a time where the predominant computing technologies were sequential in nature. This software, today referred as "Legacy code" [1], is widely used amongst the scientific community.

Scientists update the implementation of the models, introducing new features as new discoveries are made in their respective disciplines. These new features frequently do not include modern optimization techniques, thus maintaining the sequential nature of the original implementation.

The shift of paradigm in current hardware design, moving away from sequential single-core processors to parallel and distributed computing creates a new opportunity to potential performance improvement of legacy code. But performing these optimizations requires a good knowledge on these mentioned parallel and distributed architectures, which is usually agnostic to the actual model implemented.

1.1 Performance Optimizations

We can mention 2 types of performance-oriented optimization categories, as follows:

1) **Automatic optimizations.** These optimizations are implemented through compiler options. A good example of them are the well-known optimizations flags -O[1..3], which provide a very easy way of obtaining good performance results [2]. Other examples are the automatic inlining of functions, using the flag *-inline* on the ifort intel Fortran compiler [3], or the shared-memory parallelization performed by the OpenMP library [4], using compiler directives to generate automated threads in for-cycles.
2) **Manual optimizations.** These optimizations, require some understanding of the underlying code structure, in order to successfully obtain good performance results, that don't affect the execution numerical results. These are non-trivial, and require a good amount of profiling and research in order to be completely done.

1.2 Software engineering improvements

One of the characteristics of a "legacy code" implementation, is defined by software that is still being used today, but that have not been updated to modern software paradigms. The most predominant feature of Fortran code, is the usage of the GOTO statement.

While some of the GOTO usage can be removed automatically (like, for example, when it is used to create a FOR-like structure) [5], there are other cases where it requires manual interaction. Removing GOTO statements from legacy code, should be considered a must, before performing optimizations, if these statements are related to the portion of the program that needs to be optimized.

2 Performance Analysis of a Real-World Case

An example of a numerical integrator that has "legacy code" embedded into its core functionality, is the Mercury [6] N-Body integrator, and widely used by planetary astronomers around the world. Mercury is completely developed in Fortran, integrating the SWIFT [7] libraries for numerical simulation, and performs calculations of close-encounters in bodies. The Mercury integrator, has the ability to perform computations with "small bodies" and "large bodies". The main difference between them, is that the small bodies don't produce interactions between them, only with a central star, and other large bodies, while the large bodies, do interact between each other, adding complexity to the simulation. The more "large bodies" used in the context, the more compute-intensive the simulation becomes. Hence, common simulations involve a mixture of large and small bodies, with hundreds of small bodies, and tens of large ones. For the performance analysis, the research group from the *Facultad de Ciencias*

Astronomicas y Geofisicas of the *Universidad Nacional de La Plata*, provided 2 real case scenarios, that used the simulator with 2 distinct execution paths. The first case was all small bodies, one large body, and the central star, and the second case, was a collection of large bodies, all interacting between each other.

2.1 Profiling Mercury

Initially, the GNU profiler gprof [8] was used to analyze the compute-intensive parts of the code, so that the key areas of Mercury to optimize were detected. There are some subprograms appearing in the top-ten most time consuming ones, for experiments with predominant "small bodies" and "large bodies". The rest of the top-ten most time-consuming subprograms depend on the kind of bodies being simulated. We also implemented a wall-clock like time metric. This allows to measure the real-world time execution experienced by a human observer, as the gprof output only measures processor timing, not taking into account external interferences. Comparing both approaches indicated a significant difference: it was discovered that the input/output frequency and volume should be optimized as well.

3 Applying Optimizations

After the performance metrics where obtained, a number of optimizations have been applied. For each optimization applied, the numerical result was carefully controlled, so that consistency was maintained. When we found a numerical difference, the results were sent back to the scientists' research group for approval. As a general method, it is always best to perform sequential optimizations first. Then, with the optimized code, move to implement parallel optimizations.

We applied several sequential optimizations (in critical subprograms) such as the automatic ones (-O2), I/O removal, removal of GoTo statements, and intrinsic operations replacement. We found that subprogram inlining along with operations reordering and redundant operations removal were the most successful in terms of providing performance enhancement.

The small bodies case was almost discarded for including parallel computing, because the small bodies do not interact, they only require a small amount of processing power to update their velocities and positions in each cycle. The large bodies case is particularly well suited for including parallel computing, as they have to interact between every other large body in the simulation to update their properties. Thus, the parallelization of the execution is a must to reduce the simulation time. In this case, a specific effort was made to eliminate data dependency computations for aiding the inclusion of parallel computing (e.g. via OpenMP further implementation).

4 Results and Future Works

Given the unoptimized initially legacy code, the sequential optimizations provided great performance gains. For the small bodies case, the performance gain was about 50% reduction in runtime. The large bodies case, was also parallelized, and the performance gain provided by the OpenMP implementation in 8 cores was about a 40% runtime reduction. We initially take all the optimization techniques as a guideline, given its application on a real-world numerical integrator. The potential of automatically apply some of them, has to be investigated further.

While the inline optimization is provided by some compilers (e.g. the Intel compiler), it cannot always be properly implemented. GoTo statements usually prevents the usage of many of the compilers optimizations, including inlining. As of now, there is no automated way of removing GoTo statements in general, but some of them do have a structure (e.g. GoTo statements used to replace For-like iteration structures).

The proper usage of cache memory, should be a topic of future research, with an increased number of bodies in the integrators. Also, the future line of work, of parallelizing further, using SIMD processors, like GP-GPU, could potentially increase the size of the input values, to use hundreds of thousands of elements.

References

1. Fernando G. Tinetti, Mariano Méndez, Armando De Giusti.: Restructuring Fortran legacy applications for parallel computing in multiprocessors. The Journal of Supercomputing, Volume 64, Issue 2, pp. 638-659.126 (2013).
2. GNU GCC Compiler Homepage, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, last accessed 2020/03/30.
3. Intel Fortran Compiler Homepage, https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-inline-forceinline-and-noinline, last accessed 2020/03/30.
4. OpenMP Architecture Review Board., *"OpenMP Application Programming Interface", Version 5.0, (2018).*
5. Mariano Méndez, Fernando G. Tinetti.: Change-driven development for scientific software, The Journal of Supercomputing, Springer, Volume 73, Issue 5, pp. 2229-2257, (2017).
6. Chambers, J.E; Migliorini, F., *"Mercury - A New Software Package for Orbital Integrations"*, Bull. American Astron. Soc. ,29, 1024. (1997).
7. *SWIFT* Homepage, http://www.boulder.swri.edu/~hal/swift.html last accessed 2020/03/30.
8. Jay Fenlason, *"GNU gprof manual" Homepage*, http://sourceware.org/binutils/docs/gprof/index.html, last accessed 2020/03/30