



Identifying performance bottlenecks in a radiative transfer application

S. Stephany¹, R. V. Correa², C. L. Mendes¹ & A. J. Preto¹

¹*Applied Computing Lab (LAC)*

²*Baloon Facility (SLB), Brazilian Institute for Space Research (INPE),
Brazil*

Abstract

The Hydrolight software is a well known radiative transfer solver, written in Fortran 77, intended for hydrologic optics problems. It was developed by C. D. Mobley and it is based on an invariant imbedding methodology that yields a system of non-linear ordinary differential equations, the Riccati equations, which are solved by routines that implement numerical methods.

This work proposes an innovative strategy for identifying performance bottlenecks and optimising a code using both standard profiling tools and accessing specific cpu registers and a time counter. The execution profile in a RISC workstation shows that most of the processing time is spent in solving Riccati equations, what is accomplished by a Runge-Kutta algorithm. The time-consuming routines were identified and a timing routine based on system wall time was used to analyse which are the critical parts of these routines. The code was then ported to an Intel monoproccessor, running under Linux, in order to be monitored by cpu event counters. Based on this detailed data, classical optimisations were performed in the related routines and the code was re-run and its performance re-evaluated. The optimised code was ported back to the RISC workstation and benchmarked again.

Another step was to port the time-intensive routines to Fortran 90 and check performance enhancements. We are currently parallelising the new code on a Beowulf-class system based on Pentium-Pro processors. The main technique that we are employing in this task is the insertion in the Fortran 90 code of High Performance Fortran (HPF) directives yielding a code suitable for both shared memory or distributed memory parallel systems.



1 Introduction

Many softwares were not designed aiming at performance, but at correctness and availability of standard tools. The need for performance may, however, arise due to new processing-intensive applications and the amount of effort to generate an entirely new software is generally prohibitive. Therefore, an interesting issue would be the partial optimisation of existing code, leading to the problem of choosing what has to be optimised and that is achieved by means of the identification of performance bottlenecks.

Operating systems provide standard tools for cpu and memory usage, but these programs lack precision and granularity considering the needs of close code monitoring. Another issue is the potential use of optimisation levels in the compilation. Optimising compilers are at the heart of the search for performance, being customized to the machine architecture, but it is not always clear what degree of optimisation is achieved. On the other side, standard free distribution compilers may only follow common sense techniques, not being bound to a specific hardware, leading to performance penalties.

This work analyses and optimises the performance of a radiative transfer software, the Hydrolight code [4], using an innovative approach. The code is run in the usual machine and standard profiling tools are used in order to identify time-consuming routines. Further analysis is performed by a timing function using system wall time in order to identify the more time consuming routines. Then, the code is ported to a probe machine where more accurate monitoring tools are available. This allows a fine tuning of these routines and the upgraded code is ported back to the original machine. Even considering that there are different environments, each one characterized by its own architecture, operating system and compiler, the approach provided good results. The original Fortran 77 code is used in Sparc Ultra monoprocessor workstations and was ported to a Linux driven Intel Pentium for the fine tuning.

Compiler optimisation options were exercised and cpu and memory performance observed in both environments. Performance bottlenecks were unveiled and most of the processing time was found to be spent by some loops in a couple of routines. Minor, though efficient, modifications were performed using classical code optimisation techniques and the performance of the modified code was re-evaluated for the different compiler options. A further step included porting to Fortran 90 [7] and repeating the cycle, still using monoprocessor machines. The latter code was then enhanced with HPF - High Performance Fortran [2] directives and run on a Linux driven multiprocessed Intel Pentium machine to check the multiprocessed performance.

The motivation of the current work was given by the application of the Hydrolight software in inverse problems aiming at the estimation of source



profiles [9] [11] and water inherent optical properties [10] [12] from radiometric measurements of underwater fields. In order to solve the inverse problem, an implicit inversion technique was employed, leading to an iterative process, in which each candidate solution yields a in-water light field by means of solving the *direct problem* using the Hydroligh software. An objective function is then calculated from the square differences between this field and the measured data leading to the *inverse problem* of minimising this function, what constitutes a nonlinear constrained optimisation problem. The optimisation algorithm provides successive approximations of the unknown profiles until the required precision is reached or some stopping condition is met.

This iterative process demands hundreds of executions of Hydrolight, precluding a timely completion of the several inversions and monopolising the machines used for these runs. Inverse problems are ubiquitous in science and engineering, and usually a direct model is already available and an implicit methodology is chosen. Standard optimisation algorithms can be used, but one difficulty lies in affording multiple runs of the direct problem. Hence, the Hydrolight software was taken as a case study and the current work proposes a sequence of procedures in order to optimise a given code.

2 A strategy for closer monitoring

A first step towards optimisation is to identify time-consuming routines, and a preliminary analysis of their code can detect inefficiencies to be eliminated by means of classical optimisation techniques. Given current typical optimising capability of most compilers, a question that naturally arises is what kind of optimisations would be performed automatically by selecting an appropriate optimisation level of the compiler.

However, answering that question demands more accurate tools in order to perform a comprehensive analysis of the time-consuming parts of the code. The use of a time measuring routine would interfere with the performance measurements as it constitutes a kind of “invasive technique”, being difficult to split apart the overhead caused by its calls. Therefore, access to cpu counters by means of specific probes seems to be more accurate, as this is a common hardware feature nowadays. The probe can be implemented, for instance, by a dedicated device driver to interface with the kernel of the operating system and specific routines to access the driver [8].

This approach was proposed for Intel processors running Linux back in 1995 [3] and successfully used in the current work. A low level operating system instruction provides cpu identification, size and setup of the L1 cache and which registers are supplied for the above purpose. Normally, there are *model specific registers* to monitor cpu events and a 64-bit time counter.

The cornerstone of the proposed strategy is to alternate runs of the code to be optimised in different machines and benchmarking each run with each compiler option. Initially, a coarse profiling is made in the machine where



the code is typically run, in the case of Hydrolight, a RISC workstation. Standard profiling tools aid to identify which routines could be modified in order to get a performance improvement. The approach assumes uneven profiles, leading to just a few routines to be analysed, in the case of Hydrolight. In this phase, some compiler optimisation levels are exercised and benchmarking takes into account the correctness of the results for typical test cases.

The next step is the identification of the critical, time-consuming parts of these routines. A timing function was written using the system wall time in order to provide internal timing information during the execution of these routines. This timing information is very useful, since the way the actual time-intensive routines use the execution time is masked in the profiling.

Once time-consuming routines are identified and timing information about their critical parts raised, the code is ported to the probe machine, in order to perform a finer profiling by counting system clock and performance and memory related events. Again, compiler options are used, and despite the use of a different compiler in a different machine, the closer monitoring allows a more thoroughly analysis of the routines.

Code optimisations can be performed using classical techniques [1] and significant improvements can be ported to the code running on the “operational” machines re-evaluating the performance under the different compiler options. This strategy aids to effectively check the “black box” of the optimising compiler options, since the variety of available compilers are very subtle concerning the achieved degree of optimisation. For an intensively used application software the effort of identifying performance bottlenecks and improving the code is certainly worth. Furthermore, the performance of the code generated by standard free distribution compilers can be enhanced, avoiding the need of migrating to a proprietary compiler due to poor performance.

3 The Hydrolight software

The Hydrolight software [4] is a well known radiative transfer software, written by C. D. Mobley in Fortran 77 [5] and intended for hydrologic optics problems. It basically solves the radiative transfer equation for a body of water, yielding the in-water light field given the state of the wind-blown water surface, the incident atmospheric light, the bottom boundary conditions, internal sources and inherent optical properties of the medium, such as absorption and scattering coefficients, index of refraction and scattering phase function.

This code is composed of specific radiative transfer routines and of standard numerical routines taken from LAPACK (*Linear Algebra PACK-age*), BLAS (*Basic Linear Algebra Subroutines*) and *Numerical Recipes*. It assumes the radiative transfer problem as being time-independent and one-dimensional in depth, leading to a plane-parallel geometry. The index of



refraction is taken as constant and a single wavelength is considered. The related equation can be expressed as:

$$\mu \frac{dL(z, \vec{\xi})}{dz} = -c(z) L(z, \vec{\xi}) + \int_{\Xi} L(z, \vec{\xi}') \beta(z, \vec{\xi}' \rightarrow \vec{\xi}) d\vec{\xi}' + q(z, \vec{\xi}) , \quad (1)$$

where z is the depth, $L(z, \vec{\xi})$ the radiance (directional-dependent), $\vec{\xi}' = \vec{\xi}'(\theta', \phi')$ and $\vec{\xi} = \vec{\xi}(\theta, \phi)$ are the incident and scattered directions for an infinitesimal beam ($\vec{\xi}, \vec{\xi}' \in \Xi$, which includes all directions), with each $\vec{\xi}$ direction defined by a polar (θ) and an azimuthal (ϕ) angle; β is the scattering phase function, to be chosen according to the problem being solved, and q the source term; c is the beam attenuation coefficient, that accounts for the true absorption and the scattering ($c = a + b$) and $\mu = \cos(\theta)$.

Hydrolight solves this equation by an invariant imbedding method [5], where a transport theory approach allows to estimate spectral reflectance, transmittance and source term operators that model the optical behavior of the water. Local operators are defined for an infinitesimal layer of water and calculated from physical properties of this medium, such as a , b and β . Global operators, applicable for a finite slab of water, are calculated using the local operators. The former operators are the unknowns of a system of non-linear ordinary differential equations, the Riccati equations. These equations are integrated by means of a higher-order Runge-Kutta algorithm implemented in routines from the Numerical Recipes library [6] which are part of the software and call specific invariant imbedding routines.

4 Identifying performance bottlenecks

The profiling of the Hydrolight 3.1 code, running in a Sparc Ultra workstation, and shown in Table 1 for the most time-intensive routines, revealed that only three routines (*sumphas*, *derivs* and *rhotau*) accounted for more than 95 % of the roughly 2.5 minutes wall time of each run. The calling tree of the code shows that these routines compose the kernel of the Riccati solver, being used by the code in any instance and thus worth of being optimised. In order to be solved, the Riccati equations are integrated by means of a driver routine (*odeint*) that uses a stepper routine (*rkqs*) which calls an adaptive 4th order Runge-Kutta solver (*rkck*). These three routines are from the Numerical Recipes library [6] [7] and call the three time-intensive routines mentioned above, specifically written for the Hydrolight code. Each time that *odeint* calls *rkqs*, the latter triggers *rkck* that calls *derivs* for the evaluation of the right hand side of the Riccati equations. This demands calls to *rhotau*, that evaluates the local operators and in turn calls *sumphas*, for the phase function evaluation. Particularly, in a typical Hydrolight run, these three routines are called almost 8×10^6 times each one. A timing function, *walltime*, was coded in C, and uses the system wall time provided



56 Applications of High-Performance Computers in Engineering VI

Table 1: Results of Hydrolight profiling on a Sun Ultra workstation, where RHS stands for right hand side of the Riccati equations.

Routine	Description	Time (s)	Time (%)
sumphas	phase function evaluation	57.21	36.70
derivs	RHS of Riccati eqs. evaluation	46.57	29.88
rhotau	local operator evaluation	45.78	29.37
rkck	Runge-Kutta algorithm	2.79	1.79

by the UNIX *gettimeofday* function. Calls to this function are inserted in the code of these critical routines, placed before and after every code segment to be analysed. The overhead of each timing call does not invalidate the timing information as it is close to $1 \mu\text{s}$ and each of these time-intensive routines account for times in the order of hundreds of seconds. Therefore, the time-consuming parts of the time-intensive routines were identified.

Hydrolight was then ported to the probe machine, a Linux monopro-cessed machine (Intel Pentium), in order to identify which are the critical parts of these routines. A straightforward analysis of the routines code shows intensive use of loops, that could be optimised by means of classical techniques. Therefore, the code underwent a fine profiling in order to check how these loops affect the execution time. The compiler chosen for the probe machine was g77, which does not have strong optimisation capabilities. Hence, a version of Hydrolight, with these routines optimised “by hand”, was then generated and run showing a significant improvement in performance.

An important aspect arises as typical optimising compilers are not aware of the run-time profile or code segment timing information. Only a sophisticated compiler with run-time profiling capabilities would, by means of an exploratory run, take advantage of such information. However, identifying which parts of the code are intensively used was a straightforward task, particularly in the case of Hydrolight, due to its uneven profiling.

In order to access the dedicated device driver that interfaces the kernel of the Linux operating system, three specific routines (*gettime*, *eventlog* and *reset*) and a function (*testcpu*) were written in C, as shown in Table 2. *testcpu* identifies the cpu and checks existence of special registers and also supplies information about L1 cache. P5 and P6 generations of Intel Pentium processor provide two 40-bit MSR's (*model specific registers*) and a 64-bit clock counter, the TSC (*time stamp counter*). The P5 allows to select 2 out of 41 events, and the P6, 2 out of 68. MSR can count basic events that include floating point operation, pipeline flush/stall, TLB/cache miss, multiply/divide operation (P6), L2 load/store (P6) and memory reference (P6). As in the timing information based on the system wall time, there is the overhead of calling these routines, approximately $1.5 \mu\text{s}$ for each call.



Table 2: Description of the procedures used to access the dedicated device driver.

Procedure	Description	Input	Output
testcpu	checks CPU type & MSR	- -	0 - unknown CPU 5/6 - Pentium P5/P6
gettime	reads TSC	-	TSC value
eventlog	selects events	two event codes	selected events
reset	resets counters	-	-

However, selecting and monitoring key events provide accurate and detailed information about cpu and memory usage. Moreover, timing information using TSC is far more accurate, because the overhead incurred in calling it is much smaller than with *gettimeofday*. This allows for the tuning of specific code segments that have a potential to be optimised. The main candidates, the routines *sumphas*, *derivs* and *rhotau*, were timed, showing many loops of typically tenths of seconds.

5 Optimising the code

Classical optimisations were used in the time-consuming loops of *derivs* and *sumphas*, with good results, and the use of compiler options for optimisation did not reproduce these improvements. The routine *derivs* has 8 loops that are executed in a typical run and consume from 300 to 700 miliseconds. Its first loop is depicted below:

```

DO 100 I=1,nmu
DO 100 J=1,nmu
  work(I,J) = tauhat(I,J)
  DO 100 K=1,nmu
    work(I,J) = work(I,J) + rhohat(I,K)*RT(K + (J-1)*nmu)
100 CONTINUE

```

Loop unrolling was done in the K loop, using a factor of 2, as the loop counter was bound to a low value (10), in any dimension. Another step was to avoid the update of the *work* elements for each iteration, as this loop performs a reduction. Temporary variables, *temp1* and *temp2*, were used:

```

DO 101 I=1,nmu
DO 101 J=1,nmu
  temp1=tauhat(I,J)
  temp2 = 0.
  DO 100 K=1,nmu,2
    temp1 = temp1 + rhohat(I,K)*RT(K + (J-1)*nmu)
100 temp2 = temp2 + rhohat(I,K+1)*RT(K+1+(J-1)*nmu)
101 work(I,J) = temp1 + temp2

```



58 Applications of High-Performance Computers in Engineering VI

Table 3: Results of timing the first loop of the routine *derivs* according to compiler options and performed optimisations.

compiler option	no loop unrolling	loop unrolling $\times 2$	reduction
no optimisation	570 μs	380 μs	-33 %
option <i>O2</i>	137 μs	126 μs	-8 %

The optimisation of the other loops in *derivs* would require a case specific analysis. The improvement in performance can be seen in Table 3, that shows the reduction in the execution time for the Linux machine, 8 % for the *O2* option. However, this small improvement cannot be dismissed in a routine that is called millions of times. Similarly, the sole loop of *sumphas* underwent a 2×2 blocking. Porting *derivs* and *sumphas* to the Ultra workstation, similar performance improvements were achieved, using or not the proprietary compiler option *O2*.

6 Porting Hydrolight to Fortran 90 and to HPF

Only the few time-intensive routines were ported to Fortran 90, as it would be wasteful, in terms of performance improvement, to translate all the code to this language. As Fortran 90 compilers optimise the code to a further degree than older compilers, some improvement was expected concerning performance. The Fortran 90 code was executed and profiled in the workstation. Comparison of this execution profile with the former Fortran 77 profile for the same machine shows a reduction of less than 1 % in the critical routines. However, the capital importance of porting part of Hydrolight to Fortran 90 was to achieve an intermediate step towards the parallel version in HPF - High Performance Fortran. The directives of the latter are simply inserted in the Fortran 90 code.

As the problem of distributing loads among processors constitutes the core of parallel computing, the previous analysis, even performed based on benchmarking data of monoprocessor runs, is useful as it helps to identify where to concentrate the code optimisation effort, for instance trying to exploit parallelism inside loops. Hence, the proposed strategy would be suitable for both mono and multiprocessor optimisations. Obviously, it would be recommendable to code a new version in Fortran 90 and to use the proposed approach for optimisation. Adding HPF directives would be the next step in order to run in multiprocessor architectures.

7 Concluding remarks

As further steps, we are coding all the remaining optimisations in the critical loops of Hydrolight, in order to generate improved Fortran 77 and 90 versions, and we are exploring the use of the HPF directives in the latter when using multiprocessor machines, in an effort that will lead to a parallel



version of Hydrolight.

The proposed strategy yields good results for the Hydrolight code and can be adopted as a standard procedure for the analysis and optimisation of a given code. As a byproduct, it supplies some evaluation of the compiler optimising capability and the machine performance concerning the specific application.

The access to cpu registers and time counter proved to be a powerful tool for performance analysis and provided, besides accurate timing information, an overview of cpu and memory critical events. Standard profiling tools are useful for a coarser analysis, in order to indentify which routines or code segments are worth of being submitted to a closer performance and memory monitoring. Hence, time-intensive parts of the code can be optimised and the performance upgrade evaluated.

The aim of the proposed procedure is to perform a non-automatic analysis of the software performance with the overhead of porting it to a probe machine, but that is worth for intensively used applications once significant increases in performance can be achieved. Despite evolving optimising capability of modern compilers, the performance gains that can be squeezed from a code depend strongly on how it was coded. This is a relevant issue for many application softwares which were developed years ago and worth the effort of enhancing their performance. Normally these applications are huge and complex codes and a complete rewriting or porting to a new language are prohibitive due to the amount of recoding that would be involved.

Acknowledgments

Authors S. Stephany and C. L. Mendes thanks FAPESP, São Paulo State Foundation for Research Support, for supporting this study through a Thematic Project grant (process 96/07200-8) and an Individual Research grant (process 98/3119-7).

References

- [1] Hennessy, J.L., & Patterson, D.A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, 1996.
- [2] High Performance Fortran Forum, *High Performance Fortran Language specification version 1.0*, Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, 1993.
- [3] Meyer, S., *MSR x86*, [online],
<http://www.mele-nebenjobs.de/stephan/msr.html>, 1999.
- [4] Mobley, C.D., *Light and Water - Radiative Transfer in Natural Waters*, Academic Press, San Diego, 1994.



60 Applications of High-Performance Computers in Engineering VI

- [5] Mobley, C.D., *Hydrolight 3.0 User's Guide*, SRI International, Menlo Park, CA, 1995.
- [6] Press, W.H., Flannery, B.P., Teukolsky, S.A., & Vetterling, W.T., *Numerical Recipes*, Cambridge University Press, New York, 1989.
- [7] Press, W.H., Teukolsky, S.A., Vetterling, W.T., & Flannery, B.P., *Numerical Recipes in Fortran 90*, Cambridge University Press, New York, 1996.
- [8] Rubini, A., *Linux Device Drivers*, O'Reilly & Associates Inc., USA, 1998.
- [9] Stephany, S., Ramos, F.M., Campos Velho, H.F., & Mobley, C.D., Reconstruction of bioluminescence sources in natural waters, *Advances in Computational Engineering Science*, eds. S. N. Atluri & G. Yagawa, Tech Science Press, Atlanta, pp. 447-452, 1997.
- [10] Stephany, S., Ramos, F.M., Campos Velho, H.F., & Mobley, C.D., Estimation of inherent optical properties for natural waters, *Proceedings of the 2nd Italian-Latinamerican Conference on Applied and Industrial Mathematics*, Quaderno CNR/GNFM n. 53, Italian Society of Industrial and Applied Mathematics, Rome, Italy, p. 230, 1997.
- [11] Stephany, S., Ramos, F.M., Campos Velho, H.F., & Mobley, C.D., A methodology for internal light sources estimation, *Computer Modeling and Simulation in Engineering*, **3**, pp. 161-165, 1998.
- [12] Stephany, S., Campos Velho, H.F., Ramos, F.M., & Mobley, C.D., Identification of inherent optical properties and bioluminescence source term in a hydrologic optics problem, *Journal of Quantitative Spectroscopy & Radiative Transfer*, 1999 (to appear).