Fortran Legacy Code Performance Optimization: Sequential and Parallel Processing with OpenMP

Fernando G. Tinetti* Mónica A. López Pedro G. Cajaraville Diego L. Rodrigues

UNLP
III-LIDI, Fac. de Informática
La Plata, Argentina

CONICET
Centro Nacional Patagónico
Puerto Madryn, Argentina

UNPSJB
Fac. de Ingeniería, Pto. Madryn
Puerto Madryn, Argentina

Abstract

Several optimization alternatives are presented for legacy Fortran 77 scientific programs, each one with a quantitative characterization in terms of performance gain. Initially, sequential optimization is focused on the analisys of Level 3 BLAS (Basic Linear Algebra Subroutines) utilization, since BLAS have several performance optimized implementations. Also, the Fortran 90/95 array notation is used as a code upgrade from Fortran 77 to Fortran 90/95 and, also, to provide the compiler a better source code for performance optimization. Since the shared memory parallel computing model is widely available (multiple cores and/or processors), the analysis of possible parallel processing via OpenMP is presented, along with the performance gain in a specific case. Sequential optimization as well parallelization work is done on a real (production code) program: a weather climate model implemented about two decades ago and used for climate research.

1. Introduction

Optimization of scientific legacy code still represents a major challenge for several reasons. Most of this code dates from about two decades ago, when computing resources were scarce and, thus, programs not only represent a numerical implementation of a mathematical/physical model. Instead, legacy scientific programs implement also some optimizations which are intermixed with the real modelization/scientific problem. One of the best examples of such optimizations is the usage of overlapped variables in Fortran 77 common blocks. Even when Fortran is currently just one of many programming languages, it was perhaps the *only one* programming language used by the scientific community at the time when legacy programs were created [5]. Some other challenges presented by legacy scientific code can be enumerated:

- Most of the current software engineering practices were defined and widely accepted after these programs entered in production environments.
- Complex mathematical/physical models are usually implemented [3], where there is a strong work on numerical accuracy and solution feasibility [4]. Each change on these programs imply careful analysis of *a priori* unknown side effects.
- As with many implemented applications that proved to be useful and stable, optimizations are considered dangerous or, at least, not having very good cost/benefit relationship.

Performance optimization has a number of motivations which encourage the work on scientific legacy code. Running time has been the traditional focus on performance optimization, but almost every change in legacy code implies some kind of usage of current software engineering practices. From another point of view, almost every current computer is in fact a multiprocessor (with multiple cores and/or processors accessing a main shared memory) and, thus, program parallelization can be considered an optimization for resource usage [1]. Otherwise, legacy code would not use every available core or processor. In the context of high performance computing, parallelization is the usual work for applications even without real time limitations, since running time is usually reduced just as a way to allow research and/or solve problems with high computing requirements [6]. One of the best approaches to parallelize scientific on shared memory computers code has been OpenMP [2], which is also widely accepted in C and Fortran compilers.

*Investigador Asistente, Comisión de Investigaciones Científicas de la Provincia de Buenos Aires

978-0-7695-3507-4/08 $25.00 © 2008 IEEE
DOI 10.1109/CSIE.2009.90

Optimization work presented in this paper is made by means of example: a climate model implemented in Fortran 77 about two decades ago. This application is being used since many years ago for climate research. Even when the presented work is specific, the main objective is to introduce general ideas to be useful for similar scientific programs, i. e., programs with similar processing patterns. The whole optimization work has been restricted to maintain the general program structure and implementation model, no general reengineering is accepted or taking into account. The main advantage of this approach is the short-term optimization: the optimization can be started without much knowledge of the application domain (climate modeling, in this specific example), just analyzing the numerical computing patterns. Of course there are disadvantages, one of the main being the limited degree of changes considered for optimization. Just as an example: if computing is made at the time domain, it is not possible to compute at the frequency domain with a major reengineering process: transform - computing - transform, which imply many lines of code but, also, could provide a better code for sequential and/or parallel optimization.

2. Initial Optimization Steps on Legacy Code

In this section, the minimum (necessary for context) information about the scientific program to be optimized is presented, along with the standard initial steps on any optimization process. Most of the information provides the context of the production environment and the standard way for gathering data of a program performance. A bare minimum static program analysis and execution environment provides this information:

1) The program is distributed on about 300 .f (Fortran 77) files. Most of the files implement only one Fortran subroutine. Less than 10% of the files are used for common blocks and constants.

2) The number of lines in .f files add up to about 58000, and approximately 25% of them are comment lines.

Intel Fortran compiler 10.1 is used (ifort) [9], with the highest optimization level: -O3.

3) Performance experiments are carried out on dual AMD 246 Opteron computers (two single cores at 2.0 GHz) with 64 bits Linux kernel smp0 2.6.23 (Fedora Core 8 distribution), 2 GB RAM.

Running the program with profiling collects the initial most important dynamic data about performance, from which almost every optimization process begins:

1) About 230 routines are called/used at run time. Most of the runtime is spent in routines located at deep levels 5 to 7 in the dynamic call graph from the main routine.

2) The routine with most of the runtime (the "top routine" from now on) requires more than than 9% of the total program runtime and is called about 315000 times.

3) The top 10 routines (the 10 routines at the top of the flat profile) require about 50% of total runtime. Two of them are related to intrinsic Fortran functions.

Most of this information can be extracted out automatically, without previous knowledge of the application and/or specific program. Static data is extracted from simple scripts on the source code, and dynamic data is extracted by using gprof and parsing the resulting text. The first conclusion at this stage is that focusing on the top ten routines for optimization would provide a very good potential performance gain: relative to 50% of the total program runtime. The next sections will show the optimization work on the top routine, with comments on applying the same kind of optimization work in general or, at least in the top 10 routines.

3. Sequential Processing Optimizations

The initial objective on sequential optimization focused on identifying and using Basic Linear Algebra Subroutines (BLAS). The main motivating factor in order to use the BLAS is the availability of highly optimized and/or tuned BLAS implementations, such as ACML (AMD Core Math Library [7]), MKL (Intel Math Kernel Library [10]), and ATLAS (Automatically Tuned Linear Algebra Software [8]). The best performance is found at the Level 3 BLAS routines, with matrix-matrix operations, which have access to $O(n^2)$ data and requires $O(n^3)$ floating point operations. The first task is, thus, to identify BLAS 3 routines in order to make the specific calls to an optimized library.

Looking for optimization through the Level 3 BLAS the first problem was found: the top routine has very *low level* processing. Most of the numerical computing is made on *plain* vectors or matrices accessed as vectors, such as sown in Fig. 1, where four matrices (`m1`, . . ., `m4`), four vectors (`v1`, . . ., `v4`), and three constants (`c1`, . . ., `c3`) are used. This code is representative not only for the top routine but

```
 do 10 i=1,rs*cs
   m3(i,1)=c1*(v1(1)*m1(i,1)+v2(1)*m2(i,1))
   m4(i,1)=c1*(v3(1)*m1(i,1)+v4(1)*m2(i,1))
   m3(i,1)=(c2+m3(i,1))*m3(i,1)+c3
   m4(i,1)=(c2+m4(i,1))*m4(i,1)+c3
   m3(i,1)=m5(i,1)*(m3(i,1)**16)
   m4(i,1)=m6(i,1)*(m4(i,1)**16)
10 continue
```

Figure 1. Example of Code in the top routine.

for most of the computing routines in the program. Furthermore, it is not possible to relate the code in Fig. 1 with Level 3 BLAS, but with Level 1 BLAS (vector-vector operations). More specifically, most of the expresions in Fig.

1 are a sequence of `saxpy()` routine calls. Even when it cannot be expected that Level 1 BLAS are as optimized as Level 2 or Level 3 BLAS, it is expected some performance gain by calling optimized linear algebra libraries such as ACML, MKL, or ATLAS. Due to the relatively complex vector operations expressed in terms of `saxpy()`, the top routine *became* about 100% longer than the *original* one. However, most of the changes can be made automatically by a lexical parser of Fortran expresions. Preliminary experiments show that the optimized BLAS 1 routines do not provide any performance gains and, thus, this kind of code transformation can be discarded.

Summaryzing, the code does not *allow* to be optimized by means of Level 2 or Level 3 BLAS and Level 1 BLAS does not provide any performance gain. At this point, the main option seems to be a major numerical processing rearrangement/recoding such that computing is made through Level 3 BLAS. Since it was *a priori* defined to avoid a major software reengineering process, it was chosen to provide a better source code for the Fortran compiler. Basically, the idea is to transform the code to Fortran 90/95 *array notation* in order to "help" the compiler to avoid complex analysis for code optimization (always at the maximum optimization level -O3). One more simple observation: Level 1 BLAS processing-like is almost trivial to be transformed to array notation and, also, the relatively complex expressions remain almost the same but on arrays instead of scalars (individual matrix or vector elements). A simple script was developed in order to make the automatic translation of Fortran iterations such as that of Fig. 1 to Fortran 90/95 array notation. The main conversion rules are very simple:

- Only Fortran Do loops are analyzed.
- The control variable of a Do loop has to appear as a matrix index at left and right of an assignment.
- The control is used only as a matrix index, i. e. the control variable should not be involved in any arithmetical expression.
- The initial and ending values of the control variable are used on the array notation to indicate the array *slice*.

These simple transformation rules are highly effective at least in the program used in this work: successfully transformed 10 of 14 On the other top 10 routines, the simple script successfully transformed from 25% to 60% of the Fortran Do Loops. Furthermore, the script is useful to identify the routines needing special work (those routines with less than 50% of transformed loops, for example). Fig. 2 shows the code in Fig. 1 in terms of Fortran 90/95 array notation.

Rather surprisingly, the effectiveness of the compiler in optimizing processing expressed in terms of Fortran array

```
m3(1:rs*cs,1)=c1*(v1(1)*m1(1:rs*cs,1)+
              v2(1)*m2(1:rs*cs,1))
m4(1:rs*cs,1)=c1*(v3(1)*m1(1:rs*cs,1)+
              v4(1)*m2(1:rs*cs,1))
m3(1:rs*cs,1)=(c2+m3(1:rs*cs,1))*
              m3(1:rs*cs,1)+c3
m4(1:rs*cs,1)=(c2+m4(1:rs*cs,1))*
              m4(1:rs*cs,1)+c3
m3(1:rs*cs,1)=m5(1:rs*cs,1)*
              (m3(1:rs*cs,1)**16)
m4(1:rs*cs,1)=m6(1:rs*cs,1)*
              (m4(1:rs*cs,1)**16)
```

Figure 2. Example of Code in array notation.

notation provided more than 17% performance gain for the top routine. This gain is always proportional to the number of transformed Do loops into array notation. As a side effect, the routine code is shorter: the initial and ending line of Do loops are not necessary, since the array notation defines the complete range of matrix or vector elements.

Most of compiler performance gain is due to vectorization in terms of Intel compiler terminology: taking advantage of SSE (Intel Streaming SIMD Extensions), implemented also by most of current AMD processors, e.g. AMD Opteron implements SSE2 instructions. At least intuitively, this prevents from explicitly using other classical optimization techniques, such as loop unroll and block processing. Loop unroll is focused on having many scalar operations in order to take advantage of superscalar processors, and having many scalar floating point operations (scientific code) is not compatible with having multiple vector operations which enable vectorization. The example code shown in Fig. 1 processed via vectors imply a simple way of block processing. More specifically, the access to individual matrices (such as `m3` in Fig. 1) is made by contiguous elements with "SSE vectors length" stride and, also, the short term references to the same matrices imply data reuse, which proportionally improve cache hits. The effect of this simple block processing does not provide a very high optimizatin improvement as in the case of Level 3 BLAS operations, but it is in fact a good (17%) improvement in cache optimization for Level 1 BLAS operations. Is it the best improvement? As always, the answer depends on the legacy scientific code. In this special case, another level of cache usage optimization is possible by looking for references to the same matrix, e.g. `m3` in the example above, and moving them as close as possible in order to avoid losing the data in cache. This task has not being carried out, it has been left as one of the task for further improving sequential performance in the future.

Summarizing, one the most important sequential optimization tasks was not possible: identify and use Level

3 BLAS routines. Also, Level 1 BLAS routines provided by optimized libraries were tested and did not provide any performance improvement. Instead, using array notation in processing allowed the compiler (with the same compiling options) to generate a binary code with a 17% performance improvement on an AMD Opteron processor. Even more interesting is that a very good fraction of the task to transform source code can be made automatically in a few minutes with a simple tool applied on the legacy source code. Also, other optimization techniques were left to be applied in the future, including the analysis of (simple) automatic tools to source code transformation.

4. Parallel Processing on Multiprocessors

The first option to optimize resource usage on multiprocessors (more than one processor sharing memory or more than one core per processor) is OpenMP [2]. OpenMP has proven to be useful and adopted directly by most of the C and Fortran compilers, including ifort, the compiler used in the current work. It is worth noting that Fortran is a strong requirement due to the large number of legacy applications programmed in this language. Maintaining the Fortran language avoids recoding routines or parts of routines, just like the top routine being analyzed and optimized in the previous section. Furthermore, the top routine has been analyzed in order to use OpenMP after being transformed to array notation. Two main factors are important in order to apply both code optimizations:

- Sequential optimization in general and array notation in particular should not prevent from using OpenMP, both optimizations are necessary. Sequential optimization optimizes resource usage in a single processor or core, and OpenMP optimizes resource usage in a multiprocessor/multicore.
- Optimization gain is characterized in terms of global running time reduction, i.e. with every optimization applied to the legacy code.

In fact, OpenMP can be used easily on Fortran array notation: just use `WORKSHARE` OpenMP directive. However, there are at least two important factors to be taken into account:

- Level 3 BLAS routines should be the main focus for optimization/parallelization if are used in scientific code. However, most implementations, such as ACML, MKL, and ATLAS are already optimized with parallelization via OpenMP (ACML and MKL) or pthreads (ATLAS).
- The Intel Fortran compiler does not always works properly with the `WORKSHARE` OpenMP directive, thus, the parallelization task has to be made in some other way.

Given that array notation should be maintained in order to preserve the optimization gains provided by the compiler, the idea is just to distribute the arrays into as many array slices as OpenMP threads. Without the explicit distribution of array expressions to OpenMP threads, the code in Fig. 2 is easily parallelized with OpenMP as shown in refcode3. However, given that processing has to be distributed explicitly, computing is defined in terms of the array slice

```
!$OMP PARALLEL WORKSHARE
   m3(1:rs*cs,1)=c1*(v1(1)*m1(1:rs*cs,1)+
                  v2(1)*m2(1:rs*cs,1))
   m4(1:rs*cs,1)=c1*(v3(1)*m1(1:rs*cs,1)+
                  v4(1)*m2(1:rs*cs,1))
   m3(1:rs*cs,1)=(c2+m3(1:rs*cs,1))*
                  m3(1:rs*cs,1)+c3
   m4(1:rs*cs,1)=(c2+m4(1:rs*cs,1))*
                  m4(1:rs*cs,1)+c3
   m3(1:rs*cs,1)=m5(1:rs*cs,1)*
                   (m3(1:rs*cs,1)**16)
   m4(1:rs*cs,1)=m6(1:rs*cs,1)*
                   (m4(1:rs*cs,1)**16)
!$OMP END PARALLEL WORKSHARE
```

Figure 3. Usage of OpenMP WORKSHARE.

processed by each OpenMP thread, which is easily computed as shown in Fig. 4. In the same OpenMP parallel region is included the code of Fig. 2, replacing `1:rs*cs` by `ini_i:end_i`. From a more general point of view,

```
 !$OMP PARALLEL PRIVATE(t_n, ini_i, end_i)
 !$OMP SINGLE
       num_threads = OMP_GET_NUM_THREADS()
       chunk       = (rs*cs) / num_threads
 !$OMP END SINGLE
       t_n         = OMP_GET_THREAD_NUM()
       ini_i       = t_n * chunk + 1
       end_i       = ini_i + chunk - 1
       IF (end_i .GT. (rs*cs)) THEN
         end_i = rs*cs
       END IF
 ...
```

Figure 4. Array slices in every thread.

this paralelization is made with a simple code transformation: every array expression is transformed by changing array notation. Only expressions with array notation are taken into account. Furthermore, the transformation task is such that the whole array (or array slice) is divided into as many slices (or *sub slices*) as OpenMP threads, as shown in Fig. 4.

The top routine was parallelized following the guideline defined above, and the performance gain was about 19% (on a dual single core processor), which implies a total gain of more than 33%. Thus, the top routine runs in about 1/3 of the original time (i.e. without any sequential optimization) by using array notation and OpenMP directives on a computer with two processors. Performance optimization can be considered far from optimal, since 19% is just a fraction of a theoretical 100% gain by using two processors instead of one, but

- Not every loop can be translated to array notation and not every array notation expression can be parallelized because of data dependencies.
- Optimization/parallelization is made almost authomatically, and 19% is thus obtained almost authomatically at least on this legacy code.
- Multicore/multiprocessor is available on any current computer, and every optimization to use more than one processing element in parallel provides improved performance.
- The top routine is called about 315000 times at runtime, and this makes OpenMP thread overhead time (for creation, synchronization, etc.) comparable to thread processing time.

As another consequence of optimization, the top routine became the third routine in the ranking of running time given by the profiler. At this optimization level, a further analysis involves the call tree/graph in order to identify routine/s which call the top routine or the top ten ones. Basically, the main idea is to avoid the overhead by crating, synchronizong, etc. OpenMP threads less frequently than in the top routine. This optimization/parallelization process has been left for further improving sequential performance in the future.

5. Conclusions and further work

Table 1 shows a summary of sequential as well as parallel performance improvement on the top routine (R. and T. stand for Relative and Total respectively). It is worth noting that most of the performance optimization is made automatically, by identifying simple processing patterns (DO loops, basically). Sequential optimization is made via the compiler, by vectorizing (using SSE2 in terms of Intel Fortran compiler) array expressions. Code parallelization is made by distributing array expressions among OpenMP threads, wich execute in each available processor (or processor core).

Table 1. Results

Top Routine	% R. Gain	% T. Gain
Sequential (array notation)	17%	17%
Parallel (OpenMP)	19%	33%

Further sequential as well as parallel improvement is possible and -at least a priori- seems to be by means of simple source code transformations. However, no authomatic code transformations have been devised for those performance improvements (as the ones defined in this paper). None of the legacy code optimizations imply a major redesign/reengineering process, such as that proposed in [1].

Performance improvements reached on the top routine are expected to be applied at least to the top ten routines. If every code transformation is successfully applied to the legacy code, the running time can be reduced by 1/3 of the current running time. This improvement is reached almost automathically by means of simple and local (to each routine) code transformations. OpenMP has been successfully used in this work for legacy code parallelization on shared memory parallel computers (multiple processors and/or multicore processors). Parallel processing on a distributed memory architecture imply a major analysis of source code and is an open research line.

References

[1] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing. State of the Art in Scientific Computing, PARA 2006, Umeå, Sweden, June 2006*, LNCS 4699, pages 1–10, 2007.

[2] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[3] S. Chapra and R. Canale. *Numerical Methods for Engineers*. Mc Graw Hill, fifth edition, 2005.

[4] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, third edition, 1996.

[5] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[6] G. W. Sabot, editor. *High Performance Computing: Problem Solving with Parallel and Vector Architectures*. Addison-Wesley Publishing Company, 1995.

[7] *AMD Core Math Library (ACML), http://developer.amd.com/acml.jsp*.

[8] *ATLAS (Automatically Tuned Linear Algebra Software), www.netlib.org/atlas/index.html*.

[9] *Intel Corporation, Intel Fortran Compiler 10.1, Professional and Standard Editions, for Linux, http://www.intel.com/cd/software/products/asmo-na/eng/compilers/282048.htm*.

[10] *Intel Corp., Intel Math Kernel Library, http://www.intel.com/cd/software/products/asmo-na/eng/perflib/307757.htm*.