Experiences in Parallelizing a Numerical Model

Fernando G. Tinetti[1], **Maximiliano J. Perez**[3,2], **Ariel Fraidenraich**[2,3], **Adolfo E. Altenberg**[4,2]
[1]Fac. de Informática, UNLP, Comisión de Inv. Científicas Prov. Bs. As., La Plata, Argentina
[2]Universidad de Belgrano, Ciudad Autónoma de Buenos Aires, Argentina
[3]Universidad Nacional de La Matanza, Buenos Aires, Argentina
[4]UTN-FRBA Universidad Tecnológica Nacional Fac. Regional Buenos Aires, C.A.B.A., Argentina

Abstract – *We present the parallelization work on a shallow water numerical model. The standard approach of profile-parallelization is presented, where some unexpected results and problems are found and explained for this specific case. We show similarities and differences among different (e.g. production and development) computing platforms. We also show several limitations of very well-known and experimental tools and methodologies traditionally used in profiling, performance evaluation, and parallelization tasks in shared-memory computing platforms. We finally present our step-by-step parallelization approach, including some pitfalls in obtaining enhanced performance and/or evaluation of the actual performance. Combining several source code analysis techniques as well as expert users experience we have been able to speed-up the runtime on multiprocessor computers.*

Keywords: High Performance Computing, Shared Memory Parallel Computing, OpenMP, Numerical Models.

1 Introduction

We started our work on a shallow water numerical model, designed following a Taylor Galerkin scheme [1] [2]. The spatial resolution is accomplished with weighted residuals or Galerkin method, and the temporal advance by means of a Taylor series approximation. The Taylor method is based on the knowledge of the Jacobian matrices that correspond to the projections of flows in the two orthogonal Cartesian coordinates [1] [2]. The differential system of shallow water equations is expressed conservatively by Eq. (1) as follows:

$$\frac{\partial U}{\partial t} + \frac{\partial F_i}{\partial X_i} - \frac{\partial Rd_i}{\partial X_i} = R_s, \quad i = 1, 2 \tag{1}$$

Where

- $U(h, p, q)$ is the nodal unknowns vector.
- F_i is the convective flow vector in axis x and y, $i = 1, 2$, as shown below in Eq. (2) and Eq. (3).
- Rd_i is the diffusive vector, $i = 1, 2$.
- R_s is the source terms vector (containing the terms of topographic and friction variations) given by Eq. (4) below.

$$F_1 = \left(p, \frac{p^2}{h} + g\frac{h^2}{2}, \frac{p\,q}{h}\right)^T \tag{2}$$

$$F_2 = \left(q, \frac{p\,q}{h}, \frac{q^2}{h} + g\frac{h^2}{2}\right)^T \tag{3}$$

$$R_s = \left(0, -gh\left(S_0^x - S_f^x\right), -gh\left(S_0^y - S_f^y\right)^T\right) \tag{4}$$

The initial conditions are the natural quiet lake, and the boundary conditions are given by

- $P(0, y, t) = \mu(y, t)$, where the μ is the velocity at the entry section.
- $h(L_x, y, t) = h_0$, where h_0 is the final depth at L_x (out section).
- $q(L_x, y, t) = 0.0$
- $q(x, 0, t) = 0.0$, $q(x, L_y, t) = 0.0$, where these conditions represent the normal fluxes in the lateral boundary. The same boundary conditions are applied to a symmetrical circular hole.

Simulations results provide the values of $U(h, p, q)$, the vector of nodal unknowns, where h is the water depth, p is the discharge flow in the x direction (longitudinal) and q discharge flow in the direction y (transversal). The time evolution is modeled as a sequence of two half-steps of temporary advance, for which the solution of the first half-step is the initial condition of the second:

$$U^{n+\frac{1}{2}} = U^n + \frac{1}{2}\,\Delta t\,\left(R_s + \frac{\partial R_{di}}{\partial x_i} - \frac{\partial F_i}{\partial x}\right)_{(t_n)} \tag{5}$$

$$U^{n+1} = U^n - \Delta t\,\left(\frac{\partial F_i}{\partial x_i} - \frac{\partial R_{di}}{\partial x_i} - R_s\right)_{\left(t_{n+\frac{1}{2}}\right)} \tag{6}$$

Simulations are done with a triangulation similar to that shown in Fig. 1, representing finite elements domain subdivision, with radial symmetry near a circular obstacle and densification in the lateral edges.

The starting point of this work is the Fortran program that implements the method briefly described above. The objective is twofold:

- Enhance/update the Fortran source code in case it is needed.
- Speed-up program execution, taking advantage of the multiprocessing facilities provided by current computing platforms.

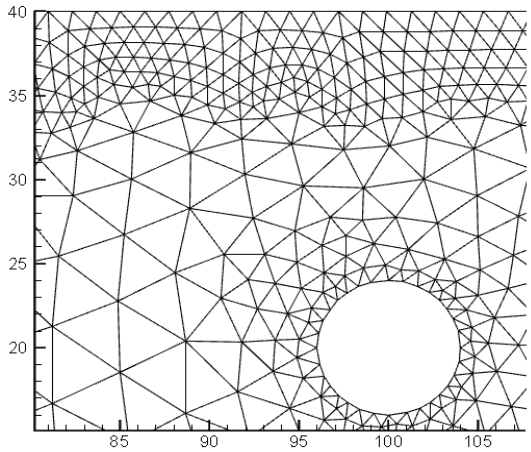

Figure 1: Finite Element Mesh.

We consider Fortran source code enhancement/update only in the case it is needed at the core of processing. We do not approach the whole Fortran program as a software engineering source code upgrade project, e.g. for producing full Fortran 2003 or Fortran 2008 code [3] [4]. Instead, we enhance source code features (e.g. readability, using new syntax forms, etc.) if we consider they are needed for the parallelizing analysis/implementation. Speed-up is approached by means of the traditional shared memory parallel processing using OpenMP [5].

The next section outlines the standard and initial work done on/with the program, including profiling, for example. Section 3 details the first parallelization work along with the obtained performance. Section 4 explains how the scientists experience and domain knowledge leads to further parallelization and performance improvement. The last section details several conclusions and further work on the program and the application area.

2 "Standard" and Initial Work

As part of the standard work on optimization and parallelization [7] we collected profiling data. This task does not require much more than an extra experiment, compiling the program with the option -pg (or -p) of the GNU Fortran compiler. The data collected is shown in Table I, which corresponds to the production environment, i.e. where the program is used by the application area scientific research. The production environment computer is an Intel I5-2310 @2.9GHz, the total runtime is about 11500s (more than 3 hours), and all subroutines with more than 5% of the runtime are shown in Table I. We do not include the actual subroutine names except for the main one because they are not relevant for the performance analysis to be carried out in this paper.

TABLE I. PROFILING DATA – PRODUCTION ENVIRONMENT

Time/s (s)	Subroutine Name
5853	subr1 (51%)
2350	main
1525	subr2
843	subr3
787	subr4

The parallelization work was carried out in a rather old computing facility, based on dual Intel Xeon 5405 @2.00GHz. We will call the computers environment where we carried out the parallelization work as the "development environment". The production environment facilities for computing is mostly focused to the scientific research in the application area (shallow water simulations in this specific example). Thus, having a development environment lets to carry out most of the development work and experimentation without any interruption of the application area scientific work. The production environment is used only when performance results must be verified in the actual working platform of the scientists. The total measured runtime by the profiler was about 21500s (approx. 6 hours). Table II shows the profiling data up to the main subroutine. There are several differences when Table I and Table II are compared:

- The profiling order for subroutines is different. Among those differences: the main subroutine is the second most important in the production environment, while it appears in the 6th position in the development environment.
- The runtime weight of each subroutine is different. Among those differences: the "top" subroutine, i.e. the subroutine with where most of the runtime is spent, subr1, is identified as requiring about 51% of the runtime in the production environment (Table I) and less than 20% in the development environment (Table II).

TABLE II. PROFILING DATA – DEVELOPMENT ENVIRONMENT

Time/s (s)	Subroutine Name
3910	subr1 (19%)
3648	subr2
2347	subr5
1580	subr4
1277	subr6
1103	main

We further experimented in the development computers in order to verify differences explained above. Besides running the program with GNU profiling we also add our own measurement on global runtime. Once again, we found (large) differences: while the profiler accounts for approx. 6 hours (about 21500s), the wall-clock time we measured was

about 10.5 hours (about 31600s). Even when the profiler accounts only program runtime (i.e. not when the program is waiting for I/O or using the CPU) [6], we are not able to assign such a difference to "non-running" time periods, because there were no other programs running in the same computer while collecting data.

We did not further investigate the possible GNU profiling errors, but we focused our effort in subr1, because it is in either case the most time-consuming program subroutine. Subroutine subr1 contains code similar to other programs' subroutines in the scientific processing field [8]:

- About 220 source lines of code.
- A rather large number of parameters/arguments (more than 20).
- Programmed in a mixture of F77 and F90 (e.g. fixed-column format and Do-End Do constructs).
- Contains a sequence of Do loops, where the processing (mostly on arrays) is carried out. More specifically this subroutine contains a sequence of 6 Do loops, most of the code is inside those Do loops.

We made a minimum upgrade/enhancement of source code in the subr1 subroutine:

- Some indentation in source code where fixed-column format was hard to read. Besides, we have taken used the compiler option in order to override the fixed line length.
- Move a few source code lines corresponding to loop-invariant code. We know that optimizing compilers can do such movement in the binary, but we manually do it for having simpler Do loop blocks. And having simpler Do loops simplifies the analysis for OpenMP/thread processing.
- Instrument Do loops for having some "extra" profiling information that no standard profiler is not actually able to provide: individual Do runtime. As a "collateral effect" we are able to aggregate the individual Do loop runtime in order to have the information about the whole subr1 runtime. And this whole subr1 runtime can be also used for verification of the data provided by the profiler.

As the result of the last source code modification (i.e. subr1 instrumentation), we have obtained very useful data, as shown in Table III:

- The total of the first five Do loops account for more than 99% of the runtime, i.e. we are able to discard the last Do loop for parallelization/optimization tasks.
- The total runtime is evenly distributed in the first five Do loops.
- The total subr1 runtime adds up to about 43% of the total runtime, a value in between that provided by the profiler in the production environment (Table I, 51%) and that provided by the profiler in the development environment (Table II, 19%).

TABLE III. INSTRUMENTATION DATA – DEVELOPMENT ENVIRONMENT

Time/s (s)	Code
31560	Whole Program
13500	subr1 (43%)
2660	subr1 1st Do
2700	subr1 2nd Do
2660	subr1 3rd Do
2660	subr1 4th Do
2700	subr1 5th Do
20	subr1 6th Do

3 First Parallelization Work

As a first and straightforward approach, we identified all the Do loops in the whole source code for OpenMP parallelization. On those Do loops, we identified a large fraction of them that are almost directly parallelizable by including simple OpenMP "parallel do" work-sharing constructs [9]. The syntactic analysis followed the lines of identifying the simplest conditions under which a Do loop processing can be carried out in parallel with OpenMP threads, such as those described in [10].

Even when we know this is not the standard way of computing parallelization approach, several reasons led to this initial (parallelization) work:

- It requires almost no prior knowledge of the source code and application. In this sense, we could call this approach as a "blind" parallelization approach.
- It does not require any change to the existing source code for processing, only the addition of OpenMP directives. Furthermore, the only OpenMP directives to be used are the parallel Do work-sharing constructs.
- The original code is maintained and can be used at runtime. Exactly the same program binary/executable is generated by the compiler when the specific option to take into account the OpenMP directives is not used (in that case, the OpenMP directives are considered as source code comment lines by the compiler).
- The "blind" parallelization approach can be used in general, not only in this program/problem. If we prove it is successful we would have at least one case for suggesting the same approach to other computational scientists/other HPC (High Performance Computing) programs/code.

Once we applied this "blind" approach, we were able to run the code and

- Verify simulation model output, which was identical (at the binary level) to that of the sequential execution with the same input. This verification let us know that the

work on Do selection and OpenMP usage was the right one.

- Analyze the obtained performance. The runtime of the parallelized code (it effectively used more than one processor) was worse than that of the sequential (original) processing. The performance loss was between 20% and 30% in the production environment as well as in the development environment. Even when we could further investigate the specific penalties introduced by parallelizing different data and, for example producing false data sharing and/or low performance memory footprints, we opted to follow a more traditional approach. We focused the work on parallelizing the most time-consuming subroutine, subr1.

Analyzing the source code of the subr1 subroutine, we have five prospective Do loops on which to apply OpenMP work-sharing constructs (see Table III). Actually, we were able to take advantage of the previous "blind" approach, because subr1 Do loops have already been analyzed in the context of the whole set of program's Do loops. From that analysis, we found out that the second and fifth Do loops are easily parallelizable, i.e. they do not have any data dependence that would cause a race conditions (and wrong results) at runtime. Applying the OpenMP parallel Do work-sharing construct to those loops we obtained the runtimes shown in Table IV in the development environment described above, dual Intel Xeon 5404 (i.e. with 8 threads). Since the Do loops are in fact a subset of those in the previous "blind" parallelization work, the numeric output is expected to be and actually are the right ones, i.e. identical to that of the sequential (original) program version.

TABLE IV. INSTRUMENTATION DATA – INITIAL PARALLELIZATION

Code	Seq. Time/s (s)	Par. Time/s (s)
Whole Program	31560	24500
subr1	13500	6370
subr1 1st Do	2660	1810
subr1 2nd Do	2700	360
subr1 3rd Do	2660	1820
subr1 4th Do	2660	1810
subr1 5th Do	2700	360

Taking into account the sequential and parallel runtime in Table IV we are able to identify a twofold performance gain:

- The expected gain in performance for the second and fifth Do loops. In this specific case, the runtime is reduced from 2700s to 360s using 8 cores/threads.
- The unexpected gain for the (still) sequential runtime of the non-parallelized first, third, and fourth Do loops. A possible explanation is that given that two out of five Do loops are computing in parallel, the data is better distributed in the cache hierarchy and this leads to less cache contention and a better cache hit ratio.
- We consider it is possible that the unexpected gain in sequential processing makes the parallel gain near the optimum one. More specifically, we think that the runtime reduction from 2700s to 360s is partially due to a faster sequential processing in each thread.

4 Further Parallelization Work

Even when we have obtained a rather large performance gain as shown in Table IV above, we have several lines of work for obtaining even better performance. The first option for further analysis are the Do loops no yet parallelized. The first, third, and fourth Do loops not only require a large amount of runtime, but now they have a larger fraction of the total subroutine subr1 runtime. Without parallel processing, the first, third, and fourth Do loops involve 3/5 of the total, runtime, i.e. about 60%. With parallel processing, almost all the runtime is spent in those three Do loops: 5440s out of 6160s. i.e. about 90% of the runtime.

Taking a deeper view of the three non-parallelized Do loops, they have a common characteristic by which they are not easily identified as parallelized. All of them include "indirect" array access: while the simple Do loops include code such as that shown in Fig. 2a), where <expr> possibly includes reference/s to Arr(I), the code in the three non-parallelized Do loop includes code as that shown in Fig. 2b). Independently of expression <expr> in Fig. 2b) it is not possible to ensure that threads processing different portions of the Do loop will not interfere with each other. Basically, if there are $Ndx(I_1) = Ndx(I_2)$ for $I_1 \neq I_2$ and $I_1, I_2 \in [1, N]$ then there will be a race condition, because it is not possible to ensure that no two threads will be asynchronously changing the same value (array index) of Arr.

```
DO I = 1, N
  ...
  Arr(I) = <expr>
  ...
END DO
```

```
DO I = 1, N
  ...
  Arr(Ndx(I)) = <expr>
  ...
END DO
```

Figure 2: Different Do Loop Codes.

Race conditions in the parallel processing of those Do loops are avoided if the following condition holds (regarding the indirection in the data array access):

$$Ndx(I_1) \neq Ndx(I_2) \; \forall \; I_1 \neq I_2, \; I_1, I_2 \in [1, N]$$

In other words: only if the previous condition holds, the first, third, and fourth subr1 subroutine Do loops can be processed in parallel. At this time, only the experts in the application field are able to know the "meaning" or contents of the Ndx array, the way in which it is used. And the experts effectively know that the condition above holds. Beyond the specific

meaning of the Ndx usage, which is strongly dependent of a number of program and programmer(s) characteristics, we think it is highly useful to know the right questions to the scientific users/programmers. In the work specifically described in this paper, we have taken advantage of what we know about parallelization of Do loops as well as the knowledge of the application field programmers which we finally used for parallel processing. Summarizing, we were able to parallelize the first, third, and fourth subr1 subroutine Do loops thanks to the information provided by the experts on the indirect access of arrays in those Do loops.

Table V shows the runtimes in the development environment described above, dual Intel Xeon 5404 (i.e. with 8 threads) with the five (main) loops of subroutine subr1 parallelized with OpenMP. We have included the sequential times for aiding the performance gain analysis. These time data are provided by our instrumented code. Clearly, having the surbr1 subroutine parallelized, most of the program's runtime is spent in other program's code sections/ subroutines. Taking into account the values in Table V, the subr1 subroutine runtime is only about 10% of the total runtime.

TABLE V. INSTRUMENTATION DATA – PARALLEL PROC.

Code	Seq. Time/s (s)	Par. Time/s (s)
Whole Program	31560	20040
subr1	13500	1950
subr1 1st Do	2660	350
subr1 2nd Do	2700	360
subr1 3rd Do	2660	340
subr1 4th Do	2660	340
subr1 5th Do	2700	360

Fig. 3 shows the runtime and speedup of the subr1 subroutine for different number of threads (CPUs) in the development environment. We have made the corresponding experiments in order to see the scalability evolution of the code at least in the development environment, where multiple performance experiment are possible.

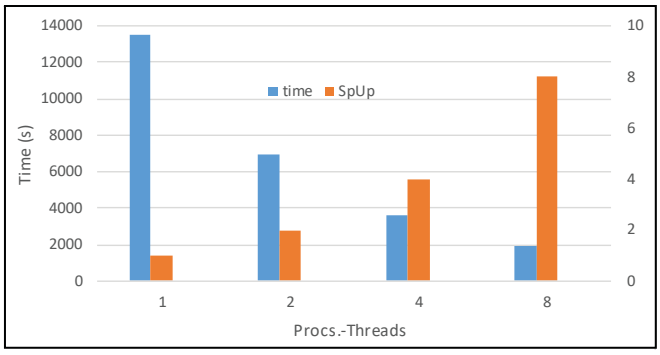

Figure 3: Runtime and Speed-Up of the subr1 Subroutine.

Given the good performance results in the development environment, we considered to use this version in the production environment. We keep the instrumented code for analyzing the real performance gain in the production environment. Hopefully, the gain would be as good as that in the development environment, but only real runtime data collection would let compare/analyze performance. Table VI shows the data provided by sequential and parallel runs instrumented program.

TABLE VI. INSTRUMENTATION DATA – PARALLEL PROC. – PROD. ENV.

Code	Seq. Time/s (s)	Par. Time/s (s)
Whole Program	13140	11820
subr1	6000	3420
subr1 1st Do	1200	740
subr1 2nd Do	1170	700
subr1 3rd Do	1200	640
subr1 4th Do	1200	630
subr1 5th Do	1170	640

The production environment facilities provide 4 CPUs (Intel I5-2310 @2.9GHz), which are used at runtime for parallel processing. As expected, the simulation model numeric output remains exactly the same (at the binary level) as that of the sequential (original) version. This also provides an objective verification (i.e. besides that of the scientists' knowledge/experience) about the parallelization correctness. More specifically, the indirect access of arrays assigned in the 1st, 3rd, and 4th subr1 subroutine do not lead to race conditions at runtime when processed in parallel.

Performance gain in the production environment is not as good as that in the development environment. More specifically, using 4 CPUs the subr1 subroutine runtime is reduced to about 50%, while the expected reduction would be 24%. In either case, the parallel runtime is reduced, and we are now taking advantage of the available processing facilities (which are always available). Besides, the performance gain in the subr1 subroutine encourages us to further parallelize the rest of the program, which involves 50% of the sequential runtime. Once the subr1 subroutine is parallelized, the original 50% of the sequential runtime "becomes" more than 70% of the runtime.

5 Conclusions and Future Work

We have carried out some standard and non-standard parallelization work on a shallow water numerical model. The parallelized version provides the same numerical output as that of the sequential version in all the experiments we carried out. Applying a "blind" parallelization does not provide acceptable performance results. However, we have been able to take advantage of previous parallelization

methodology in terms of the simulation numerical output correctness.

We have found several non-expected performance analysis data and performance results. On one side, we think those non-expected data and results are worth being documented in this paper and, on the other, they provide some insight in the short-term future work. We have found that the standard and well-known profile data and tools of GNU profiling are not always as accurate as expected. Besides, in legacy Fortran programs those tools not always provide enough information. More specifically, the runtime distribution in the code contained in a single subroutine has to be obtained by means of instrumentation code.

Even when we have provided specific performance and scalability data, we have to further investigate more thoroughly details such as:

- The difference in performance among the development and production environments, e.g. are they due either to processor architecture differences or complete system construction (e.g. different memory hierarchy)?
- Performance and scalability in computing platforms with more CPUs (and the corresponding threads).
- Some sequential well-known processing optimization techniques, e.g. block/tiling processing, which sometimes lead to "extra" parallel performance gains (or avoid parallel performance penalties).

6 References

[1] B. Roig, "One-Step Taylor Galerkin Methods for convection diffusion problems". Computational And Applied Mathematics, Vol. 204, pp. 95-101, 2007.

[2] O. C. Zienkiewicz, R. L. Taylor, The Finite Element Method, 6th Ed., Vol. 3, Butterworth-Heinemann, Oxford, 2005.

[3] M. Metcalf, J. Reid, M. Cohen, Fortran 95/2003 explained, 2004, Oxford University Press.

[4] W. S. Brainerd, Guide to Fortran 2008 Programming, 2nd Ed., 2015, Springer.

[5] R. van der Pas, E. Stotzer, C. Terboven Using OpenMP - The Next Step, 2017, MIT Press.

[6] GNU gprof. https://sourceware.org/binutils/docs/gprof/index.html

[7] F. G. Tinetti, M. A. López, P. G. Cajaraville, D. L. Rodrigues, "Fortran Legacy Code Performance Optimization: Sequential and Parallel Processing with OpenMP", Proc. 2009 World Congress on Computer Science and Information Engineering, IEEE Computer Society, March 31 - April 2, 2009, Los Angeles/Anaheim, USA.

[8] M. Méndez, Fernando G. Tinetti, J. L. Overbey, "Climate Models: Challenges for Fortran Development Tools", Second International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering, November 21, 2014, IEEE Computer Society, New Orleans, USA.

[9] B. Barney, OpenMP, Lawrence Livermore National Laboratory, 2017, https://computing.llnl.gov/tutorials/openMP

F. G. Tinetti, M. Méndez, A. De Giusti, "Restructuring Fortran legacy applications for parallel computing in multiprocessors", The Journal of Supercomputing, Volume 64, Issue 2, pp 638-659, May 2013.