EXPERIENCE IN PARALLELIZING LARGE APPLICATION PROGRAMS

Mario Juaçaba Teixeira

CEPEL – Centro de Pesquisas de Energia Elétrica
C.P. 2754
20001 – Rio de Janeiro – RJ – Brasil

ABSTRACT

This paper reports experience and describes the tools used in parallelizing large application programs of two kinds : Monte–Carlo simulation and linear programming optimization. These techniques are used in a large number of applications in power system operations and planning, and in various fields of engineering and science. Parallelism is analysed on the subroutine level and the programmer will be responsible for adapting the existing application code to the parallel environment. The tools comprise a multi–processor computing system with parallel processing software facilities, a concurrent processing simulator for helping on problem partition and debugging, and a data–base containing information on the application's variables and subroutines. The goal is to reduce execution times taking into account the solution structure. Three case studies illustrate the application of the above tools and the development methodology. Results obtained show a very high efficiency in the use of the concurrent processors.

RESUMO

Este artigo descreve a experiência e as ferramentas usadas na paralelização de grandes programas de aplicação de dois tipos : simulação de Monte–Carlo e otimização em programação linear. Estas técnicas são empregadas em um grande número de aplicações de planejamento e operação de sistemas elétricos e em vários campos da engenharia e ciência em geral. O paralelismo é analisado no nível de subrotina e o programador será responsável por adaptar o código existente ao ambiente paralelo. As ferramentas compreendem um sistema multi–processador com facilidades para programação de aplicações paralelas, um simulador para ajudar na partição do problema e sua depuração, e um banco de dados contendo informações sobre as variáveis e subrotinas da aplicação. O objetivo é reduzir os tempos de execução levando em conta a estrutura da solução. Três casos–estudo ilustram o emprego das ferramentas acima e a metodologia de desenvolvimento. Os resultados obtidos mostram uma alta eficiência no uso de processadores concorrentes.

I – INTRODUCTION

The advantage of parallelism over a sequential approach in accelerating the solution of a number of scientific and engineering problems is now a widespread concept. Techniques on how to design programs specifically to parallel machines are already being proposed [1,2]. On the other hand, although much effort has been spent towards finding a completely transparent way of taking large sequential applications and parallelizing them [3], up to now there is no general solution. Measures of potential parallelism on a high level statement basis in large applications [4] indicate that they would benefit from this technique as soon as the hardware and software that can exploit them are available.

In the meantime, programmers who wish and need their applications to benefit from existing hardware and software will be responsible for parallelizing them [5]. Their main concern will be the conversion of tested applications running on a sequential machine to a parallel environment, in a fast, reliable and efficient way.

The particular applications we deal with concern very large simulations used in power system planning and operations [6,7,9]. Today, these simulations try to circumvent the computational limitations of conventional computers by considering reduced electric systems or analysing a limited number of cases, for instance. Minimum cost planning alternatives and security analysis are examples of applications which would directly benefit from extending this scope. Besides, the methodology used for these problems is general, and has application in many other fields.

Given an application that is suitable for parallelization – in the sense that it allows a partition in "coarse grains" which are themselves parallelizable –, the applications' programmer faces the following question : which tools shall be used to help to convert the application to a parallel environment? Programmers are not able to explore parallelism on a statement basis, but will be interested in getting the benefit of a parallel computer at a procedure level.

This paper describes experience acquired doing this work and the tools which were developed to help in this task. Existing algorithms were implemented in a 16–cpu system, and benchmarked against their serial counterparts. The paper does not have the intention of being completely general, but rather to provide very simple and successfully tested tools for the development of parallel programs.

The following hypothesis are being made throughout the text :

- parallelism will be treated on the procedure level
- applications are in principle suitable to parallelization
- the programmer will be responsible for the parallelization (what means that his/her expertise will be needed)
- FORTRAN is the programming language
- a parallel machine with part of its addressing space shared by all processors is available.

The last item is not mandatory and has been included because it may lead to simpler code in actual programming.

The next section of the paper describes the code structure in the applications used as test cases. Section III describes the tools used and the results obtained in real cases. The tools comprise a data–base of routines and variables, a parallel processing simulator and a multi–processor system. The summary and conclusions are given in section IV.

II – STRUCTURE OF THE PROBLEMS

The applications used as test cases are of two types : Monte–Carlo simulation and linear optimizations problems.

Monte–Carlo simulation is a widespread tool used in applications ranging from physics to economics to power system security analysis. This method models uncontrolled variables as random and studies the statistical behaviour of the system under various inputs. On the other hand, large linear optimization problems with hundreds of thousands constraints arise in power system planning [7]. These problems may be decomposed by the Benders algorithm [8] leading to smaller size problems which are themselves amenable to parallelization.

In both kinds of applications there is one very big loop which is parallelizable. Nevertheless, it is not possible to write in the code some construct like "PARALLEL LOOP" and hope the existing compilers will do all the work efficiently. At this point the programmer expertise seems essential to decide on a number of practical issues that do arise.

II.1 – Monte–Carlo simulation

In a procedural approach, a Monte–Carlo simulation program may be characterized as a simple tree having three main branches : an initialization code, a simulation loop and a termination code (Figure 1).

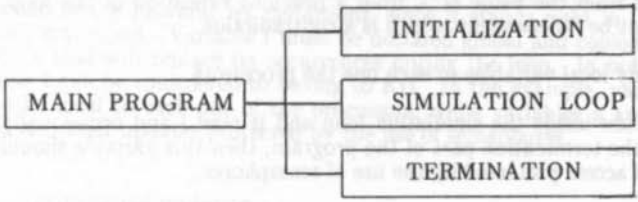

Figure 1 – Structure of Monte–Carlo simulation

The initialization code calculates the conditions for every simulation in the simulation loop. During the loop the variables which accumulate the results are updated. These variables are then output in the termination code, after the final calculations. For each run of the simulation loop there is a variable or a set of variables, called "seeds", which are random variables. The simulation loop goes on

until a maximum number of iterations is run or some convergence criterium is met. Each execution of the loop code is independent of results from any other loop execution.

Let us define the following sets of variables (which belong to COMMON blocks, i.e., are shared by more than one procedure) for this problem structure :

a. Constant group (KG) : all variables which are calculated (written) in the initialization part and read inside the loop.

b. Reduced safe group (RSG) : KG ∪ { random seeds }.

c. Safe group (SG) : to find the SG one needs to take the following steps –
a – do SG = RSG
b – add to SG all variables that are assigned values exclusively from variables in SG
c – repeat b until no more variables are added.

If a graph, with nodes representing the variables and the arcs dependences between variables (with one arc from a node to another if a variable is used in updating another), the problem of calculating SG can be formulated as finding all nodes belonging to paths starting from sources in RSG.

In the parallel implementation every processor will execute exactly the same code and only variables that need to be shared between processors will be placed in common memory. It can be affirmed that :

1 – If the simulations in the loop are independent then
SG = { all variables appearing in the loop }
Note : this can be used as a debugging tool.

Proof : Suppose there is a variable X which appears in the loop and does not belong to SG. X is written in the loop, because if it was only read then it would belong to KG. Besides, X = f(X,SG), since X cannot be only f(SG) – because it would belong to SG by definition – and it cannot be the output of a random function – X = f(). So the value of X is dependent from the value of X from a previous execution of the loop, and so the loop would not be independent, which is a contradiction.

2 – RSG will contain only local variables to each one the processors.

3 – If a variable is written inside the simulation loop and is read (and occasionally written afterwards) in the termination part of the program, then this variable should be declared global an its access protected by the use of semaphores.

Proof : let X be a variable in this case. If X is read after the simulation loop then it contains a result that is available only after the whole simulation is over, since one does not know when it will be updated. So, keeping it local to each processor would lead to erroneous results, since the contributions from other processors (which are also executing simulations) would not be taken in account.

This last result shows the importance of programmer's intervention : given that a variable is global, decisions on which and when variables should be locked depend on the particular implementation of the code, which may vary widely.

The analysis of loop independence may be influenced by the existence of boolean conditions (C), i.e., if–then–else constructs. Some of the assignment statements considered before may be subject to these constraints. The following can be affirmed :

4 – Execution of either the THEN or ELSE clauses must not change SG. If it does, the independence of the simulation loop will be a function of the value of C.

In the latter case, if all variables involved in calculating C belong to KG, then it can be known a priori whether for a particular run the loop will be independent or not.

Figure 2 shows a simple programming example, where the loop in subroutine C is fully parallelizable.

```
PROGRAM A
DATA PAR /1.0/
COMMON /COM/ m(100),n
CALL B
CALL C(PAR)
END

SUBROUTINE B            SUBROUTINE C(PAR)
READ(5,*) m             DO 10 I=1,100
RETURN              10  n = n + m(I) * PAR
                        RETURN
```

Figure 2 – Programming example

From the example above one notes that the loop control variable I has not been addressed in the previous set definitions, since it stands right in the frontier of what can be parallelized. Variable I must be declared global and copied to a local variable ILOCAL that will replace its occurrences during the loop. In order to calculate SG, variable I can be considered to belong to KG. In the example, variables m and PAR would be local to each one of the processors, while variables n and I would be made global and their updates protected by the use of semaphores.

II.2 – Optimization problems

Large optimization problems may be decomposed according to the Benders algorithm into smaller size ones. These new problems may be viewed as having a master/slave structure (Figure 3). For example, the objective of security constrained dispatch with post–contingency corrective rescheduling is to determine a minimum cost operation point which will not lead to overloads, if any contingency out of a given list occurs, taking post–contingency corrective actions into account. The security-constrained dispatch corresponds to a very large optimization problem, which

is solved by Benders decomposition [7]. By this approach the problem is decomposed in one "base case" optimal dispatch ("master") and NC separate "post contingency" dispatch subproblems ("slaves"), each of these with a smaller dimension than the original problem. The "base case" and "post-contingency" subproblems are solved iteratively until the global optimum solution is reached. This problem can be interpreted as a two–stage decision process :

- first, one calculates the optimal dispatch operating point x_0 :

$$z = \min c^t x_0$$
$$\text{s.t.} \quad [a_0]\, x_0 \leq b_0 \qquad (1)$$

where x_0 is the set of state and control variables in a base case system configuration, and (1) represents the load–flow equations plus operating constraints.

- second, given x_0, one calculates a new operating point x_i (system state after the i–th contingency, $i \in [1,NC]$), such that

$$[a_i]\, x_i \leq b_i \qquad (2)$$
$$\| x_0 - x_i \| \leq \Delta_i \qquad (3)$$

where (3), the coupling constraints, represent the post–contingency rescheduling limits.

This decomposition algorithm was used as a basis for the parallel processing implementation : each processor is loaded either with the "base case" or with one of the NC "post-contingency" subproblems, as shown in figure 3.

The master sends information about its state **M** to the slave, which sends it back information S to eventually calculate a new state **M'**. The iterative process goes on until a given convergence criterium $\epsilon = f(M)$ is met. The slave part is in fact a big loop which can be analysed using the same criteria as before and, of course, fully parallelized (Figure 3). Differently from the previous case, it is now necessary to determine **M** and S_i, i.e., the information passed between the master and slave parts of the application.

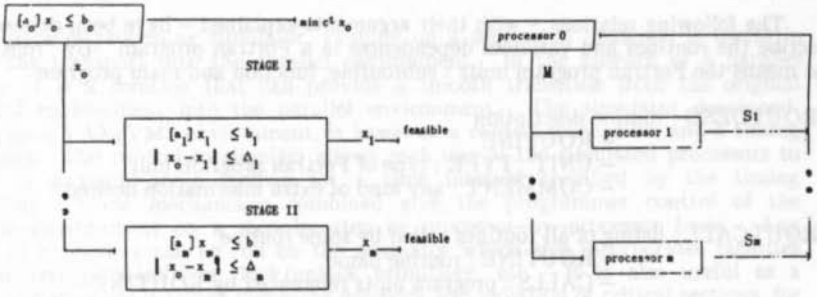

Figure 3 – Optmization problem structure and its parallel partition

It can be affirmed that :

5 – $M = M_w \cap S_r$, i.e., the variables written in the master and read by the slave processor.

6 – $S_i = S_w \cap M_r$, i.e., the variables written in the slave and read in the master processor.

In practice these statements may not be clear from the syntax of the program. For instance, S_i may be stored in one vector changing only the index or this area may be used by the master as a draft area after reading the information. The order in which the read/write operations are performed is also an important issue and must be considered when determining M, so that it will not be considered larger than it really needs to be. This question has not been tackled because, in general, the order in which operations are performed depends on the program execution and cannot be determined only by analysing its syntax. Programmer's intervention at this point seems again essential.

III – TOOLS AND RESULTS OBTAINED

Three tools were used to develop parallel versions of sequential programs on a procedure level :

- a data–base describing dependences between routines and variables
- a parallel processor simulator
- a multi–processor system with part of its address space shared by all processors.

The data–base was conceived to FORTRAN programs and takes in account both mechanisms in this language of passing variables between routines : arguments

declared on the subroutine statement or common blocks. The data–base is automatically generated from compiler listings. As a side–effect, this data–base is also being used as a documentation and development tool.

The following relations – with their arguments explained – have been defined to describe the routines and variables dependences in a Fortran program. By "routine" one means the Fortran program units : subroutine, function and main program.

- ROUTDESC : routine description
  - ROUTINE
  - UNIT_TYPE : type of Fortran program unit
  - COMMENT : any kind of extra information desired

- ROUTCALL : listing of all routines called by some routine
  - ROUTINE : routine name
  - CALLS : program units referenced by ROUTINE

- ROUTARGU : routines which bear arguments
  - ROUTINE
  - ARG_NAME : dummy argument name
  - NUMBER_ARG : relative position in argument list
  - STATUS : whether the argument is read/written

- CALLARGV : all calling references to routines which bear arguments
  - ROUTINE
  - CALLED_BY : program unit calling ROUTINE
  - ARGUMENT : actual argument used by CALLED_BY
  - NUMBER_ARG : relative position of argument in the original argument list

- ROUTCOMM : all routines which use common blocks
  - ROUTINE
  - COMMON : common name

- ROUTVARI : all routines and the variables they refer to
  - ROUTINE
  - VAR_NAME
  - STATUS : whether the variable is read/written in routine

- COMMVARI : variables in common blocks
  - COMMON
  - VAR_NAME

- VARIABLE : common block variables in the program
  - VAR_NAME : variable name
  - VAR_TYPE : variable type
  - VAR_DIMENS : variable dimension
  - COMMENT

The data–base is used as a powerful aid to the programmer in the task of partitioning the application, but it cannot be used as the only tool to analyze the code. For instance, using the same name of variable for completely different purposes

in two program sections is a semantic fact that cannot be discovered by a syntax analyzer. So, programmer's intervention in the program's parallelization task cannot be excluded.

A parallel processing simulator, running on a sequential machine, is useful in helping the transport into the parallel environment. In the absence of a parallel debugger it is a solution that can provide a smooth transition from the original sequential environment into the parallel environment. The simulator developed, running on a VAX/VMS environment, is based on a control mechanism and a timing mechanism. The control mechanism allows each one of the simulated processors to execute in a round–robin fashion for a time interval specified by the timing mechanism. These mechanisms combined give the programmer control of the execution environment on a step–by–step or processor–by–processor basis. Any number of processors can be run on the simulator which also incorporates facilities like send/receive primitives, lock/unlock primitives, etc. It is also useful as a debugging tool, since it allows observing deadlock and violation of critical sections, for example.

The multi–processor system being used, called the Preferential Processor (PP), consists of 16 processor boards, iAPX 286/287 based, with 128k of common memory. This prototype machine is the result of a joint effort from CPqD/Telebras for the hardware ([10]), and CEPEL, for the software ([11]). The 286/287 set will be upgraded to 386SX/387 particularly to improve floating point performance. PP presents to the user the very familiar PC–type interface, since each cpu runs DOS. The user has available services for communication/synchronization which are the same as those offered in the simulator.

Two programs applying the Monte–Carlo methodology were used as test–cases : a moderate size transmission reliability evaluation program [6] and a big multi–year power system planning and operation program [9]. The first one allowed a fast and error–free transition from the sequential to the parallel version. The speed–up's obtained on the PP may be resumed saying that for 16 processors the speed–up obtained was higher than 15. This impressing speed–up can be easily justified by the fact that practically the only information exchanged between processors are contributions to means and standard deviations, what is done after a long series of calculations [6].

The second application is the result of a joint effort from various power utility companies from the Northwest of the U.S.. It makes heavy use of files, even during the simulation loop. For sequential output files updated during the loop the solution taken was to make each processor write to its own private file (indexed by the loop control variable) and let one cpu take care of organizing the file as in the original sequential version. This program is currently running on the simulator and the results on simulator runs are shown in Figure 4. To run on the Preferential Processor the program will be re–structured so that memory restrictions might be circumvented.

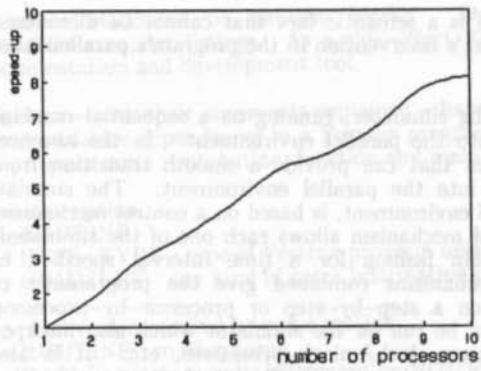

Figure 4 – System Analysis Model speed–up (simulator)

The optimal dispatch with security constraints was considered to illustrate the second kind of application. The parallel version of the security-constrained dispatch was tested with a configuration of the Brazilian electric system, with 504 buses, 880 circuits and 72 controllable generators with 20% corrective capacity. The objective function was the minimum deviation from the optimal operating point obtained without security constraints. The preventive dispatch was calculated for a list of 718 contingencies, corresponding to a linear programming problem with about one million constraints. With the decomposition algorithm, this problem is divided into one "master" problem and 718 "post–contingency" subproblems, each with about 1400 constraints. Its results are depicted in Figure 5. The lower efficiency can be explained by a tighter coupling between the master/slave parts of the program than that exhibited by Monte–Carlo simulation. What is being done now is to devise new mechanisms of cooperation between master and slave processors so that convergence will be accelerated and, in consequence, efficiency will be higher.

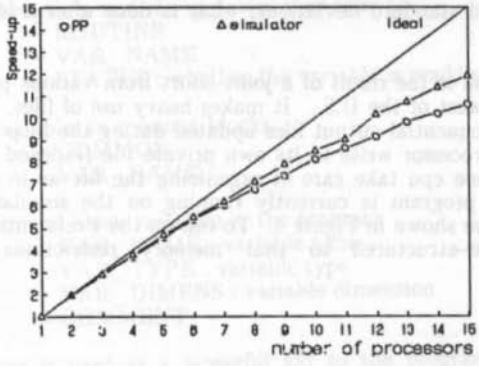

Figure 5 – Large linear optimization problem speed–up

IV – CONCLUSIONS

The paper described the experience in parallelizing large sequential applications of two classes : stochastic simulation and linear optimization problems. These types of problems arise in many power system applications as well as in many fields of engineering and science. The tools developed permitted a fast and reliable transition from the original sequential applications to their parallel versions on the multi–processor. This approach has prooved to be cost–effective and is now being employed to convert other programs which also allow "coarse grain" parallelism. An expert system might be useful to easen the decision–taking process by the programmer, for instance, in order to decide which variables are global, or to generate the code modifications in a user–friendly way.

REFERENCES

[1] M.J. Quinn, "Designing efficient algorithms for parallel computers". McGraw–Hill, 1987.

[2] M. Chandy, J. Misra, "Parallel program design", Addison–Wesley, 1988.

[3] D.A. Padua, D.J. Kuck, D.H. Lawrie, "High–speed multiprocessors and compilation techniques". IEEE Tr. on Computers, vol. C–29, no.9, Sep. 1980.

[4] M. Kumar, "Measuring parallelism in computation–intensive scientific/ engineering applications". IEEE Tr. on Computers, vol. 37, no.9, Sep. 1988.

[5] A.H. Karp, "Programming for Parallelism". IEEE Computer, pp. 43–56, May 1987.

[6] G.C. Oliveira, S.H.F. Cunha and M.V.F. Pereira, "Direct method for multi–area reliability evaluation". IEEE Tr. on Power Systems, pp. 934–942, November 1987.

[7] A. Monticelli, M.V.F. Pereira and L.M.V.G. Pinto,"Security–Constrained Optimal Power Flow with Post–Contingency Corrective Reschedulig", IEEE Transactions on Power Systems, Vol. PWRS–2, N0. 1, February 1987

[8] L. Lasdon, "Optimization theory for large systems", New York, McMillan, 1970.

[9] System Analysis Model – Methods and Theory Manual, PNUCC System Analysis Committee, November 1983.

[10] CPqD, "Specification and characteristics of the Preferential Processor". Doc. PP.EEA.001/ CA–01–AB, 1987.

[11] M.J. Teixeira, M.V.F. Pereira, L.A. Terry and H.J.C.P. Pinto, "Environment for Developing Loosely Coupled Parallel Programs". 7th. SBA Conference, S.J. dos Campos, SP, Brazil, Aug. 1988.

ACKNOWLEDGMENTS

The author would like to thank M. Pereira for his helpful support and H. Pinto for the work involved in the implementation of the algorithms.