A Cray T3E implementation of a parallel stochastic dynamic assets and liabilities management model

G. Zanghirati a,b,*, F. Cocco c, G. Paruolo d, F. Taddei d

a Dipartimento di Matematica, Università di Ferrara, via Machiavelli 35, I-44100 Ferrara, Italy
b SMART s.r.l., via Don Minzoni 25, I-40057 Cadriano (Bologna), Italy
c Prometeia Calcolo s.r.l., via G. Marconi 43, I-40122 Bologna, Italy
d CINECA, via Magnanelli 63, I-40033 Casalecchio di Reno (Bologna), Italy

Received 21 December 1998; received in revised form 22 September 1999

Abstract

Asset and liability management (ALM) models represent an important tool for banks and finance companies to measure the volatility of expected revenues. These models – usually static and deterministic to fit conventional computer resources – may be much more useful if a dynamic stochastic simulation is adopted. This makes it possible to increase the precision of risk estimation. In this paper the parallelization strategy adopted to implement such a stochastic ALM code is described, together with porting and code performance issues. The very good timings obtained on a 128-proc Cray T3E are reported. Anyway the code is easily portable on other, possibly heterogeneous, high-performance computing platforms. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Asset liability management; Parallel simulation; Financial model; Cray T3E

1. Introduction

An asset and liability management (ALM) model is a mathematical and statistical model aimed at forecasting the future expected revenues of the financial firm and at
measuring the volatility of expected revenues. The dynamics of company revenues is
the result of the interaction of two factors: the dynamics of financial and credit
markets, which are the main source of uncertainty for the company, and strategies
with respect to the set of variables under the management’s control (assets and li-
abilities composition, pricing). The main goal of an ALM system is to support the
management in improving the performance of the company by optimizing capital
allocation among different business opportunities and by properly pricing products
and services with respect to the level of risk undertaken in different lines of business.
ALM systems are build up on three main blocks, as Fig. 1 shows. The topic of this
project is the core of the analysis tool. To be actually useful, the model addressed the
problem across multiple dimensions with respect to the firm: it considered multiple
types of risk undertaken by the firm (market risk, credit risk, liquidity risk, etc.),
multiple products and services which embed risk, multiple business units where risk
is taken.

In modeling the financial and economic environment, all the variables impacting
on the performance of the firm (such as interest and exchange rates, credits demand,
savings supply, etc.) should be taken into account where possible. Furthermore, as
far as different strategies must be compared not only on the basis of their expected
outcomes, but also for the uncertainty (i.e., the risk) linked to them, the model has
defined the stochastic process driving the dynamics of the environmental variables.
All these elements contribute to increase the size of the problem.

The two main approaches to the decision problem under uncertainty are the
optimization and the simulation methods. Both are aim at supporting managers in
the choice of the best strategy to be adopted across the temporal horizon (defined as
a discrete sequence of time steps, with the single time step value chosen in the order
of weeks, or months, or years). In both cases a set of economic–financial scenarios
have to be generated and computed, and are used for the model evolution (CALM
approach, see [12,22]).

The optimization approach implies translating the case to be studied into a
constrained minimization problem, usually linear or quadratic and with linear
constraints [9,17,35,39,43,44]. The most recent general formulations are multi-stage
recourse problems where the objective function is a sequence of nested optimization
problems [12,18,23,29]. In general, the constraints and the objective function de-
pend on the scenarios. The solution of the constrained optimization problem
provides the optimal strategy (as a set of subsequent decisions) in the considered

\[ \text{Data Warehouse} \] where all information needed for
\[ \text{Analysis Tools} \] the dynamic simulation model of
\[ \text{Reporting Tools} \] future income
\[ \text{and} \] for communicating the results of the
\[ \text{and} \] analysis back to decision makers

Fig. 1. Main blocks for an ALM system.
scenarios. The computing power needed is substantial. In fact the number of variables in the problem grows at least linearly with the number of scenarios and exponentially with the number of portfolio rebalancing dates (in a lattice setting). Even exploiting the possible structural features, like sparsity in the matrices defining the constraints, the computational cost becomes rapidly unacceptable on traditional sequential platforms. For example, Consigli and Dempster [12] recently presented a study where the largest dynamic ALM optimization problem solved on an IBM R/6000 workstation involved 2688 scenarios. To solve it more than 1000 s of CPU time were needed, using state-of-the-art constrained linear optimization software tools. On the other hand, Gondzio and Kouwenberg [30] succeeded in solving a very large ALM stochastic programming problem with more than 24 million variables and 12 million constraints in about $14 \times 10^3$ s on a 16 RISC processors parallel machine, using a scenario lattice with more than 4 million scenarios at the end of the horizon. Some authors also addressed the problem of the constrained scenario generation [6,8,33,45], which will be a relevant point in the next development of PALMA project. However, it's not often highlighted how the pseudo-random numbers correlation problem is managed in the scenarios generation.

The simulation approach does not solve any minimization problem and follows the evolution of the model along the temporal horizon [9,14,48]. Lacking a computational burden, the advantage here is that it is possible to run a much larger number of scenarios. This permits to compute more accurately the probability distribution of the final results, thus allowing the managers to cope with the decision process without automatic suggestions, but with the help of a better estimate of the returns and the related risk levels. Furthermore, multi-factor interest rate processes used to model sophisticated term structure theories significantly complicate the computer implementation of other approaches (like lattice or finite difference), while Monte Carlo approach has a very natural extension of the one-factor models, thus becoming a more viable alternative [5,36]. At the present stage, PALMA project is a simulation tool and its features are best exploited in running a large number of scenarios. Anyway, the addition of a module devoted to automatic strategies optimization, of the kind previously described for optimization codes, is presently foreseen in the future evolution of the project.

A mixed simulation/optimization approach is also investigated by different authors [4,14,15], which seems to give promising results and to be a favorable framework for the future development of the PALMA project.

Several ALM simulation packages are currently available on the market [56], but they are mainly tailored to traditional sequential computers. Due to the reasons described above, a stochastic simulation might require several days of CPU time to be performed on traditional computing systems.

As a result, the ALM systems presently on the market are stretched between two options:
- to put severe restrictions on the level of detail used to describe the strategies of the firm and its economic environment, in order to maintain the capability of running truly stochastic simulations;
to ignore the stochastic nature of the problem to favor the care of strategies and environment description, thus running deterministic simulation to compare the effects of different strategies and different economic scenarios.

Both these approaches are unsatisfactory. The first one makes it difficult for managers to derive the correct decision steps to be undertaken, due to the high level of product grouping and a simplified strategy description. The second one does not allow an accurate enough risk estimate.

The use of HPCN resources in this field allows to overcome these shortcomings and thus has to be considered as extremely promising, but it is also at a very early stage. Hence, PALMA should be considered as one of the first results in this area [13,23,24].

In this project, the authors implemented a dynamic stochastic model for distributed memory parallel platforms, adopting a modular, portable and scalable approach. The parallel implementation extracts parallelism both from controls and data and relies on standard message-passing interfaces (MPI) to produce a scalable code for platforms with an arbitrary number of nodes (possibly heterogeneous). The major objective of the project was to demonstrate the key relevance of HPCN in the implementation of decision support system for portfolio management of banks and financial firms. The implemented ALM model is specifically tailored for the Italian market and has been proved and validated on a real-world data set provided by a relevant financial firm, namely Credito Italiano (CREDIT).

The paper is organized as follows: Section 2 introduces the model, Section 3 presents the parallel algorithm, Section 4 describes the implementation of the scenario generator, Section 5 reports code performances and efficiency and Section 6 gives the conclusions. In the appendices, the interested readers can find more details about the parallel code optimization and the random numbers generation problem.

2. The dynamic asset and liabilities model

Two main modules, namely the Economic Scenario Generator and the Firm’s Model, form the dynamic simulation module: their interactions are briefly shown in Fig. 2, and their descriptions follow.

2.1. Economic Scenario Generator

This module is an econometric model to forecast the main economic variables determining the evolution of the balance sheet of the firm, i.e., the prices and returns of assets, interest and exchange rates, loans’ demand, etc., trough the temporal horizon. The firm’s financial risk is mainly determined by the uncertainties on the future evolution of these variables. In the proposed model, this uncertainty is modeled through a Monte Carlo simulation: at each time step of the horizon (one month in this work) a random quantity, extracted from a probability distribution defined in the model, is added to the forecast variable value. This random choice is
replicated many times, to obtain a cluster of evolutive scenarios representing the possible value fields of the variables of interest. The Economic Scenario Generator is described by a system of stochastic differential equations, modeling the impact of two sets of inputs on the dynamics of the relevant variables, as Fig. 3 shows.

This method, well known in the literature [1,4,16,21,34,40], permits the use of relatively simple tools to represent complex situations, like the change of stock prices volatility or non-gaussian distributions of returns, thus allowing a better fit of the real evolution of these variables.

2.2. The Firm’s Model

This model is a representation of the process of Firm’s profits and losses and balance sheet build-up. A computer simulation starts with a current balance sheet, including detailed maturity or re-pricing schedules, the associated rates and yields of those balances, forecasts of income statement, balance sheets, and cash flow schedules for a series of future time periods. This is accomplished by literally simulating the re-pricing maturities, rollovers, and new business originations for all

---

**Fig. 3. Relationships scheme of the scenario generator main components.**
balance sheet activities of the firm. To generate a plausible set of financial statements, assumptions must be made about a number of important issues, including target balances, maturity schedules and pricing assumptions for new business. The model permits to compute at each time step of the simulation the current profit and loss and balance sheet on the basis of two variable sets: stochastic variables determined by the economic scenario generator; control variables, represented by the amounts of different balance sheet accounts (bonds, stocks, real estate, mortgages, etc.) and prices of different products. Input and output flows, together with the fund accumulation process and regulatory constraints, are represented with a set of linear equations. Market equilibrium and arbitrage-free conditions are also included in the model, of course, but their description is beyond the scope of this paper (see [10] for more details).

For each elaborated scenario, the simulation process can be summarized as Fig. 4 shows. The planned addition of a “strategies optimization” module will allow an automatic restart of the process, taking the previous results into account in the current computations.

In the Credito Italiano data set that we use in our experiments, the bank balance sheet is organized in 11 main accounts containing more than 200 different products (short, medium and long-term loans, bonds and financial assets, liquidity, demand deposits, time and savings deposits, etc.). At the end of the simulation, a comparison is made between the current balance sheet, income statement and interest rate sensitivity and previous forecasts of the same variable, and between the updated and previous forecasts. The results of the simulation are also compared with the profitability targets and the acceptable risk profile of the company defined in the

![Bank Balance Sheet and Activity Description](image)

Fig. 4. Scheme of the simulation process.
planning process, and with the estimates of profitability produced by the different business units. The model’s outcomes are therefore used to adjust balances and income statement target stated in the business plan of the bank.

3. The parallel algorithm

In this section, we briefly outline the implemented code parallelization: the interested reader is referred to Appendix A for more details. The code was realized with C++ using object-oriented programming for the Cray T3E present at CINECA [2,3,7,46,49,50]. This supercomputer is a 3D torus MIMD architecture currently with 256 DEC Alpha 21164 Processors (PEs) and a 1.67 ns clock cycle. They are “partitioned” into two sets of 128 PEs each: one set has 256 MB local RAM, while the other has 128 MB local RAM, for a total of 36 GB global memory. The peak performance achieves 192 GFlops. A very good communication speed is also achieved: 480 MB payload bandwidth through each one of the 6 torus directions. At the time of code testing, only the first processor set was available for study.

The model exhibits a coarse-grain parallelism, and its basic scheme can be summarized as shown in Table 1, where \( N_{\text{PEs}} \) is the number of PEs involved in the simulation and \( N_{\text{scn}}^{(i)} \) is the number of scenarios to be elaborated on \( i \)-th PE.

This stochastic parallel model could be logically described as of SPMD type, but in reality the implementation is MIMD type. The core of the model is structured into three main tasks:

1. Read company data from files (RFF phase). These data have been extracted from the company database and qualify its patrimonial status and its financial products at the starting date. The data are organized into several files, to be read at the beginning of point 2.1 in Table 1.

2. Generation of new scenarios and elaboration of the data through the temporal horizon (elaboration phase).

3. Collection of the results for the post-processing, i.e., store data to files (STF phase).

Table 1
Non-optimal pseudo-code showing the coarse-grain parallelism of the simulation model

\[
\text{for } i = 1, \ldots, N_{\text{PEs}} \text{ in parallel} \\
\quad 1. \text{compute } N_{\text{scn}}^{(i)} = \text{number of scenarios to be elaborated on } i\text{th PE} \\
\quad 2. \text{for } j = 1, \ldots, N_{\text{scn}}^{(i)} \text{ do} \\
\quad \quad \text{/** main local loop **/} \\
\quad \quad 1. \text{read initial data} \\
\quad \quad 2. \text{generate a new scenario and flush model structures} \\
\quad \quad 2.3 \text{elaborate current scenario trough temporal horizon} \\
\quad \quad 2.4 \text{store computed data} \\
\quad \text{end} \\
\text{end}
\]
The choice of the simulation starting date is free, provided that sufficient historical data are available.

The parallel implementation of point (1) would require access to the same files by all PEs. Moreover, it might be convenient not to retain the data in memory when they are a large amount, retrieving them anew from files for each scenario generation in order to free up memory space during the computation. This can be efficiently done in a sequential platform by using a database, but it represented a non-trivial problem to solve in the parallel code, where useless and chaotic data transfers can slow down execution. The problem was even greater when overlapped to the STF phase.

Therefore for efficient parallel porting of the code, I/O optimization is mandatory. The parallel algorithm that we finally implemented is described in Table 2, where \( N_{\text{rnd}}^{(i)} \) is the amount of random numbers needed by \( i \)th PE up to the end of the simulation.

Table 2
Parallel optimized pseudo-code

<table>
<thead>
<tr>
<th>for ( i = 1, \ldots, N_{\text{PEs}} ) do in parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. compute environmental variables</td>
</tr>
<tr>
<td>2. for each input data file do</td>
</tr>
<tr>
<td>2.1 alloc suitable MPI local buffer</td>
</tr>
<tr>
<td>2.2 if PE = master then</td>
</tr>
<tr>
<td>2.2.1 read data from disk and decode it in the MPI local buffer</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>2.3 broadcast data in master MPI buffer to all PEs</td>
</tr>
<tr>
<td>/**** implicit synchronization barrier ****/</td>
</tr>
<tr>
<td>2.4 store data in classes and free MPI local buffer</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>3. compute ( N_{\text{scn}}^{(i)} ) = number of scenarios to be elaborated on ( i )th PE</td>
</tr>
<tr>
<td>4. compute ( N_{\text{rnd}}^{(i)} ) = amount of random numbers needed by ( i )th PE</td>
</tr>
<tr>
<td>5. initialize random numbers generator and alloc local array for output buffer</td>
</tr>
<tr>
<td>6. for ( j = 1, \ldots, N_{\text{scn}}^{(i)} ) do</td>
</tr>
<tr>
<td>/**** main local loop ****/</td>
</tr>
<tr>
<td>6.1 generate a new scenario and flush model structures</td>
</tr>
<tr>
<td>6.2 elaborate current scenario through temporal horizon</td>
</tr>
<tr>
<td>6.3 store computed data in local output buffer</td>
</tr>
<tr>
<td>6.4 if buffer is full then</td>
</tr>
<tr>
<td>6.4.1 write buffer to disk and clean it</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>7. if buffer not empty then write buffer to disk</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

The related percentage refers to the whole execution time: the sum is less than 100% because of initialization, broadcast and system overhead, which however result almost negligible.
Finally, in Table 4 we give an idea of what kind of simulations the current version of the code can reach on the Cray T3E, performing a 12 months temporal horizon on the real-world data set provided by Credito Italiano, without requiring intermediate file output during the internal main loop. Note that 242 MB of RAM out of the 256 available on each PE are left to output data. We just mention that to complete a simulation for three forecast scenarios and two different strategies, the code installed on the bank mainframe requires about 7 h. Further details are given in Section 5.

4. Scenario generator implementation

In the implementation of the scenario generator, we followed Prometeia’s analysis of both the Italian financial market and the company model, as described in [10]. This task is really “local” to each PE during the execution of the main loop, after the initialization of the random number generator (see Table 2). From the parallelization viewpoint, it must be ensured that random numbers generated on one processor neither overlap nor be correlated with those generated on other PEs. This issue is of great importance for the success of stochastic simulations: in the authors’ case a suitable initialization of pseudo-random generators can be
made at the very beginning of the program, ensuring that the previous conditions are verified. In the following subsection, we describe the scenario generation strategy in the hypothesis that a sufficient amount of well uncorrelated “random” numbers are available to each processor. Then the issue of parallel generation of random numbers is addressed.

4.1. Coding the mathematical formulation

Analysis results are based on historical data from two sets of main financial interest indices, called, respectively, endogenous and exogenous variables: the latter are supposed to be known along the temporal horizon (from previous estimations), while the former have to be computed. Using these historical data, a covariance matrix $G$ is defined, expressing mutual correlations between endogenous variables. Furthermore, a set of coefficients is estimated to express dependencies at (discrete) time steps $t_k > 0$ of endogenous variables on both endogenous and exogenous variables at some previous or contemporary time steps $t_j < 0$.

Let $x_i^{(k)} = x_i(t_k), \ i = 1, \ldots, N$, the endogenous variables and $y_j^{(k)} = y_j(t_k), \ j = 1, \ldots, M$, the exogenous variables at time step $t_k$. The evolution of endogenous variables may thus be expressed by means of a system of delay stochastic equations having the form

$$x_i^{(k)} = f_i(r^{(k)}, x^{(k-d_1)}, \ldots, x^{(k-d_1)}; y^{(k)}, y^{(k-d_2)}, \ldots, y^{(k-d_2)}), \ i = 1, \ldots, N, \quad (1)$$

where $d_1 > 0$, $d_2 \geq 0$,

$$x^{(k-p)} = x(t_{k-p}) = (x_1^{(k-p)}; \ldots; x_N^{(k-p)})^T, \quad p = 1, \ldots, d_1,$$

$$y^{(k-q)} = y(t_{k-q}) = (y_1^{(k-q)}; \ldots; y_M^{(k-q)})^T, \quad q = 0, \ldots, d_2,$$

where $f_i$ is a linear function. The values $d_1$ and $d_2$ express the delay level, i.e., the values of how many time steps behind the current one affect the current vector $x$. The vector $r^{(k)} = r(t_k) = (r_1^{(k)}; \ldots; r_N^{(k)})^T$ is the pseudo-random vector which gives the basis for the stochastic simulation.

Let $t_K = T > 0$ be the final time step of the simulation (i.e., the end of the temporal horizon). For each time step $t_k, \ 1 \leq k \leq K$, $x^{(k)}$ is calculated as follows: $r^{(k)}$ is generated, then its components are linearly transformed to take the mutual correlation into account, and finally the function $f = (f_1, \ldots, f_N)$ is applied. The above-mentioned second step is obtained via the Cholesky decomposition of matrix $G$:

$$\tilde{r}^{(k)} = Lr^{(k)}, \quad G = LL^T, \quad (2)$$

where $L$ is the non-singular lower triangular Cholesky factor of the positive definite matrix $G$. This transformation clearly implies that an ordering of the components of $r$, hence of $x$, be defined: this is indeed what happens, based on the dependency analysis outlined in [10].
Now, given that \( f_i, i = 1, \ldots, N, \) are linear functions, they can be expressed as
\[
\begin{align*}
f_i(r_i^{(k)}, x^{(k-1)}, \ldots, x^{(k-d_1)}, y^{(k)}, y^{(k-1)}, \ldots, y^{(k-d_2)}) &= g_i^{(k)} + \ell_{i1} r_i^{(k)} + \cdots + \ell_{iN} r_i^{(k)} + b_{i1}^{(-d_1)} x_i^{(k-1)} + \cdots + b_{iN}^{(-d_1)} x_N^{(k-1)} + \\
&+ b_{i1}^{(-d_2)} x_i^{(k-d_1)} + \cdots + b_{iN}^{(-d_2)} x_N^{(k-d_1)} + c_{i1}^{(0)} y_i^{(k)} + \cdots + c_{iM}^{(0)} y_M^{(k)} + \\
&+ c_{i1}^{(-d_2)} y_i^{(k-d_2)} + \cdots + c_{iM}^{(-d_2)} y_M^{(k-d_2)},
\end{align*}
\]

where \( \ell_{ij} \) are the elements of \( L \) and \( g_i^{(k)} \) are “constant” terms: they depend upon the equation number and on the month corresponding to time step \( k \), i.e., \( g_i^{(k)} = g_i^{(h)} \) when \( k \equiv h \pmod{12} \). To give the previous expression in matrix form, let us define the vectors
\[
\begin{align*}
v^{(k)} &= (x_1^{(k-1)}, \ldots, x_1^{(k-d_1)}, \ldots, x_N^{(k-1)}, \ldots, x_N^{(k-d_1)}, y_1^{(k)}, \ldots, y_1^{(k-d_2)}, \ldots, y_M^{(k)}, \ldots, y_M^{(k-d_2)})^T, \\
\mathbf{g}^{(k)} &= (g_1^{(k)}, \ldots, g_N^{(k)})^T,
\end{align*}
\]
of size \( S = Nd_1 + M(d_2 + 1) \) and \( N \), respectively, and the constant matrix \( D = (B \ C) \) of size \( N \times S \), where
\[
B = \begin{pmatrix}
b_{11}^{(-1)} & \cdots & b_{11}^{(-d_1)} & \cdots & b_{1N}^{(-1)} & \cdots & b_{1N}^{(-d_1)} \\
b_{21}^{(-1)} & \cdots & b_{21}^{(-d_1)} & \cdots & b_{2N}^{(-1)} & \cdots & b_{2N}^{(-d_1)} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
b_{N1}^{(-1)} & \cdots & b_{N1}^{(-d_1)} & \cdots & b_{NN}^{(-1)} & \cdots & b_{NN}^{(-d_1)}
\end{pmatrix}_{N \times Nd_1},
\]
\[
C = \begin{pmatrix}
c_{11}^{(0)} & \cdots & c_{11}^{(-d_2)} & \cdots & c_{1M}^{(0)} & \cdots & c_{1M}^{(-d_2)} \\
c_{21}^{(0)} & \cdots & c_{21}^{(-d_2)} & \cdots & c_{2M}^{(0)} & \cdots & c_{2M}^{(-d_2)} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
c_{N1}^{(0)} & \cdots & c_{N1}^{(-d_2)} & \cdots & c_{NM}^{(0)} & \cdots & c_{NM}^{(-d_2)}
\end{pmatrix}_{N \times M(d_2 + 1)}.
\]

Thus system (1) can be written as
\[
x^{(k)} = Dv^{(k)} + Lr^{(k)} + \mathbf{g}^{(k)},
\]
where \( \mathbf{g}^{(k)} = \mathbf{g}^{(h)} \) is stressed for \( k \equiv h \pmod{12} \). Given the intrinsic temporal dependency of each vector \( x^{(k)} \) upon previous ones, no further vectorization can be made. Thus Eq. (5) has to be applied recursively \( K \) times: each time, a new vector \( r^{(k)} \) is generated and a new \( x^{(k)} \) is computed from (5), and then vectors \( x^{(k-p)} \) and \( y^{(k-q)} \) in (1) have to be shifted one place to the right, i.e.,
\[
\begin{align*}
x^{(k-p)} &\to x^{(k-p-1)}, \quad p = 1, \ldots, d_1 - 1, \\
y^{(k-q)} &\to y^{(k-q-1)}, \quad q = 0, \ldots, d_2 - 1.
\end{align*}
\]
The previous $x^{(k-d_1)}$ and $y^{(k-d_2)}$ have to be removed from vector $v^{(k)}$ and finally both $y^{(k)}$ and the just computed $x^{(k)}$ have to be placed in $v$. Given that the computational complexity to generate vector $r^{(k)}$ is linear in $N$, the complexity for the system (5) is $NS + N^2/2 + O(N)$, thus the computational complexity of one full scenario generation is given by

$$R = K[N^2(d_1 + 1/2) + NM(d_2 + 1)] + O(N).$$

Generally speaking, $M$ is some fraction of $N$, say $N/4$ or less, and $d_2 = d_1$, so that $R = O(N^2(d_1 + 1))$.

In the implementation of the updating mechanism, the storage, memory retrieval, and cache alignment considerations are commended. First of all, the easiest way to manage the scenarios is using a matrix, say $Y$, whose $k$th row or column is $((x^{(k)})^T, (y^{(k)})^T)$. But which one is to be preferred? It should be considered that in the elaboration phase all the values assumed by each component $x_i$ or $y_j$ are often successively accessed, while in the generation phase the vectors $x^{(k)}$ and $y^{(k)}$ are accessed for a fixed $k$. Since it is expected that $K > S$, it is more convenient to stack $((x^{(k)})^T, (y^{(k)})^T)$ in the $k$th row of $Y$. In this way, the successive evolution values of an endogenous or exogenous variable are contiguous in memory and can be efficiently retrieved in data caches. On the other hand, during generation the same small number of contiguous rows of $Y$ is repeatedly accessed in a short time: even if pointers to these rows could be used, they would not allow maximum efficiency of the basic linear algebra routines (BLAS). For this reason, two temporary arrays are used as a sort of “window” on the rows of $Y$: one for endogenous variables, say $W_1^{(k)}$, sized $d_1 \times N$, and the other for exogenous ones, say $W_2^{(k)}$ with size $(d_2 + 1) \times M$. They can be written as

$$W_1^{(k)} = \begin{pmatrix}
  x_1^{(k-1)} & x_2^{(k-1)} & \cdots & x_N^{(k-1)} \\
  x_1^{(k-2)} & x_2^{(k-2)} & \cdots & x_N^{(k-2)} \\
  \vdots & \vdots & \ddots & \vdots \\
  x_1^{(k-d_1)} & x_2^{(k-d_1)} & \cdots & x_N^{(k-d_1)}
\end{pmatrix} \leftarrow (x^{(k-1)})^T$$

$$W_2^{(k)} = \begin{pmatrix}
  y_1^{(k-1)} & y_2^{(k-1)} & \cdots & y_M^{(k-1)} \\
  y_1^{(k-2)} & y_2^{(k-2)} & \cdots & y_M^{(k-2)} \\
  \vdots & \vdots & \ddots & \vdots \\
  y_1^{(k-d_2)} & y_2^{(k-d_2)} & \cdots & y_M^{(k-d_2)}
\end{pmatrix} \leftarrow (y^{(k-1)})^T$$

In this way, all the values needed in (5) are stored together, thus enabling considerable advantages from pre-fetching techniques in data caches. The vector obtained by columnwise storing first $W_1^{(k)}$ then $W_2^{(k)}$ is in fact the vector $v^{(k)}$: this requires no effort in implementation. At successive time steps, the updating mechanism corresponds to a one-place downwards shift of the rows in $W_1^{(k)}$ and $W_2^{(k)}$, followed by an update of their first rows with the newest $x^{(k)}$ and $y^{(k)}$. 
The matrix form (5) of the delayed system allows a very efficient implementation of the scenario generator, using perfectly optimized linear algebra routines common to all platforms: in fact the described technique needs only levels 1 and 2 BLAS routines. Their interfaces are highly standardized and they are now available also as C++ methods [26]. On Cray T3E, vector versions are available. Similarly, the LAPACK routine \texttt{SPOTRF} implemented in the optimized Cray scientific library \texttt{scilib} [25, 58] was used to exploit the Cholesky decomposition of matrix $G$ (with the associated \texttt{SPOCON} routine to check for ill-conditioning). This factorization step could be removed if the factor $L$ be given in place of $G$, but we retain it for convenience in view of data updating from Prometeia’s estimates. In all non-Cray versions of the code, double precision arithmetic is used throughout all scenario generation.

The amount of (pseudo-)random numbers needed by each PE, called $N_{\text{rnd}}^{(i)}$, is computed at the beginning of the program and is given by $N_{\text{rnd}}^{(i)} = NK_{\text{scn}}^{(i)}$. For the entire simulation $N_{\text{rnd}} = NK_{\text{scn}}$ (pseudo-)random numbers are needed (see Tables 4 and 6). In our implementation they are generated using \texttt{PRNGlib} [38] to correctly deal with the pseudo-random numbers long-term correlation problem. For additional informations about this relevant point, the interested reader is referred to Appendix B. In the experiments we sampled the random numbers from a normal distribution for simplicity, since this is of a negligible relevance for the performances analysis.

5. The stochastic simulation

In this section we give the results of the experiments. In all our tests, we used codes compiled with aggressive optimization enabled and the program was run from a directory enabling fast disk access. How much faster such access is than normal depends on the global amount of I/O requests of the system, and typically a factor of between 1.7 and 2 is reached. This affects only the STF phase, so that very few should be expected on the overall behaviour when there is no intermediate disk access during the main loop. When this is not so, only the loop time of the iteration involved in the disk access is affected.

5.1. Preliminary considerations

To test whether good use was being made of the resources available, we first ran a special version of the code, compiled to enable the use of hardware monitors. Indeed, a double analysis was performed: first using Cray MPP Apprentice [28, 31, 52], to evaluate the amount of parallelization effectively reached, and secondly using the Performance Analysis Tool [28, 31, 51], principally to evaluate memorization strategy quality.

The results with MPP Apprentice simply confirm the good scalability properties shown below (vide next section).

The results obtained with the Performance Analysis Tool are mainly concerned with single PE hardware use and workload balance. It was used to a biggest routines level, since the profiling itself influences the runtime behavior of the program: the
so-called “Heisenberg uncertainty” impedes the obtaining of significant results if high-time accuracy, i.e., a very low sampling rate, is required (for example, see [47, Section 6.3.3] for a more detailed discussion of the subject). Given the type of program parallelism, the workload balance is rather good and becomes almost optimal as $N_{\text{scn}}^{(i)}$ increases.

The production version of the code was used to evaluate parallel performances. To this end, two types of simulations were performed: in the first one 2048 scenarios per run were used, with each run being repeated 10 times in order to obtain confidence statistics (mean and standard deviation) about the time sampling; in the second simulation, 10240 scenarios per run were used, with only one repetition. This two-stage approach was motivated by the aim of estimating system-dependent deviations in measures, due to other contemporarily active system processes.

The basic statistics considered were thought to lead to a sort of “system immunization”. Averages and standard deviations were used precisely because they are affected by outliers, so code branches sensitive to system overhead were able to be detected. Nevertheless, the batch queuing policy forces jobs requiring a large number of PEs to run during the night or week-end: system-dependent statistics of simulations involving more than 64 PEs could thus be different from the ones given by jobs run during the normal production hours of the Cray T3E. Finally, it should be noticed that 10 repetitions really do not constitute a sufficiently wide set of statistical samples. However, this seemed to be the best compromise between system resource and code evaluation targets.

5.2. Experiments

The two sets of simulations performed differ in the number of scenarios and in the number of repetitions: in case of 2048 scenarios and 10 repetitions, “averages of standard deviations” means that for each monitored time sample the standard deviation was calculated with respect to the current number of PEs and then the mean of the 10 resulting values was taken. In case of 10240 scenario simulations, meanwhile, the standard deviations across the processors are taken simply, given that only one repetition is made. When not otherwise stated, “average” means exactly the mean of all available samples.

Let us analyze the 2048 scenarios simulation first. In this case, the number of scenarios elaborated by each processor ranges from 1024 with 2 PEs to 16 with 128 PEs. Fig. 5 reports the averaged overall behavior of the whole program and of each PE, together with the related mean standard deviations. It is shown how well the code scales and the success of the code optimization strategies: communication times and parallelization overhead remain almost constant and negligible with respect to the computational time, at least until the latter is of a considerable amount. Things may change for a larger number of processors, because the time spent in computation becomes smaller; however, good qualitative behaviour is preserved, as can be seen from Fig. 6.

In particular, third picture in Fig. 6 shows that the measure distortions are concentrated in the STF phase, as predicted, where conflicts with other system jobs
appear. The very small values of the local loop standard deviations shown for 128 PEs have been observed for all the other simulations too: hence the mean values are well representative. Furthermore, the total main loop time remains almost constant across all PEs, thus by averaging it over all the simulations it gives a quite good estimate of the main loop time needed by each scenario: this is shown in Table 5.

This information was used to have an estimate of how much system time a whole simulation might require, as shown in the last column of Table 6.

In the fourth picture of Fig. 5 oddly high values of elaboration phase standard deviations for 4 and 8 processors are shown. After checking the data, it was discovered that they are due to only one PE of the corresponding simulation, probably due to some system interrupts which happened during those simulations. Anyway, the values are negligible if compared with the related averaged times (295.4 and 146.8 s, respectively) and by removing these two outliers standard deviations lower than 0.7 s were obtainable, according to the other simulations.
The situation is quite different for the STF phase statistics: higher variability is shown due to disk access conflicts, compensated only in part by the decreasing amount of data to output. The maximum is reached for 128 PE: once again this is not only due to system interaction, but also to the fact that computational time in this case is of the same magnitude of STF time. Given that statistics may hide the details of simulation behavior and that this is the most critical part of the code, it is

![Fig. 6. Time distribution and standard deviations of global phases times and of main loop phases times with 128 processors, in the case of 2048 scenarios.](image)

<table>
<thead>
<tr>
<th>Local main loop phases</th>
<th>Seconds</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario generation and lists flushing</td>
<td>0.16</td>
<td>0.0052</td>
</tr>
<tr>
<td>Scenario elaboration</td>
<td>0.58</td>
<td>0.0331</td>
</tr>
<tr>
<td>Store output data in memory buffer (no disk access)</td>
<td>0.11</td>
<td>0.0006</td>
</tr>
<tr>
<td>Total</td>
<td>0.85</td>
<td>0.0358</td>
</tr>
</tbody>
</table>

The situation is quite different for the STF phase statistics: higher variability is shown due to disk access conflicts, compensated only in part by the decreasing amount of data to output. The maximum is reached for 128 PE: once again this is not only due to system interaction, but also to the fact that computational time in this case is of the same magnitude of STF time. Given that statistics may hide the details of simulation behavior and that this is the most critical part of the code, it is
interesting to see how it changes. Fig. 7 shows the increasing influence of this phase, up to the very critical last situation. It demonstrates that the uncertainty in run times is due to the STF phase only, while RFF and broadcast phases give a negligible contribution.

Finally, Fig. 8 reports (averaged) code performances. The main loop exhibits very good scalability and efficiency, as it was expected from the type of code parallelism. The small super-linear speed-up should be considered as a statistical side effect. The other interesting contribution is found in the STF phase: not surprisingly, it gives the worst case. Indeed, the low level mechanism for writing binary data to disk is faster if only one processor asks for it, but also the physical location of the PE running the program and a free output channel available are important. For all these sources of uncertainty, speed-up and efficiency analysis of the STF phase has a very low confidence level. In our case, the mean of the 10 STF phase time samples on a single PE simulation is 0.2 s, which became 13.3 s for the two-processor simulation, even if each one has to transfer half the data. This is the reason why the corresponding line is practically 0. Both the initialization and the read-and-broadcast phases efficiency decrease as $N_{PE}$ increases, as is evident: at any rate, their contribution to global performances is of no importance. Hence, globally, the code exhibits very good performances (solid line). The efficiency is quite good too, but for 128 PEs we have a minimum due to the critical STF phase (compare last picture in Fig. 7).

For the simulations with 10240 scenarios, the two pictures on top of Fig. 9 report the overall behavior with respect to time, which is similar to the one discussed above. However, it is worth remembering that in this case array size needed by each PE to store the output data does not fit in memory when configurations with 1, 2, or 4 processors are used. This means that a smaller memory buffer is allocated, and intermediate disk accesses take place to write the data when the buffer is full. While poor performances were to be expected, the results show an acceptable degradation with respect to the global times, as the very low super-linear speed-up in Fig. 10 shows. Once again, we stress that disk access conflicts at the end of the simulation becomes noticeable only for a large number of PEs: compare the two pictures at the bottom of Fig. 9. Fig. 10 presents scaled speed-ups and efficiencies. Given the

<table>
<thead>
<tr>
<th>Number of simulation months</th>
<th>Per PE max. number of scenarios</th>
<th>Total number of scenarios on 128 PEs</th>
<th>Amount of random numbers</th>
<th>Estimated run time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>1450</td>
<td>190720</td>
<td>$1.78 \times 10^7$</td>
<td>1200</td>
</tr>
<tr>
<td>24</td>
<td>775</td>
<td>99200</td>
<td>$1.90 \times 10^7$</td>
<td>1270</td>
</tr>
<tr>
<td>36</td>
<td>524</td>
<td>67072</td>
<td>$1.93 \times 10^7$</td>
<td>1400</td>
</tr>
<tr>
<td>48</td>
<td>395</td>
<td>50560</td>
<td>$1.94 \times 10^7$</td>
<td>1550</td>
</tr>
<tr>
<td>60</td>
<td>318</td>
<td>40704</td>
<td>$1.95 \times 10^7$</td>
<td>1700</td>
</tr>
</tbody>
</table>
previous discussion about the STF time sampling variability and since in the 10240 scenarios case
- only one simulation per configuration is run,
- intermediate disk accesses occur when 1, 2, or 4 PEs are used,
- the time spent for disk access remains negligible,
Fig. 8. Efficiency for the mean total times, in the case of 2048 scenarios.

Fig. 9. On top: global times taken by the different phases of the whole program and of the main loop phases executed on each PE, in the case of 10240 scenarios. At the bottom: total times of the whole simulation with 48 and 128 PEs.
it was decided not to consider STF samples in the speed-up analysis. There are very limited differences with respect to the same graphs of the 2048 scenarios case, but two things should be noticed:

- the computational effort in the main loop is much greater, so that the degradation due to the final STF phase for the 128 PEs simulation is much less important, raising speed-up and efficiency to a very good level;
- the main loop small super-linear speed-up is not due, this time, to statistical side effects: it appears because the code executed is not exactly the same when more than four processors are used, because of the intermediate disk swap of the memory buffer.

Finally, we wish to mention, without reporting results, that a simulation was attempted near the maximum allowed to fit the output buffer in memory: over 158,000 scenarios were elaborated on 128 PEs (1250 per PE) in less than 1000 s. At the end, each processor had to store an array of more than 200 MB to the disk, so that a disk full error aborted the job.

**5.3. Comparison with CREDIT’s tools**

On average, current Credito Italiano’s ALM code takes 1 h and 10 min on the bank’s mainframe to simulate one business strategy under one economic scenario. In order to compare the computational time between the existing scalar code and PALMA it must be considered that the existing model works at a more detailed level, simulating the effects of the economic scenario for each balance sheet position. From a theoretical perspective this approach gives a better accuracy but it is uselessly expensive, since the output is aggregated in a chart of accounts with a detail of about 200 products, as previously mentioned. The simulation of one scenario/strategy pair requires the same execution time for both the two systems, but while the existing model takes the same time for each further simulation, PALMA needs only 1 min, with a computing time reduction of about 60 times.

Since a stochastic simulation requires at least 10,000 scenarios to be robust enough, using the Cray T3E with 128 processors we can simulate more than 40,000

![Fig. 10. Scaled speed-up and efficiency for total times, in the case of 10240 scenarios.](image-url)
scenarios with a 60-periods horizon, in about 30 min. The same simulation would requires 168 h on a PC running the single-processor version of PALMA, while the existing mainframe code would be about 85,000 times slower [11].

6. Conclusions

In the present study, the implementation issues of a parallel simulation code for a stochastic dynamic model of the assets and liabilities management problem has been presented. Efficiency and portability aspects of the code, as well as different code optimization techniques have been analyzed. The ALM code, originally developed in Delphi by Prometeia Calcolo for scalar environments, has been ported and parallelized by SMART on the Cray T3E at CINECA, who contributed to the optimization and looked after its execution. The code has been tested on a real-world data set provided by Credito Italiano.

The experiments performed showed that very good speed-up results can be achieved in this growing field by means of HPCN technology. Further developments are planned in the near future, mainly concerning additional improvements of the computational core and of the web interface (currently developed by SMART), to allow the remote utilization of the code through the Internet. Furthermore, the addition of a “strategies optimization” module is under study, possibly enabling the code to achieve the status of a complete tool for financial decision-making support.

Acknowledgements

The authors acknowledge the help received by various members of SMART, Prometeia Calcolo and CINECA staff during the completion of this work. In particular the authors are highly grateful to P. Coriazzi for the statistical analysis on which the scenario generator is based, to D. Borrelli and G. Gervasio, who developed the first Delphi version of the code and provided helpful discussions about firm’s model and data warehousing, to G. Zuccaro who coded the scalar C++ version of the ALM system in Prometeia Calcolo, to M. Matteuzzi, developer of the web interface, to G. Erbacci and Elda Rossi for their extremely helpful suggestions. Thanks to the referees too, who helped us in improving and clarify the presentation.

Appendix A. Input/output optimization

In this section we sketch the relevant parallel issues of the implementation of algorithm in Table 2.
A.1. Read-from-files optimization

With reference to Table 1, the first thing to do is to move the RFF phase of each processor to outside the main loop so that only one disk access per input file is needed for each PE. It is evident that the initial data must be retained in the local memory of each PE and not released during the elaboration phase. At any rate, concurrent access to the same set of files by all PEs at startup should be reduced.

The time reduction factor that can be theoretically reached by parallel implementation is

$$r_1 \approx \max_{0 \leq i \leq N_{\text{PEs}}} N_{\text{scn}}(i) = \begin{cases} \left\lfloor \frac{N_{\text{scn}}}{N_{\text{PEs}}} \right\rfloor & \text{if } N_{\text{scn}} \equiv 0 \pmod{N_{\text{PEs}}}, \\ \left\lfloor \frac{N_{\text{scn}}}{N_{\text{PEs}}} \right\rfloor + 1 & \text{if } N_{\text{scn}} \not\equiv 0 \pmod{N_{\text{PEs}}}. \end{cases}$$

but this is not the case in practice, because of the I/O conflicts due to multiple concurrent accesses to the same file. The replication of the input data files is of some help for small values of $N_{\text{PEs}}$, but the benefit decreases as $N_{\text{PEs}}$ increases, while disk space required becomes rapidly unacceptable.

The dramatic improvement can be reached by making only one PE (the master) read data from disk and then broadcast them to all other PEs. This solution, while requiring more memory space in order to ensure an optimal data broadcasting, presents several good advantages:

1. I/O conflicts are completely removed in this phase;
2. an efficient use of direct disk access can be performed from the master processor, bypassing the automatic buffering strategies of the system;
3. the data broadcasting via MPI routines is faster;
4. the RFF phase becomes less sensitive to disk access requests of other users of the system;
5. in distributed-disk systems (like clusters of PCs or workstations) the input files can be stored on the master disk only, avoiding further migrations and simplifying security controls.

It is worth noticing that items 2 and 3 give the quality of a safe portability also to a cluster of different platforms.

A temporary MPI structure was implemented in the RFF phase for each input data file, as a buffer on each processor: the master reads the company data from disk files, decodes it in the local MPI structure, and then issues a broadcast to all the other PEs. Each node is thus enabled to proceed to filling its own local structures with the received data as soon as the transmission ends, asynchronously. After that, each node removes the temporary MPI buffer and waits for the next communication, until the completion of the input files.

An implicit synchronization barrier at broadcast call is unavoidable. However, non-blocking implementation of the MPI_Bcast routine \(^2\) can reduce the idle times

\(^2\) The non-blocking MPI_Bcast call could depend on the specific implementation of MPI [54, pp. 93–94].
As an example, compare the second row of Table 3, obtained with this implementation.

This technique requires more memory space because of the creation of the temporary structures: in fact, class organization of the company data is not very suitable for fast and efficient broadcasting because data contiguity cannot be ensured. MPI_Structures are the most general derived datatypes allowed in MPI [54, pp. 68] and ensure data contiguity. Furthermore, company input data files are organized in records and fields that need some decoding before being assigned to the C++ classes: this implies many loops spanning fields and classes, the same for all processors. A key point for the correctness of this phase is the correct use of structure addresses: both MPI_Address and MPI_UB was used to ensure a correct data transfer.

Once again, attention was paid to the portability of the code, keeping in mind the general idea of clusters of (possibly heterogeneous) computers.

The doubled memory space requirement, used to store data first in buffers and then in classes, does not represent a severe memory constraint because it is required for only one class at a time, via dynamic memory allocation and deallocation.

The new interfaces of MPI-2 for parallel disk accesses require substantially the same storage considerations to implement the reading of our input data files, but access conflicts still remain a matter. In fact, to perform a collective read from a file with a call to MPI/File read at all C routine (or its equivalent C++ method Read at all for an MPI/File object), de facto hides the conflicts due to multiple concurrent file accesses at the same position, hence resulting in the same inefficiencies as described before.

At the very end of this phase, all the processors have in their local classes all the data needed to start each simulation across the horizon. The broadcast calls performed up to this point represent the totality of the explicit inter-nodes communications needed: no data exchanges or remote memory references will occur until the end of the elaboration of all scenarios. This lack of communication is typical of Monte Carlo simulations [47] and, indeed, qualifies this problem as an embarrassingly parallel one.

A.2. Store-to-files optimization

The goal is to have all the data available, in order to perform an a posteriori statistical analysis. Basic statistics could be easily computed “on the fly” during execution, but more detailed analysis requires the storing of detailed data. A record-based output is also required, suitable for a post-processing performed with the help of either common databases or target-oriented customized codes. For these reasons, the evolutionary behavior of all the given company’s products have to be stored to files.

In our implementation, each PE stores its output data to an unshared sequential access file. Since portability is also a matter, the data should not be stored unformatted.
A reliable way is to compose the whole output for the current scenario in a local memory character buffer and then send it for output: this could be viable in a scalar or distributed-disk environment, but it still causes too many output requests (and hence long delays) on a shared-disk parallel machine.

The solution to efficiently exploit unformatted output while maintaining a character-oriented structure is to compose the output of all scenarios in a memory character array, formatted, and then perform an unformatted write to disk of this ASCII array. In the authors’ experience, this resulted in an improvement of the performances, but still it seemed sub-optimal. A satisfactory solution was reached by coupling “hardware” and software improvements: the automatic system buffering strategy for output was turned off, using a suitable flag in the open statement that allows the user to write binary data directly to disk.

This results in the expected speed-up for two reasons: first, the I/O requests to the disks are decreased by a factor of $N_{\text{scn}}^{(i)}$ and, second, the very minimum system overhead per node is involved. The last line in Table 3 gives the results for the previously considered example: in this case each scenario generates 3069 records, each one 55 bytes long, and the character array is about 2.7 MB. In this configuration, the same 12-month simulation could be performed up to about 1440 scenarios on each PE, for a total of more than 46 000 scenarios.

The speed-up is very favourable and the most extensive use of local PE memory is made, which is highly preferable in all distributed-memory multiprocessor systems. Many factors may contribute to increase the resource requirements, such as longer simulation horizon, larger number of company products to be monitored, more complex classes of product aggregation (macros), wider model specification of the company’s ALM, including for example internal transfers, etc. Thus, an adaptive strategy is applied to meet the available free memory space. During execution, each time the character array is full a file output is performed, clearing the array for reuse. With this solution a virtually unlimited number of scenarios can be elaborated by each PE. When an intermediate write-to-file occurred, poor performances were to be expected. However, we underline that this could be the case only when a very small number of PEs is used to work on a very large number of scenarios (or on a very large company’s data set, or on a long temporal horizon). Of course, the memory allocation step does not overlap local MPI input buffer allocation.

Here, the new MPI-2 interfaces could give some advantages: in fact, it is very simple to open a random access file where all PEs can asynchronously write data safely, starting at different offsets. However, to optimize the device access, system specifics in form of hints still have to be passed as parameters when the file is opened, thus reducing portability. Furthermore, the MPI_File_prealloc function (or the C++ Preallocate method) should be used to require the needed space to the system, but very large files are likely to be non-contiguous on disk: hence, system overhead to reference splitted view [55, p. 210] for the same PE costs almost the same as to reference two files. [55 p. 260]. MPI-2 enables different datatype representations allowing file inter-operability [55, Section 9.5, p. 246]: this could really be useful if the post-processing is performed with software that can import data stored in a portable unformatted way. In our case, the End-User requirements on output
format make the basic MPI_BYTE 3 data type the most appropriate for file output. Consequently, MPI-2 MPI_File_write function (or its equivalent C++ method Write) adds no gain to the standard C write function, while the Cray version of the latter has switches to enable the lowest overhead file access mechanisms as described before.

Finally, it has to be noticed that collective file operations are subject to the same restrictions as collective communication operations [55, pp. 258–265]. Upcoming releases of MPI standards will probably improve some of these drawbacks. Future developments of the code can get benefits from the use of MPI-2 I/O. However, we plan to conform the code as soon as possible to the C++ syntax described in the C++ bindings section of MPI-2 official document [55].

A.3. Elaboration phase

This phase is the computational core of the simulation. Its execution consists in a replication on each processor of the same code, with no communications: it gives the “embarrassingly” parallel nature of the simulation. The code is very similar to the one that runs on PCs, with very little modification to take advantage from some of the Cray T3E features not available on other platforms. These few modifications concern mainly memory allocation and the syntax of disk writing, as described in the previous subsection. Implementation details of this part are beyond the scope of this paper and, however, they have no specific parallel relevance, concerning only efficiency of a single processor and its own local memory use.

The very complex and multilevel mechanism is coded in C++ classes with a large use of lists, in a hierarchical way. Basically, for each iteration the three tasks of the main local loop in Table 2 do the following:

6.1: generate a new scenario (see Section 4), then restore initial data in the class structures by reinitializing the classes' hierarchy depending upon the new scenario;
6.2: elaboration of the current scenario consists in computing the dynamic evolution of each of the company’s financial products through the time steps until the end of the temporal horizon. In our case, time steps are given in months;
6.3: this task, together with 6.4, has been already described in Section A.2.

Part of the operations involved in task 6.2 are very similar to database queries (and they are, indeed, in the original PC software). To use a portable database or SQL queries could be a viable solution, but we preferred to avoid it mainly for two reasons: firstly, the project targets and the parallel porting level of the code does not require the use of many functionalities offered by an even minimal database kernel; secondly, C++ language features and flexible specialized classes (such as lists, tree, etc.) allow the efficient “custom” organization of data hierarchy. This implies both more memory space available for data and wider portability and flexibility in view of future updates and improvements: only a standard C++ compiler is needed.

3 This MPI data type works also as a “wildcard” for datatypes in MPI file I/O.
A.4. Cray T3E-specific issues

Since the stream buffers of Cray T3E are hardware devices not common to other computers, we decided not to plan the implementation for their extensive use, in the search for portability. However, compiler optimization options were switched on, together with the environmental variables to enable use of data and instruction streams [2,49,53].

Appendix B. The random numbers generation problem

As is well known, Monte Carlo codes base their capabilities on the assumption that a sufficiently broad set of well uncorrelated random numbers is available. A large number of research projects have been dedicated to this subject to date [19,27,32,41,42,57]. This effort has increased in recent years, since powerful computational tools and largely analyzed methods allow statistical and probabilistic theories to be correctly applied to a suitably large amount of results.

As stochastic simulations grow, the requisite theoretical properties of pseudo-random sequences might be lost, practically invalidating simulation results. Furthermore, problems are increasing in parallel environments, because different processors generate pseudo-random sequences which, even if “locally” well uncorrelated, can turn out highly correlated (even partially overlapped) to the sequences of other processors: this is the well-known and widely studied long term correlation problem [20]. Note, however, that it can appear in scalar environments too.

Given that we planned to use a huge quantity of pseudo-random numbers, system functions like RANF() or nrand() are not suitable to ensure the necessary properties to make the simulation statistically significant. To overcome this problem, while constantly taking portability into account, two excellent packages were considered: PRNGlib (parallel random number generator library) by N. Masuda and F. Zimmermann [38] and SPRNG (scalable parallel random number generator) from the NCSA [37]. The first one, chosen at the outset of the project, is written in Fortran, whereas the second is coded in C++. In the PRNGlib library “skip-ahead” routines are available to compute very rapidly the starting element of the pseudo-random sequence section owned by each processor. They can be exploited because of the possibility of computing the amount of pseudo-random numbers needed by each PE at the beginning of the program, i.e., \( N_{\text{rnd}}^{(i)} = N K N_{\text{scn}}^{(i)} \).

References


[47] Introducing the Program Browser, Cray Research Inc., pub. IN-2140 3.0.
[48] Introducing the MPP Apprentice Tool, Cray Research Inc., pub. IN-2511 3.0.