Automatic Tuning for a Systemic Model of Banking Originated Losses (SYMBOL) Tool on Multicore

Ronal Muresano, Andrea Pagano

***Abstract*—**Nowadays, the mathematical/statistical applications are developed with more complexity and accuracy. However, these precisions and complexities have brought as result that applications need more computational power in order to be executed faster. In this sense, the multicore environments are playing an important role to improve and to optimize the execution time of these applications. These environments allow us the inclusion of more parallelism inside the node. However, to take advantage of this parallelism is not an easy task, because we have to deal with some problems such as: cores communications, data locality, memory sizes (cache and RAM), synchronizations, data dependencies on the model, etc. These issues are becoming more important when we wish to improve the application's performance and scalability. Hence, this paper describes an optimization method developed for Systemic Model of Banking Originated Losses (SYMBOL) tool developed by the European Commission, which is based on analyzing the application's weakness in order to exploit the advantages of the multicore. All these improvements are done in an automatic and transparent manner with the aim of improving the performance metrics of our tool. Finally, experimental evaluations show the effectiveness of our new optimized version, in which we have achieved a considerable improvement on the execution time. The time has been reduced around 96% for the best case tested, between the original serial version and the automatic parallel version.

***Keywords*—**Algorithm optimization, Bank Failures, OpenMP, Parallel Techniques, Statistical tool.

I. Introduction

NOWADAYS economy analyses play a key role policy decisions. For this reason, the number of mathematical/statistical models used by academia and by practitioners is rapidly increasing. Most of them have been then translated into tools or software with the aim of simulating the behavior of the economy, the financial markets behaviour etc. However, these tools rely on models, which are more and more complex to obtain more and more accurate results. This of course requires high computational power to be executed in a reasonable amount of time.

Also, there are models that have been developed in serial and then they have been translated to parallel. However to perform this upgrade to the parallel architectures, we have to consider diverse key points such as: core communications, data localities, dependencies, process communications, memory size, network exchanges etc., in order to improve the performance metrics. For this reason, it is important to develop suitable strategies in order to manage the inefficiencies generated by the overhead added by the parallel library [1], in order to obtain benefits from such computational multi-core or parallel capacities and to improve the performance application metrics.

In this sense, this paper describes the adaptation techniques and the optimization process done on the SYMBOL model to improve its performance in two directions: execution time and scalability. SYMBOL is a statistical tool, which estimates the losses deriving from bank defaults, explicitly linking Basel capital requirements to the other key tools of the banking safety net, i.e. Deposit Guarantee Schemes, and bank Resolution Funds [2]. This tool has been used by Commission Services to prepare various Impact Assessments of European Commission (EC) regulatory proposals to enhance financial stability and prevent future crises (Capital Requirement Directive Proposal, Bank and Financial Institutions resolution Framework and Financial Transactions Tax). Moreover, SYMBOL is used to analyze the contributions of individual banks to total losses originated in the banking sector. This is an area of particular interest to policy-makers, as information on the factors determining risk contributions could be used in areas such as taxation of financial institutions (i.e. risk levies) and structural reform (e.g. analysis of too big to fail issues).

To improve SYMBOL, we have developed an optimization method that includes five steps. These steps allow us not only to optimize the tool; they also permit us to execute it in an automatic and transparent manner in order to exploit in the best manner the multicore architecture. Results obtained show an improvement of more than 90% comparing the original serial version with the new optimized parallel version.

Moreover based on the input data, the model select the best configuration (number of parallel thread) to minimize the execution time taking into account different aspects as: cores communications, data locality, memory sizes (cache and RAM), synchronizations, data dependencies on the model. Hence, this new approach takes advantage of the benefits of the computational power of multi-core architecture in order to improve the execution time with two goals: executing with large data sets and scaling the number or default scenarios. However, these goals have to consider obtaining shorter response time (drooping the execution time), and also using in an efficient manner the computational resources.

The paper is structured as follows: a brief description of the SYMBOL model in Section II. Section III describes a model and its step in order to make an automatic tuning of SYMBOL

Ronal Muresano and Andrea Pagano are from European Commission, Joint Research Centre (JRC), Institute for the Protection and the Security of the Citizen (IPSC), Scientific Support to Financial Analysis Unit, Via E. Fermi, 2749, CP. 21027 Ispra, Italy (Tel.: +39 0332-78-9315, e-mail: ronal.muresano@jrc.europa.ec.eu, andrea.pagano@jrc.europa.ec.eu).

The opinions presented here are exclusively those of the authors and do not in any way represent those of the European Commission.

on multicore environments. Section IV presents the optimization results for the SYMBOL model. Conclusions are discussed in Section V.

II. Systemic Model of Banking Originated Losses (SYMBOL) Model

The SYMBOL model simulates the distribution of losses in excess of banks' capital within a banking system (usually a country) using micro-data from banks' balance sheets. This distribution is derived aggregating at system level losses in excess of banks' capital of individual institutions in the system.

Individual banks losses are generated via Monte Carlo simulation using the Basel Foundation Internal Ratings Based (FIRB) loss distribution function. This function is based on the Vasicek model [3], which in broad terms extends the Merton model [4] to a portfolio of borrowers[1],based on an estimate of the average default probability of the portfolio of assets of individual bank. Usually, each SYMBOL simulation ends when 100,000 runs with at least one default are obtained[2]. The model can also be run under the hypothesis that contagion can start among banks, via the interbank lending market. In this case, additional losses due to a contagion mechanism are added on top of the losses generated via Monte Carlo simulations, hence additional banks may default (see also Step 4 below). The model can take into account the existence of a safety-net for bank recovery and resolution, where Bail-in (BiB), Deposit Guarantee Schemes (DGS) and Resolution Funds (RF) intervene to cover losses exceeding bank capital before they can hit PF.

Before entering into the technical details of the model, we briefly state its underlying assumptions:

1. SYMBOL approximates all risks as if they were credit risk;
2. SYMBOL assumes that the FIRB formula applies for all banks and adequately represents risks banks are exposed to;
3. Banks in the system are correlated with a given factor (see Step 2 below)
4. The only contagion channel is the interbank lending market (see Step 4 below)
5. SYMBOL assumes that all events happen at the same time (i.e. there is no sequencing in the simulated events, except when contagion between banks is considered).

We continue this section detailing steps/assumptions of SYMBOL.

A. Steps of SYMBOL Model

1. Estimation of the Implied Obligors Probability of Default (IOPD) of the Portfolio of Each Individual Bank

SYMBOL approximates all risks as if they were credit risk and assumes that the Basel FIRB approach appropriately describes credit risk banks are exposed to. The model estimates the average IOPD of the portfolio of each individual bank using its total MCR[3]declared in the balance sheet by numerical inversion of the Basel FIRB formula for credit risk. Individual bank data needed to estimate the IOPD are banks MCR and total assets, which can be derived from the balance sheet. All other parameters are set to their regulatory default values.

2. Simulation of Correlated Losses for the Banks in the System

Given the estimated average IOPD, SYMBOL assumes that correlated losses hitting banks can be simulated via Monte Carlo using the same FIRB formula and imposing a correlation structure among banks (with a correlation set to R=50%). This correlation exists either as a consequence of the banks' common exposure to the same borrower or, more generally, to a particular common influence of the business cycle[4]. In each simulation run j, losses for bank I are simulated as (1):

$$L_{ij} = LGD * N\left[\sqrt{\frac{1}{1-R}} * N^{-1} * (IOPD_i) + \sqrt{\frac{R}{1-R}} * N^{-1}(\alpha_{i,j})\right] \tag{1}$$

where N is the normal probability function and $N^{-1}(\alpha_{i,j})$are correlated normal random shocks, and $IOPD_i$ is the average implied obligors probability of default estimated for each bank in step 1. LGD is the Loss Given Default, set as in Basel regulation to 45%.

3. Determination of the Default Event

Given the matrix of correlated losses, SYMBOL determines which banks fail. As illustrated in Fig. 1, a bank default happens when simulated obligor portfolio losses exceed the sum of the expected losses (EL) and the total actual capital (K) given by the sum of its MCR plus the bank's excess capital, if any (2):

[1] The Basel Committee permits banks a choice between two broad methodologies for calculating their capital requirements for credit risk. One alternative, the Standardized Approach, measures credit risk in a standardized manner, supported by external credit assessments. The other alternative is the Internal Rating-Based (IRB) approach which allows institutions to use their own internal rating-based measures for key drivers of credit risk as primary inputs to the capital calculation. Institutions using the Foundation IRB (FIRB) approach are allowed to determine the borrowers' probabilities of default while those using the Advanced IRB (AIRB) approach are permitted to rely on own estimates of loss given default and exposure at default. These risk measures are converted into risk weights and regulator estimates of loss given default and exposure at default. These risk measures are converted into risk weights and regulatory capital requirements by means of risk weight formulas specified by the Basel Committee. The Basel risk weight formula for market risk is specifically calibrated version of the Vasicek model for credit portfolio losses. On the Basel FIRB approach see [5]-[7].

[2] This is needed to guarantee stability of the tail of the distribution simulated.

[3] Banks must comply with capital requirements not only for their lending activity and credit risk component. Banks assets are in fact not only made up of loans, and there are capital requirements that derive from market risk, counter-party risk, and operational risk, etc. The main assumption currently behind SYMBOL is that all risk can be approximated as credit risk. Except for very large banks with extensive and complex trading activities, this simplifying assumption is not excessively distortive as credit risk usually accounts for a very large share of banks' total capital requirements.

[4] The choice of the 50% correlation is based on [8] a discussion and a sensitivity check on this assumption can be found in [2].

$$L_{i,j} > El_i + K_i \tag{2}$$

The light-gray area in Fig. 1 represents the region where losses are covered by provisions and total capital are represented, while the dark-gray shows when banks default under the adopted definition.

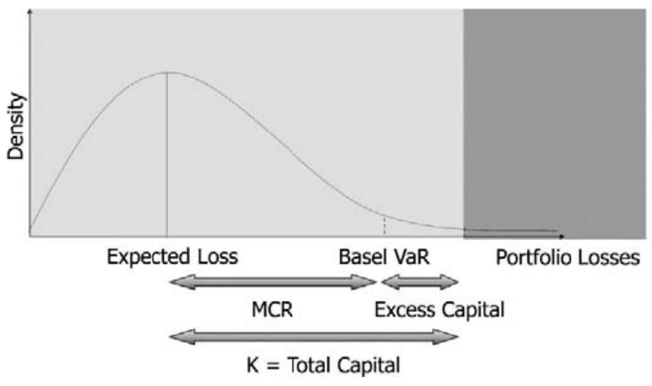

Fig. 1 Individual bank loss probability density function

More, the probability density function of losses for an individual bank is skewed to the right, i.e. there is a very small probability of extremely large losses and a high probability of losses that are closer to the average/expected loss. The Basel VaR is the Value at Risk corresponding to a confidence level of 0.1%, i.e. losses from the obligors' portfolio are lower that the Basel VaR with probability 99.9%. This percentile falls in the light-gray area as banks generally hold an excess capital buffer on top of the minimum capital requirement. Data needed for determining the default event for each bank is its level of total capital.

4.Contagion Mechanism (Optional)

SYMBOL can include a direct contagion mechanism since the default of one bank can compromise the solvency of its creditor banks, thus triggering a domino effect in the banking system. SYMBOL focuses on the role of the interbank lending market in causing contagion. The more a bank is exposed in the interbank market, the more it will suffer from a default in the system. In fact, the failure of a bank drives additional losses on the others, equal to 40% of the amounts of its total interbank debts.

As bank-to-bank interbank lending positions are not publicly available, an approximation is needed to build the whole matrix of interbank linkages. SYMBOL assumes that each bank is linked with all others and uses a criterion of proportionality to distribute additional contagion losses: the amount of losses distributed to each bank is determined by the share of its creditor exposure in the interbank market. In the case of a bank j fails, losses due to the contagion on bank k are equal to (3), where IB- and IB+ are respectively the interbank debts and credits of a bank:

$$L_k^{Contagion}{}_i = 40\% IB_j - \frac{IB_k^+}{\sum h \neq j IB_h^+} \tag{3}$$

A default driven by contagion effects occurs whenever additional due to the contagion channel loss causes any new bank default. This contagion process stops when no new additional bank defaults.

The magnitude of contagion effects depends on the two assumptions made:

First, the 40% percentage of interbank debits that are passed on as losses to creditor banks in case of failure, and, second, the criterion of proportionality used to distribute these losses across banks.

A loss of 40% on the interbank exposure is coherent with the upper bound of economic research on this issue, see e.g. [9]-[11]. Concerning the fact that the model distributes extra losses according to a criterion of proportionality, a sensitivity test has been developed in [12] in order to test if variations in the structure of the interbank positions systematically change the magnitude of contagion. The test, which varies the concentration of interbank linkages, demonstrates that losses SYMBOL are not relevantly affected by this proportionality assumption. Data needed to simulate contagion is the amount of interbank debts and credits for each individual bank.

5.Aggregated Distribution of Losses for the Whole System

Aggregate losses are obtained summing losses in excess of capital plus potential recapitalisation needs of all banks in the system in each simulation run. This includes both failed and non-failed banks, reflecting the fact that all banks in the system remain viable.

The model can also estimate the distribution of losses plus recapitalisation needs, i.e. the amount of capital required to maintain all banks in the system viable. Recapitalisation needs can be set equal to the Basel Minimum Capital requirements (4.5% or 8% Risk Weighted Assets (RWA)) and are estimated for non-defaulted banks when simulated losses erode this threshold.

These steps have been divided into three main parts (pre-processing, the SYMBOL execution and the post-processing). The first step is the pre-processing part in which the input data are generated, then the steps 2 to 4 are the main core of SYMBOL, where are simulated the economic scenarios of the bank system integrated into each country of the European Union. Also, this part compares the losses with the bank's actual capital. All this part was developed in a serial C tool. Finally, last part is the post-processing of the model, where we take into account different setting and asset the impact of foreseen reforms. Some examples of SYMBOL uses can be found on [13], [14].

III. A Method for Optimizing SYMBOL on Multicore Environments

Adapting an application to a multi-core node is not an easy task due to data dependencies many economic models depend on. For this reason, we have created a method for analyzing and executing SYMBOL considering as a key point the performance improvement (execution time, efficiency and scalability). There are researchers exploiting the parallelism on these architectures [1], [15]. Many of these solutions arising from such studies have allowed programmers to take a direction, in which are the best solutions to apply in the optimization process. In this sense, we have developed a set of steps, which allow us to evaluate the application's characteristics in order to define the optimization policies.

This method was mainly designed to tune SYMBOL; however, it can be clearly applied to other econometric models having similar behavior. Our method is integrated by five phases: code and model analysis (where we analyze the model and how it must be modified in order to increment the speed of the execution), designing optimization techniques (where all the key elements are evaluated and designed, hence modified to take advantage of the multi-core architecture), software implementation (here we apply all the changes proposed in the previous step and we create an automatic tool that allows SYMBOL to be executed in its best version (serial or parallel) in order to improve the performance). Finally the last two steps concern the testing version and data verification (where are validated that data obtained are 100% equal to the original data) (Fig. 2).

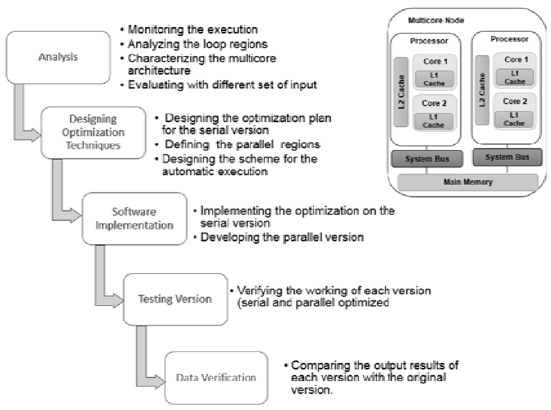
Analysis
• Monitoring the execution
• Analyzing the loop regions
• Characterizing the multicore architecture
• Evaluating with different set of input
Designing Optimization Techniques
• Designing the optimization plan for the serial version
• Defining the parallel regions
• Designing the scheme for the automatic execution
Software Implementation
• Implementing the optimization on the serial version
• Developing the parallel version
Testing Version
• Verifying the working of each version (serial and parallel optimized
Data Verification
• Comparing the output results of each version with the original version.
Multicore Node
Processor
Core 1
L1 Cache
Core 2
L1 Cache
L2 Cache
System Bus
Main Memory

Fig. 2 Method for improving and analyzing SYMBOL performance

These steps allow us to apply the optimization and parallel techniques, as to manage two main issues of SYMBOL with respect to the execution time: the first one is related to the execution with large input data (e.g. large banking systems) and second is when we are trying to scale the number of default scenarios. In other words, we create a new version improving the performance in terms of decreasing the execution time as well as enhancing the scalability of SYMBOL (executing with large input data and a bigger amount of defaults). At the end, final output result should be 100% equal to the original version.

Finally, our method not only deals with optimization techniques, but it also selects in an automatic and transparent manner the best amount of cores in order to take advantages of the multi-core architecture.

A. Code and Model Analysis on the Multicore Node

The goal of this step is to draw the behavior of SYMBOL in order to define the optimization policies to be applied on the multi-core architecture. However, we have to consider that SYMBOL has data dependencies behavior, which means that it does not always grow when the number of banks is increased. To observe this, we have executed SYMBOL using different sets of data input from different European Union (EU) countries. It has been executed with small input example as Lithuania (with 10 banks) and with big samples as Germany (with more than 1100 banks). The results of this analysis are detailed in Fig. 3, where we can observe that execution time will not only depend on the number of banks. If we observe, for example, the execution time for France (149 banks) and Spain (133 banks), we can see that France's time is much lower than Spain (almost divided by two).

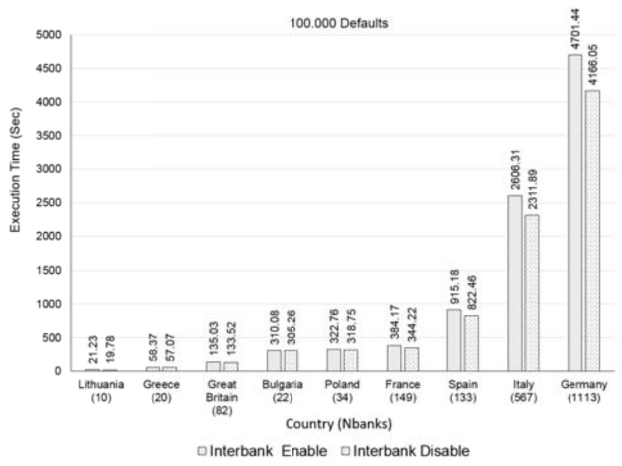

Fig. 3 SYMBOL Application analysis

A similar situation happens with Greece and Bulgaria in which the number of banks is similar, but the execution time varies almost of one order of magnitude, both when using the contagion part (interbank flag enable) or not (interbank flag disable). In conclusion, the execution time will depend on the data input executed as a whole, not just on their size (number of banks). The reason of this behavior is because each input data can need a different amount of iterations to get the same default scenarios.

On the other hand, the other parameter, which affects the execution time, is the number of defaults that we wish to simulate. This parameter establishes the limit where an execution will be stopped. One advantages of this parameter is that it makes to grow the execution time linearly as can be evidenced on the Fig. 4.

Fig. 4 shows the execution time averages of each, and we can observe a linear relation between number of defaults and execution time in both cases. We could infer the execution time for an input data, when we scale the number of defaults scenarios. This linear behavior can be evidenced in Fig. 4, where we have selected two large input, the first one is Germany (1113 banks) and the second is a mix between a set of banks from different EU countries (2727 banks). In both cases, the execution time grows considerably if we increase the number of default. For example, for the case of some EU countries with 1.000.000 defaults the execution time is around 4 days and 5 hours. Hence, the execution time begins to be non-viable, considering that simulations of the 28 countries of the EU should be executed frequently. These simulations justify the need for parallelizing and optimizing the SYMBOL code.

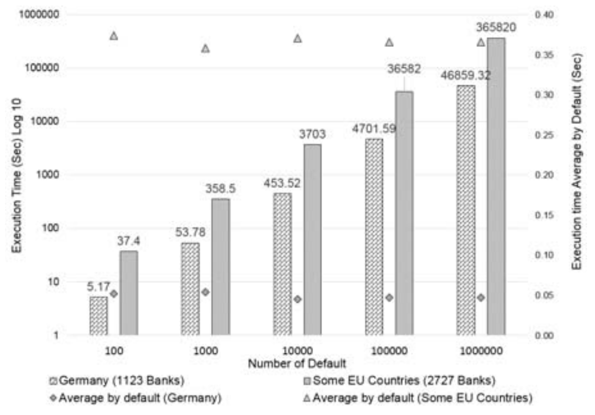

Fig. 4 Effects of scaling the default parameters on SYMBOL

Up to now, we have analyzed the input parameters. Next step is to analyze the execution on the multi-core architecture. In this sense, we have to clearly define the benefits of the multi-core architecture in order to improve the execution time keeping in mind: executing with larger problem sizes and scaling the number or default scenarios. However, these goals have to be achieved obtaining shorter response time (dropping the execution time), but also using in an efficient manner the computational resources. Hence, we have analyzed the SYMBOL model in order to identify which part of the code presents dependencies and which part of the code can be executed in parallel. It is important to understand that these dependencies create synchronization problems that affect seriously the execution time in a parallel version.

Under this focus, we have divided the code on five main parts, which represent more than 96% of the execution on SYMBOL. These parts are: initialization of the problem (where are define all the variables and are created all the elements need to start the execution), a random generator, Monte Carlo simulation module, Regulatory capital (Regcap) analysis function (all these three part are repeated depending on the number of defaults), and the last part is the contagion analyzing (this is performed when a default scenario is found). Then, to evaluate each part of the code we have executed using different data input (small, medium and large) in order to know the real behavior of SYMBOL. The results obtained are summarized on Fig. 5, where we can observe how each part can vary on time depending on the data input.

If we observe the Fig. 5, we can detail the impact of each function on the execution. For example, the Regcap function on the small input as Lithuania has a bigger impact than other functions with almost 54% of the execution, but for a big input, as Italy, the Monte Carlo simulation represents near 93%. On the contrary, when we compare the results of Poland and Bulgaria, we see that both RegCap functions and the Monte Carlo simulation have similar impact over the execution time. Hence, the main conclusion we can derive from these results is that the RegCap function has a big impact on small input, while the Monte Carlo function has a huge impact on large input data. Both parts need to be addressed in the optimization process.

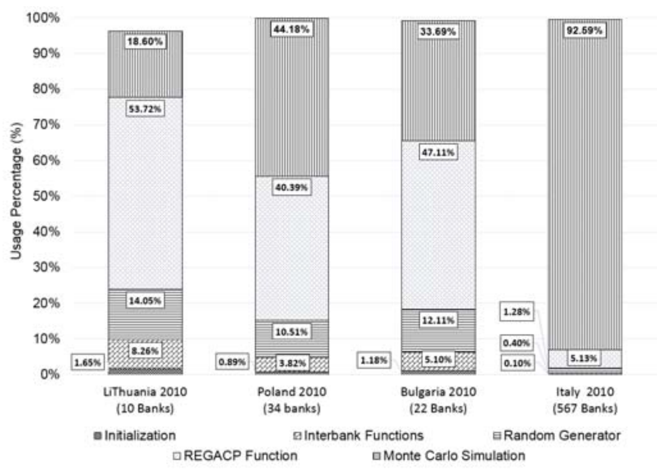

Fig. 5 SYMBOL function analysis execution time

Another analysis that we have to consider is related to the multi-core architecture that we will use to execute SYMBOL. In particular, one needs to take into account different architectures, for example, dual core, quad core, double quad core, etc. (Fig. 6), as well as the way they use different cache levels as L2 or L3, the use hyperthreading technology or hypertransport implementation to communicate between cores, different RAM memory size, etc. All these elements can be used to increment the speed on the execution, but, on the other hand they also raise issues, if they are not managed correctly. For this reason, we will optimize SYMBOL tool considering the main multi-core architectural aspect in order to obtain the best profit on the executions.

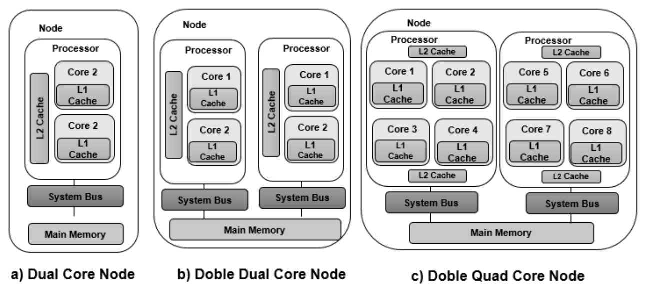

Fig. 6 Different Multicore Architecture

Once all parameters, architecture and the code, we start to design and apply the optimization techniques as will be explained below.

B. *Designing and Implementing the Optimization Process*

Here, we create and develop the optimization techniques that will improve the execution of SYMBOL. This optimization will allow setting up the best running configuration in order to take the most advantages of the multi-core architecture. However, it is important to mark that parallel execution is not always the best solution, because of the overhead added by the parallel libraries implementation such as OpenMP, MPIch, OpenMPI, OpenMP-MPI, etc. All these parallel tools (or strategies) add overhead that have to be manage carefully. In some cases, the parallelization will lead to worse execution time when we have small workload and if the code has a lot of communication exchanges, and/or there are huge amount of data synchronization inside the algorithm. Hence, we have decided to divide SYMBOL into two versions: an optimized serial (used for small input) and a parallel version (for large and medium input).

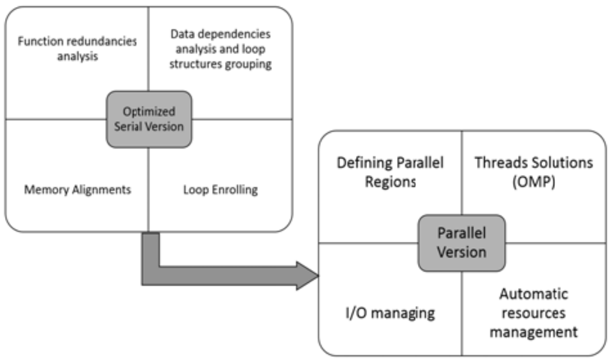
Function redundancies analysis
Data dependencies analysis and loop structures grouping
Optimized Serial Version
Memory Alignments
Loop Enrolling
Defining Parallel Regions
Threads Solutions (OMP)
Parallel Version
I/O managing
Automatic resources management

Fig. 7 SYMBOL Optimization Procedure

In the new serial version, we mainly remove all the function's redundancies calls and we group the loop structure in order to delete a set of repeated instructions. This grouping can be performed considering data dependencies on the SYMBOL model. Out of this serial version, we have developed a parallel version, which include a set of parallel strategies in order to optimize the execution time. Summarizing, the serial version is mainly focused on deleting the loop and function calls redundancies, managing the data structures using memory alignments, and applying the loop unrolling technique (Fig. 7) for deleting the computation redundancy. Then, the parallel approach of SYMBOL moves from this new serial version and it uses OpenMP[16], which is a well-known parallel library.

C. Optimized Serial Version

To create a new optimized serial version, we have to analyze the redundancies inside of the code. In this sense, we have evaluated both the analytical model of SYMBOL and the C code in order to determine which instructions can be removed or be grouped for deleting computation redundancies. An example is shown in Fig. 8, which represents the code of the Regulatory capital calculation (it calculates the banks losses for banks *i* as was estimated in (1).This function is called across all iterations for each bank. This means that it is executed thousands or millions times depending on the number of defaults that users wish to simulate.

As we observed before in Fig. 5, this function has big impact on small and medium input. Then, our proposal is to make an analysis of the code with the objective of determining points with and without dependencies with the aim of splitting them. Then, next step is to separate the part dealing with the Regcap equation, which is evaluated for each iteration and the elements that can be calculated as constants values. Hence, a new algorithm is proposed (see Fig. 9), where all elements without any crucial dependency are calculated first.

```
double regcap(double x, double y, double k) {
    double a, ex1, ex2, TheR;
    ex1 = 1-exp(-50*x);
    ex2 = 1/(1-exp(-50));
    TheR = 0.12*ex1*ex2+0.24*((1-ex1)*ex2);
              a = (k*gsl_cdf_ugaussian_P(
                          pow(1-TheR, -0.5)*
                              gsl_cdf_ugaussian_Pinv(x
                              ) + pow(TheR/(1-TheR),
                              0.5)*y) - x*k)* pow
                              (1-1.5*ba(x), -1)*1.06;
    return(a);
        }
/* Main SYMBOL Code  */
While(nDefaults < nTotDefaults) //
{
  /* 1) Random Generator Function and Monte Carlo
       simulation*/
        for (i=0; i<nbanks; i++)
  {  Baseloss = regcap(data[i][0], market[b], LGD);}
        /* 3) Evaluating of a Default scenario */
        /* 4) Contagion Part of the Algorithm */
        if (hadDefault) //if there is a default
           nDefaults++;
}
```

Fig. 8 Regcap Function in C Algorithm in C

The new code integrates 4 global variables (A, B, C, and D), which are the decomposition of the initial Regcap equation, and they are calculated for all banks in the simulation (line 9, Fig. 9). In other words, we have deleted the repetitiveness of computation, which was a weakness in the original code. These values are calculated in the lines 6 to 16 before of the loop structure where the core of SYMBOL is run. This optimization decreases considerably the execution time, but it also increases the memory allocation, due to the data of A, B, C and D have to be maintained during all the execution. However, for the most demanding test performed (2773 banks), the amount of memory was 22 KB for each variable, which is not a relevant value considering that current computers integrate more than 4GB. This example shows how we can remove the redundancies on the code.

The next optimization is related to the data structure. SYMBOL was originally created using the GSL library due to it use two functions of this library inside of the code. One example used is "gsl_linalg_cholesky_decomp", which calculates the Cholesky decomposition applied to the correlation matrix used in the Monte Carlo simulation. However, the overhead added by the GLS data structure is considerable, if we are not using many math operations to balance this overhead, the use of GSL library can worsen the execution time. Hence, it is better to avoid defining GSL data structures, if are not really needed. The main reason is that this kind of matrix structure integrates a set of data definition inside of the structure specifically six elements for each data [17] These definitions increase the memory allocation needs, which is an important key, when data on the code are aligned to take advantage of L1 and L2 cache memory.

For this reason, we have decided to modify the data structure using the memory alignment and pointer arithmetic. These allocations allow us to improve the spatial locality [18]. In addition, using this technique, we can reduce the cache misses and improve the execution time considerably. Also, these modifications do not only allow us to improve the computational speed, but also, it creates the possibility to apply parallel strategies as it will be detailed in next subsection.

The last optimization is about the loop unrolling. This technique is a well-known code transformation for improving the application performance and it has been applied in the most time consuming function Monte Carlo simulation (Fig. 5). It can improve from 10% to 30% the execution time, depending on the impact of the loop inside the code [19], because the branch instructions are reduced and the index variable is modified fewer times. Also, this method exploits greater instruction-level parallelism.

All these changes give rise to a new version of SYMBOL, with the main objective being minimizing the execution time. For this reason, once we finished all the modifications, we have evaluated a set of different data input from small to big. This evaluation is detailed in Fig. 10, where we can observe the positive effect of the optimization process. Here, we have a considerable improvement, up to 60% on the best case (Lithuania) where the execution time goes from 23 to 8 seconds.

However, our main objective is to reduce considerably for the medium and big test cases. We can observe that for Germany we get a 51% of improvement (execution time goes from 1 hour and 18 minutes to less than 38 minutes). For the other medium input cases, the improvements are around 56% (Fig. 10). It is important to note that this new version is not only faster, bur even more important, it has been developed thinking to a parallel version.

D.Parallel Version of SYMBOL

Once we have finished the serial version, we start the analysis of the parallel version. Hence, we analyze the parallel standards to be applied to SYMBOL. In other words, we have exploited the possibility of developing a version which allows us to execute SYMBOL on a multi-core cluster through message passing interface (MPI) libraries [20], or in a hybrid environment using multi-core and graphics processing unit (GPU). However, it is worthwhile to notice that, SYMBOL presents a lot of data dependencies, and they increase when we are executing the model with the contagion flag on. It means that we face a module by module execution and each module needs to collect the information in order to calculate the next step. At the end, if we use MPI we need to send a huge amount of messages by iteration, which can degrade execution performance due to synchronization problems.

```
double regcap2(int i, double y)
{       return((LGD*gsl_cdf_ugaussian_P(*(A+i)+*(B+i)* y)
     -*(C+i))*(*(D+i)));}

/* Main SYMBOL Code  */
/*Optimization of the Regcap function, deleting the
     redundancies */
     A= (double *) malloc (sizeof(double)*nbanks);
                  /*the same allocation must be done for B,C
                       and D */
     double ex2= 1/(1-exp(-50));
     for(i=0;i< nbanks; i++)
     {    ex1=1-exp(-50*data[i][0]);
          TheR=0.12*ex1*ex2+0.24*((1-ex1)*ex2);
          ba=pow(0.11852-0.05478*log(data[i][0]), 2);
          *(A+i)= pow(1-TheR, -0.5)*gsl_cdf_ugaussian_Pinv(
               data[i][0]);
          *(B+i)=pow(TheR/(1-TheR), 0.5);
          *(C+i)=data[i][0]*LGD;
          *(D+i)=pow(1-1.5*ba, -1)*1.06;
                  }
While(nDefaults < nTotDefaults) {
        /* 1) Random Generator Function */
          for (b=0; b<nbanks; b++)
                  Baseloss = regcap2(b, market[b]);
          /* 3) Evaluating of a Default scenario */
     /* 4) Contagion Part of the Algorithm */
                  if (hadDefault) //if there is a default
            nDefaults++;
}
```

Fig. 9 Recode Regcap Function, Optimized Version of SYMBOL

For this reason, we look for a solution, which allow us to make a parallelization inside the machine with the aim of managing the overhead added by the parallel library. In particular we can use OpenMP library. This is a well-known set of parallel libraries becoming more and more popular, since the number of cores inside the computer is increasing. Currently, we have systems where the number of cores can be 4, 6, 8, etc., per processor and with multiple processors in each machine. This large number of cores would allow us to create a small high-speed environment.

OpenMP is a portable, scalable tool, which allows programmers, using a simple interface, to create parallel region where the code is executed in parallel on multiple threads sharing data. However, we have to take care of data sharing between threads and we have to avoid the dependencies inside of the loop structures in order not to create deadlock scenarios or situations where the overhead can get worse the execution time.

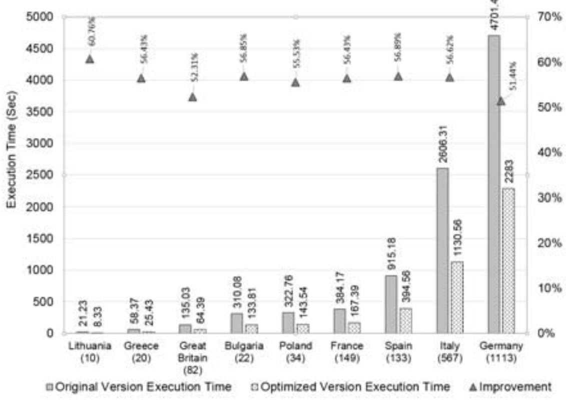

Fig. 10 Serial Execution using both original and optimized version of SYMBOL (100.000) defaults

In some cases, the parallel execution can take more time than the serial version, mainly because of synchronization problems between the parallel processes (or threads) such as: communication imbalances, congestion in the communication paths (memory or network), overheads added by the parallelism, few amount of computed data against the communications performed, etc. This can be evidenced on Fig. 11, where execution performances of the small inputs are seriously affected by the parallelization. Execution time can be double when we are executing with two or more threads.

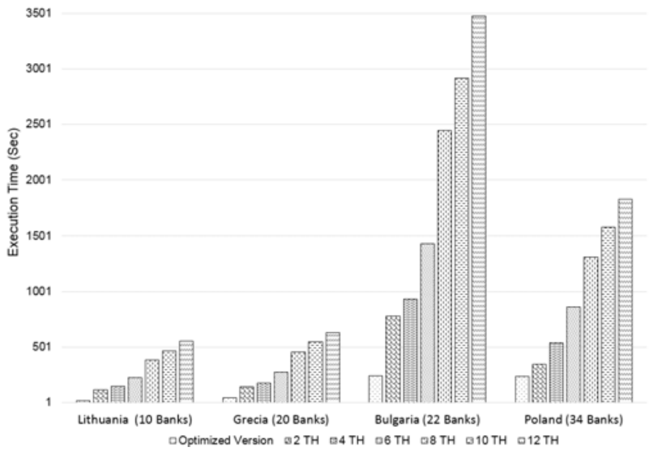

Fig. 11 Worst Parallel results using SYMBOL and OpenMP

However, we have selected a medium and big input to test the improvement achieved using the parallelization. Using a correct input data, the parallel version needs shorter execution time, but results depend on the number of threads open for execution. As an example, we look at the case of Great Britain (Fig. 12). In this case when using two threads, the parallel version is faster than serial version. Hence, if we increase the number of threads (e.g. 4,6,8,10, ...) the performance will not improve because the overhead generated by threads is bigger than computation time of the workload. A similar behavior has been obtained for Spain, where the execution time improve until four threads (Fig. 12). These results show that as the number of banks increases, execution using threads starts to be an ideal solution.

Also, Fig. 12 shows two additional examples where the OpenMP version is exploiting the parallelism (Italy and Germany examples). In both cases we have opened until 12 threads and with these setting, we have the fastest execution time. For example in the case of Germany, the execution time has dropped off from 38 minutes of the optimized version, to less than 5 minutes using the OpenMP version.

This improvement represents gains around to 87% of the time. However, it is important to understand that the number of threads will be limited by two main factors: number of physical cores on the machine and the computational workload of the input data[5].

[5] For these experiments, we have used a machine with two Intel processor Xeon X5670 with 6 core each processor.

E. Automatic Tuning of SYMBOL on Multicore Environments

To address the problem of the threads discussed in the previous Section, we need to create an analysis to determine the ideal number of threads according to the input data in a transparent manner. Hence, we have created an automatic procedure which set the ideal number of threads to be opened for the parallel version or it simply selects the optimized serial version. To define the ideal number of banks by thread that we have to open, we have evaluated the overhead created by the OpenMP implementation on each iteration on SYMBOL.

An example of this characterization is detailed in Fig. 13, where we have selected the worse cases (i.e. Great Britain and Spain) or small inputs. We have divided the code in two main parts, the code that use the OMP primitives and the rest of the program (Fig. 13). The serial execution gives better performances than the Omp Parallel using 2 threads for small input (Lithuania, Greece and Bulgaria), mainly because of the overhead added by the OpenMP library. For example, if we observe the average on the execution time by thread and iteration, they are around 8.0 E-6 and 9 E-6 second for all these small inputs, considering that the number of total iteration were 4.692.777, 5.874.684 and 2.787.7592 respectively for 100.000 defaults. However, if we analyze the average execution time for the serial version, their values are lower than 3E-6. Then, we can conclude that the minimum overhead by iteration is around 9.0E-6 and 1.0E-5 second by iteration. In this case, all executions with a lower average by thread than this threshold overhead has to use the serial optimized version.

On the other hand, we have analyzed the medium input to observe the impact of the thread overhead. In this case, we have selected two cases Great Britain with 82 banks and Spain with 133 banks. The results of the characterization can be found on Fig. 14, and we can observe that the OpenMP version is getting better results that the serial version

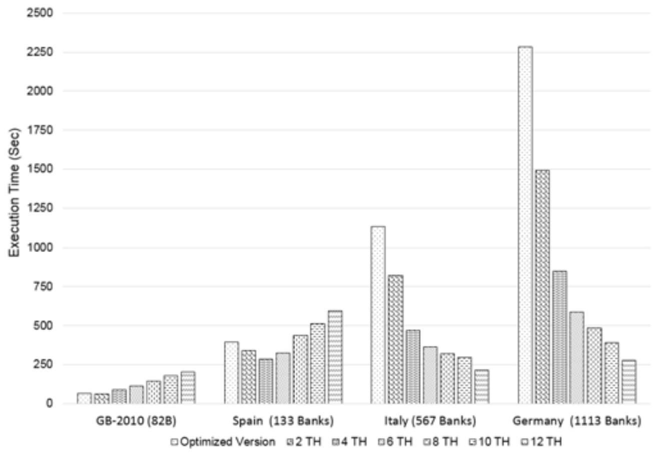

Fig. 12 Improving Parallel execution of SYMBOL using OpenMP

However, these improvements are achieved until a specific point. For example in Great Britain, the execution with two threads is better than serial but when we execute with four threads the execution begins to be worse. A similar behavior occurs with Spain, where the execution time is better when we execute with two or four threads but it start to get worse when the number of thread is equal to six.

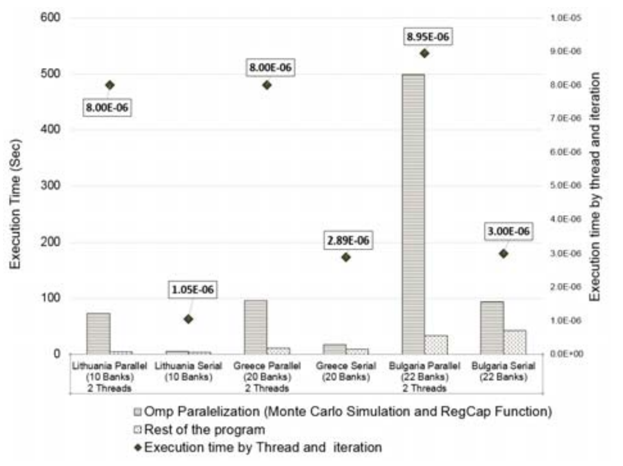

Fig. 13 Serial and OpenMP Thread characterization for small inputs

In particular, if we observe the execution time average for the worse cases (4 threads for Great Britain and 6 threads for Spain), we can determine that the overhead of the parallel implementation is affecting the performance. In both cases the values of the execution time average by threads are lower than 10E-5. This means that they are below of the threshold of the minimum added overhead of the OpenMP. Then, if we consider the execution time of Spain with 4 threads open, we can observe that the execution average is around 1.3E-5 that is higher than the minimum overhead and its more or less 33 banks by thread. For this reason, we have defined our threshold as 30 banks in order to define an approximate value that gives us the minimum requirement in order to cover the overhead added by the OpenMp solution. However, the maximum number of threads is limited by the physical capacity of the multi-core architecture. This value has been tested with a set of different input as can be evidenced on the performance evaluation.

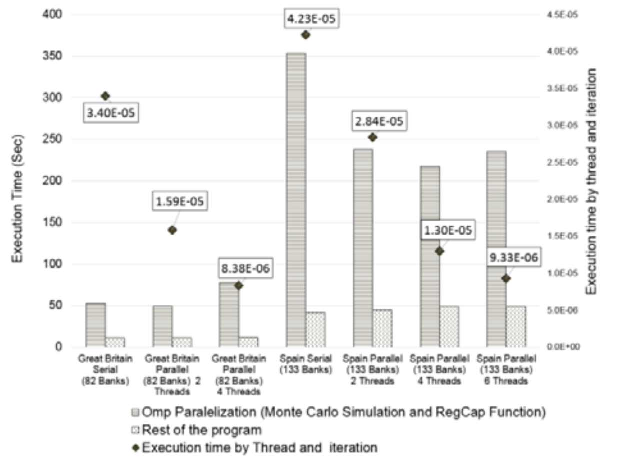

Fig. 14 Thread characterization for medium inputs

Once defined this value we can create an automatic tool of SYMBOL. This tool is compiled using pre-processor macros, where, by using conditional compilation, we can control which part of the code will be executed in serial or parallel. Also, in the case of executing in parallel, we have to define the maximum number of threads that the machine can support using the analysis explained before of the number of threads.

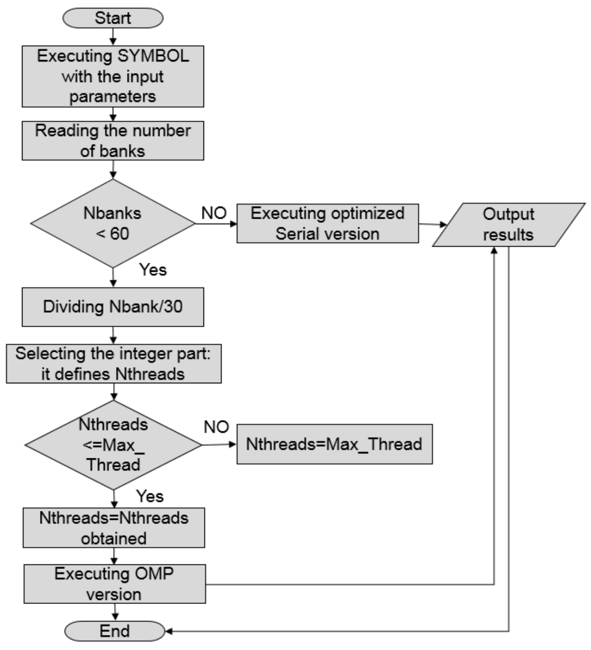
Start
Executing SYMBOL with the input parameters
Reading the number of banks
Nbanks < 60
NO
Executing optimized Serial version
Output results
Yes
Dividing Nbank/30
Selecting the integer part: it defines Nthreads
Nthreads <=Max_ Thread
NO
Nthreads=Max_Thread
Yes
Nthreads=Nthreads obtained
Executing OMP version
End

Fig. 15 Thread characterization for medium inputs

The parallel and serial version are called using a batch file, which would determine the version (serial or parallel) and the amount of threads that has to be used considering the input.

Summarizing SYMBOL is executed automatically and it follows the flowchart defined on Fig. 15, where an input needs at least 60 banks in order to open at least two OMP threads on the parallel version. Also, if the number of threads obtained is bigger than the number of cores of the machine, then SYMBOL uses as a number of threads the maximum capacity of the system. For example, Germany has 1113 banks divided by 30 is equal to 37 threads, but this number cannot be bigger that the physical support of the machine. For this reason, the number of threads defined will be the maximum established by the multi-core architecture.

F. Testing and Data Verification of SYMBOL

All versions developed have been tested with a strict procedure, where we have compared the results of original version using a set of different input data. Then, we evaluated all results obtained through the different versions with the aim of determining the effectiveness of each version. The results obtained show an accuracy of 100% between original version output data and the output of the optimized version of SYMBOL.

IV. Performance Evaluation of SYMBOL

To test the new optimization of SYMBOL, we have used a MAC machine composed by two processor of 2.93 Ghz 6 Core Intel Xeon, 64 GB 1333 Mhz DDR3 memory RAM, Mac OS X lion 10.7.5 operative system, and gcc 4.6.2 compiler. Furthermore, the inputs used have been classified in small, medium and large according to the number of banks. This evaluation tries to demonstrate the improvement of the two main issues of SYMBOL: execution time and scalability of the default scenarios. In this sense, Fig. 16 shows the execution time obtained for 100.000 defaults using the new automatic SYMBOL tool.

We recall that the number of threads to be opened, according to the number of banks, is automatically chosen by our tool, and we can take the most out of the multi-core machine. For example, in the Great Britain example, the number of threads opened is two, while in Germany the number of threads is 12 (see Fig. 16). For the cases of small input as a Lithuania and Poland are used the serial version and their improvements are around 60% and 56% respectively. Also, analyzing the performance results in Fig. 16, we can see the considerable improvements we have comparing the original SYMBOL and the new optimized SYMBOL tool. These improvements range from 52% to 94%. In this case, we can observe how our tool can open the right number of cores in order to improve the execution time.

Another analysis is related to the scalability of SYMBOL when we increase the default scenarios. In this sense, we have observed that SYMBOL can request a lot of time to execute big simulations (Fig. 4). In this case, we increased the number of default scenarios from ten thousand to one million and the results are summarized in Fig. 17, where considerable improvement is achieved by the new SYMBOL parallel version when it is run with a large number of banks and with large number of default scenarios. These tests were done using a mix input of different EU countries with 2727 banks.

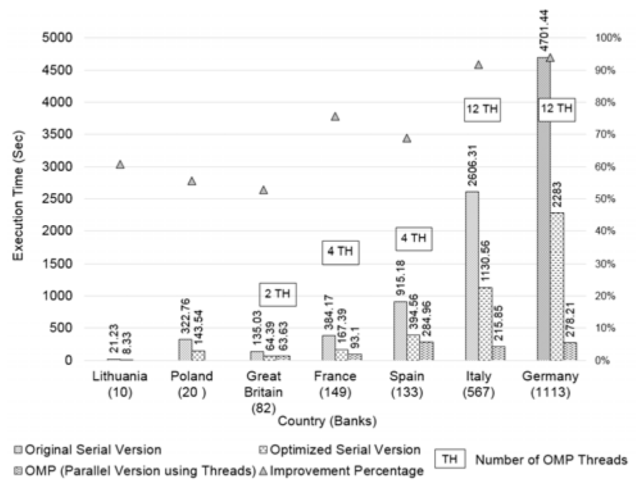

Fig. 16 SYMBOL execution using the automatic tool

Results show a huge reduction on the execution time (96% in the best case), which leads to have simulation running for few hours instead of several days as is presented in Fig. 17.

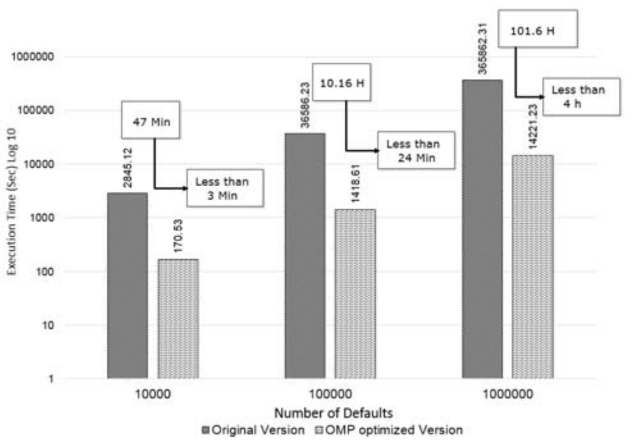

Fig. 17 SYMBOL Execution using the automatic tool and increasing the problem size

The last evaluation of the tool is the performance analysis on speed and efficiency, between the new optimized serial version and the OMP parallel. Overall performance evaluation are shown in Table I, where two performance metrics have been analyzed: speedup and computational efficiency. The speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. In this sense, we calculate it using the serial time divided by the parallel time.

As we can detail on the Table I, the efficiency begin to be around 100% using all the architecture and a big data input (Some EU Countries). This is due to the overhead of the OMP library start to be irrelevant on the execution time. However, For Spain and France the comparison between the optimized serial and the parallel version using 4 threads (see Table I), will give: speedups around 1.44 and 1.85 and the efficiency are 36% and 46.35%, respectively. In both cases, the impact of the added overhead affects the linearity grow of the speedup.

TABLE I
Performance Evaluation of SYMBOL Tool

Data Input	N Threads Opened	Speedup Optim /Parallel	Efficiency Optim /Parallel
Spain (133)	4	1.44	36.00%
France (149)	4	1.85	46.35%
Italy (567)	12	5.57	46.40%
Germany (1113)	12	9.42	78.51%
Some EU Countries (2727)	12	11.93	99.34%

V. Conclusions

In this paper, we have presented a case-study where SYMBOL model has been adapted automatically and transparently to the multi-core architecture. To make this, we have created a method, where we have analyzed the analytical model and its characteristics, and then we have proposed a set of changes that have been performed on SYMBOL tool in order to improve its overall performance. This method integrates five main steps, starting to an analysis of the SYMBOL and finishing with the verification of the output results. All these steps allow us to improve and to execute SYMBOL exploiting the multi-core architecture.

In addition, the new SYMBOL tool can automatically set up its working framework to take the most advantages from the multi-core architecture. In particular, it detects which version to use (either serial or parallel) using the input information, and then it defines the number of threads needed to get a faster execution. The new approach of SYMBOL has been tested using a set of different input data and the results show the effectiveness of the improvements, where for some cases can achieve a reduction around 96% on the execution time using the parallel version and around 55% for the optimized serial.

Finally, we have evaluated the effects on the new SYMBOL when we increase the number of default scenarios. In this case, we can scale both input data and number of defaults and the results show a considerable improvement around 96% on the execution time for the best case tested. Now it is viable to use this new SYMBOL version because we can analyze the consequences of bank failures and its effects over a larger amount of banks in the European Union banking sector as was demonstrated in our experimental validation.

REFERENCES

[1] Michailidis, P., Margaritis, K. .Efficient Multi-Core Computations in Computational Statistics and Econometrics, IEEE 15th Int.Conference on Computational Science and Engineering (CSE), pp.267274.

[2] De Lisa R., Zedda S., Vallascas F., Campolongo F., Marchesi M., 2011,Modelling Deposit Insurance Scheme losses in a Basel 2 framework, Journal of Financial Services Research, Volume: 40 Issue: 3 pp.123-141

[3] Vasicek O. A., 2002, Loan portfolio value, Risk http://www.risk.net/data/Pay per view/risk/technical/2002/1202 loan.pdf

[4] Merton R.C., 1974, On the pricing of corporate debt: the risk structureof interest rates, Journal of Finance, 29, 449-470

[5] Basel Committee on Banking Supervision, 2005, An Explanatory Noteon the Basel II IRB Risk Weight Functions http://www.bis.org/bcbs/irbriskweight.pdf

[6] Basel Committee on Banking Supervision, 2006, International Convergence of Capital Measurement and Capital Standards http://www.bis.org/publ/bcbs128.pdf

[7] Basel Committee on Banking Supervision, 2010 rev 2011, A global regulatory framework for more resilient banks and banking systems http://www.bis.org/publ/bcbs189.pdf

[8] Sironi A., Zazzara C., 2004, Applying Credit Risk Models to Deposit Insurance Pricing: Empirical Evidence from the Italian Banking System, Journal of International Banking Regulation, 6(1)

[9] James C., 1991, The Loss Realized in Bank Failures, Journal of Finance,46, 1223-42

[10] Mistrulli P.E., 2007, Assessing Financial Contagion in the Interbank Market: Maximum Entropy versus Observed Interbank Lending Patterns, Bank of Italy Working Papers n. 641

[11] Upper C., Worms A., 2004, Estimating Bilateral Exposures in the German Interbank Market: Is there Danger of Contagion?, European Economic Review, 8, 827-849

[12] Zedda S., Cannas G., Galliani C., De Lisa R., 2012, The role of contagion in financial crises: an uncertainty test on interbank patterns, EUR Report 25287, ISSN 1831-9424, ISBN 978-92-79-23849-9 http://publications.jrc.ec.europa.eu/repository/bitstream/111111111/25695/1/lbna25287enn.pdf

[13] European Commission, Directorate-General for Economic and Financial Affairs, 2011, Public finances in EMU 2011, European Economy 3 2011 http://ec.europa.eu/economyfinance/publications/european economy/2011/pdf/ee-2011-3 en.pdf

[14] European Commission, Directorate-General for Economic and Financial Affairs, 2012, Fiscal Sustainability Report, European Economy 8—2012http://ec.europa.eu/economyfinance/publications/european economy/2012/pdf/ee-2012-8 en.pdf

[15] De Rose C., Fernandes P., Lima A, Sales A. and Webber, 2011, Exploiting Multi-core Architectures in Clusters for Enhancing the Performance of the Parallel Bootstrap Simulation Algorithm, IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), pp 1442-1451

[16] OpenMP Architecture Review Board, 2013, OpenMP Application Program Interface

[17] Galassi M, Davies J, Theiler J, Brian G, Jungman G., Alken P., Booth M., Rossi F., 2013, GNU Scientic Library Reference Manual, http://www.gnu.org/software/gsl/manual/gsl-ref.pdf

[18] Faria Nuno, Silva Rui and Sobral Joao, 2013, Impact of Data Structure Layout on Performance, 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, pp. 117-120,Ireland

[19] Davidson, Jack W., Jinturkar, Sanjay, 2001, An Aggressive Approach to Loop Unrolling, Technical Report, University of Virginia, USA

[20] Message Passing Interface Forum, 2012, MPI: A Message-Passing Interface Standard Version 3.0 Technical report, 2012

Ronal Muresano was born in Barquisimeto Venezuela, in 1979. He received the Bachelor degree in Computer Engineering from the University Valle del Momboy, Valera, Trujillo, Venezuela in 2002, and the Master and PHD in high performance computing by University Autonoma of Barcelona (UAB), Spain, in 2008 and 2011 respectively.

In 2002, he has joined to the Computer Engineering Department as a professor at the University Valle del Momboy until 2007. Since 2007 until 2013, he was part of the Computer Architecture and Operative Systems (CAOS) department as a research staff and assistant professor at the University Autonoma of Barcelona (UAB), Spain, where his main interest was on software optimization on multicore architecture and parallel computing. His is currently working in the European Commission (EU), at Joint Research Center (JRC) in the Scientific Support to Financial Analysis (SFA) unit as a research staff.

Andrea Pagano was born in Rome in 1964. He has a Ph.D. in mathematics from Bown Universisty.

After few years in academia working on complex geometry, he started to work on environmental modeling at ENEA. In 2005 he joined the Joint Research Center (JRC) of the European Commission (EC), where he worked on Global Sensitivity Analysis. In 2011 he joined the group of Economic and Financial Analyses where he works as data analyst and modeler.