Large Scale Distribution of Stochastic Control Algorithms for Gas Storage Valuation

Constantinos Makassikis, Stéphane Vialle
SUPELEC, IMS research group, 2 rue Edouard Belin, 57070 Metz, France
LORIA, ALGORILLE project team, BP 239, 54506 Vandoeuvre-lès-Nancy, France
Constantinos.Makassikis@supelec.fr, *Stephane.Vialle@supelec.fr*

Xavier Warin
EDF - R&D, OSIRIS group, 1 Avenue Charles de Gaulle, 92141 Clamart, France
Xavier.Warin@edf.fr

Abstract

This paper introduces the distribution of a stochastic control algorithm which is applied to gas storage valuation, and presents its experimental performances on two PC clusters and an IBM Blue Gene/L supercomputer. This research is part of a French national project which gathers people from the academic world (computer scientists, mathematicians, ...) as well as people from the industry of energy and finance in order to provide concrete answers on the use of computational clusters, grids and supercomputers applied to problems of financial mathematics.

The designed distribution allows to run gas storage valuation models which require considerable amounts of computational power and memory space while achieving both speedup and size-up: it has been successfully implemented and experimented on PC clusters (up to 144 processors) and on a Blue Gene supercomputer (up to 1024 processors). Finally, our distributed algorithm allows to use more computing resources in order to maintain constant the execution time while increasing the calculation accuracy.

1 Introduction and objectives

1.1 Project overview

Gas prices exhibit fluctuations which are mainly due to the modification of demand. Because of the inelasticity of production and demand, prices are, for example, higher during winter than in summer. A gas storage facility allows its owner to take advantage of the price dynamic to do some arbitrage between periods where prices are high and periods where prices are low. Recently, a lot of research has been achieved in the field of gas storage valuation (see [2, 3] for example). As a result, many different price models can be used to carry out this valorization. However, these models are usually CPU and memory-consuming, and need to be run in limited time. So, the design and implementation of parallel algorithms are recommended to use these models on distributed architectures in industrial environments. Specific parallelizations have been designed and experimented in this project to run financial applications requiring both large amount of CPU and memory in limited time. Even if large scale distribution is not mandatory for this problem (small clusters should be enough) we have extensively tested this method in order to prepare for the parallelization of huge multi-stock stochastic problems used at EDF company to globally optimize its electricity production assets.

The following subsections introduce the project objectives and challenges. Then, section 2 presents the mathematical problem and the sequential algorithm. The design of our optimized distributed algorithm is described in section 3. Section 4 describes the experimental testbeds used to evaluate our distributed algorithm. Performance measures are analyzed in section 5, while section 6 focuses on a scalability experiment of a huge benchmark. Finally, section 7 summarizes the results of this research and introduces some future investigation ways.

1.2 Financial computing objectives

In our study, we use three different models which are based on the dynamic of the forward curve that is given by the gas market prices for a future delivery of energy:

- the first one is a one-factor model based on an Ornstein-Uhlenbeck process described in [6] and com-

monly used for energy and general commodity:

$$\frac{dF(t,T)}{F(t,T)} = \sigma_S(t)e^{-a_S(T-t)}dz_t^S$$

where z_t^S is a brownian motion on a probability space (Ω, F, P) endowed with a filtration $\{\mathcal{F}_t, t \in [0,T]\}$, σ_S is the short-term volatility, a_S is a mean-reverting term.

- the second one is based on a two-factor Ornstein-Uhlenbeck process, hence a two-factor model designed to catch the medium-long term behaviour of the forward curve:

$$\frac{dF(t,T)}{F(t,T)} = \sigma_S(t)e^{-a_S(T-t)}dz_t^S + \sigma_L(t)e^{-a_L(T-t)}dz_t^L$$

where z_t^L, $\sigma_L(t)$, and a_L are characteristics of the long-term Ornstein-Uhlenbeck factor.

- the third one is a one-factor model similar to the first model except that the brownian motion used is replaced by a normalized Normal Inverse Gaussian process [1]:

$$F(t,T) = F(t_0,T)e^{M(t,T)+\int_0^t \sigma_S(u)e^{-a_S(T-u)}dL_u}$$

where L_u is the normalized Normal Inverse Gaussian process with parameters $\alpha, \beta, \delta, \mu = 0$. The relation $\frac{\delta\alpha^2}{\gamma^3} = 1$ is imposed to get the L_u variance equal to u.

$$M(t,T) = -\int_0^t \delta(\gamma - \sqrt{\alpha^2 - (\beta + \sigma_S e^{-a(T-s)})^2})ds$$

is designed so that the price process is a martingale. This price model exhibits spikes that the Gaussian model cannot reproduce.

All valuations are achieved by a stochastic programming approach described in [6] and in section 2. If the time needed to compute the solution with the first model is not too long (typically less than 10 minutes), the time needed by the other models makes them virtually unusable. Hence the need of parallelization.

1.3 Computer science challenges

From a computer science point of view, this is a *Stochastic Dynamic Programming algorithm* which is complex to parallelize. Despite some natural parallelism (see section 2), computations at any given step depend on previous results and the range of data to be computed changes regularly. As a result, computations and data need to be redistributed at each step. This requirement leads to compute and execute a complete routing plan at each step on each processor. Moreover, the data required by each processor for the next step needs to be finely identified, in order to route and store the minimal amount of data on each processor. This strategy is necessary to process large scale problems on large numbers of processors. A systematic broadcast and storage of all previous results on each processor would be easy to implement but would require too much memory on all processors.

A lot of research in financial computations focus on the parallelization of option pricing, considering independent computations [5] as well as computations requiring many communications between processors [4]. Gas storage valuation equations take into account *gas stock levels* and lead to large distributed computations composed of independent computing steps and complex inter-task communication steps.

2 Stochastic control application

2.1 Description of the problem

A gas storage facility presents three regimes: injecting gas, withdrawing gas, and just storing the gas. The gas storage is a cavity characterized by:

- its size given in giga British Thermal Units (BTU) or MWh (a standard conversion rate is used to convert BTU to MWh preferred by electric utility);
- the daily injection/withdrawal capacity a_{in}/a_{out} which depends on the stock level of the cavity I_t;
- the standard operating and managing cost per day which depends on the operating regime: $K_{in}(I_t)$, $K_s(I_t)$, $K_{out}(I_t)$.

The storage size can be variable in time because we may want for example to hire a portion of the cavity.

Most of the time, the gas storage manager uses its facility according to a *bang bang strategy*. In this case, the instantaneous gain (or cost) at a date t depends on the gas price S_t and the management regime. Here are the characteristics of the three regimes:

$$\begin{cases} \text{Injection} & a_{in,s}(I_t), \text{ with cost:} \\ & \phi_{-1}(S_t, I_t) = -S_t a_{in}(I_t) - K_{in}(I_t) \\ \text{Storage} & \text{with cost:} \\ & \phi_0(S_t, I_t) = -K_s(I_t) \\ \text{Withdrawal} & a_{out,e}(I_t), \text{ with gain:} \\ & \phi_1(S_t, I_t) = S_t a_{out}(I_t) - K_{out}(I_t) \end{cases}$$

The instantaneous evolution of the stock I_t depends on the regime of the facility:

$$\begin{cases} dI_t &= a_{in,s}(I_t)dt & \text{in injection} \\ dI_t &= 0 & \text{in storing} \\ dI_t &= -a_{out,e}(I_t)dt & \text{in withdrawal} \end{cases}$$

We suppose that a strategy u_t describing the regime taken at date t can take three values: 1 in withdrawal regime, 0 in storing regime, -1 in injection regime. We suppose that the gas storage is hired between t and T, and that the regime switching can occur at any date. At last, for simplicity, we take a zero interest rate. Then the gain obtained by managing the facility from a date t to a date T with a strategy u is given by:

$$J(t,s,c,i,u) = \mathbb{E}(\int_t^T \phi_{u_r}(r,S_r,I_r)dr + J(T,S_T,I_T,i_T,u_T)|S_t = s, I_t = c, u_t = i)$$

where:

- $J(T,S_T,I_T,i_T,u_T)$ is a given final value function, for example a penalization of the difference between I_T and a target final value I_T^{target};
- s is the gas price at time t given by a Markovian process;
- c is the stock level at time t;
- i is the regime at time t.

The goal of the manager is to find an optimal admissible adapted strategy in a given set $\mathcal{U}_t$, in order to maximize its income and therefore to solve:

$$J^*(t,s,c,i) = \sup_{u \in \mathcal{U}_t} J(t,s,c,i,u)$$

2.2 Algorithms

2.2.1 Stochastic control algorithm

In our models, the price of electricity is given by a Markovian process. We use stochastic dynamic programming in order to optimize the management of the facility. The stock is discretized in equally spaced levels. Furthermore, the regime switching occurs only at given dates (once a day): so we discretize time in equally sized intervals Δt. From the final value of J^*, we evaluate the value J^* for all the previous dates and all the levels of the stock with the algorithm of figure 1.

This algorithm is a generic one: the price model only appears in the conditional expectation. The time loop (t variable) is inherently sequential, unlike the stock level loop (c variable) which can be efficiently parallelized. However, some complex data exchange will be mandatory at the end of each time step (see section 3).

For $t := (N-1)\Delta t$ *to* 0
  For $c \in$ admissible stock levels
    For $s \in$ all possible price levels
      $\tilde{J^*}(s,c) := \max(-(a_{in}s + K_{in})\Delta t + \mathbb{E}(J^*(S_{t+\Delta t}, c + a_{in}\Delta t)|S_t = s),$
      $(a_{out}s - K_{out})\Delta t + \mathbb{E}(J^*(S_{t+\Delta t}, c - a_{out}\Delta t)|S_t = s),$
      $-K_s\Delta t + \mathbb{E}(J^*(S_{t+\Delta t}, c)|S_t = s))$
  $J^* := \tilde{J^*}$ *//Set J^* for the next time step*

Figure 1. Stochastic control algorithm.

2.2.2 Conditional expectation algorithm

The previous generic algorithm used for stochastic control uses the calculation of the conditional expected gain associated with price uncertainties. In order to evaluate this expectation, different techniques are used:

- A trinomial tree is used to generate uncertainty factors for the Ornstein-Uhlenbeck processes [6]. With a one-factor Gaussian model a single tree is generated, so the expectation is evaluated very quickly (will lead to our "G" algorithm). With a two-factor model two trees are combined generating far more calculation (that will lead to our "G-2f" algorithm). Furthermore, the long-term tree has far more branches than the short-term tree due to a small value of the long-term mean-reverting coefficient. In the second case the memory needed explodes with the maturity of the evaluation.
- A Partial Integro Differential Equation of that kind is used to calculate the expectation in the third model:

$$\frac{\partial f}{\partial t} - \int_{\mathcal{R}} \left(f(x+y) - f(x) - \frac{\partial f}{\partial x}(x)y \right) K_{NIG(\alpha,\beta,\delta)}(y)dy - (\sigma_S \frac{\delta\beta}{\gamma} - a_S)\frac{\partial f}{\partial x} = S(x)$$

  where the kernel $K_{NIG(\alpha,\beta,\delta)}(y)$ behaves as $O(1/y^2)$ in 0 (it will lead to our "NIG" algorithm). This calculation is not very memory-demanding but is far more costly than in the Gaussian model.

3 Optimized distributed algorithm

3.1 Distribution strategy

To achieve large speedup and size-up, we have decided to parallelize the stochastic control algorithm of figure 1 on scalable distributed architectures, such as PC clusters and distributed memory supercomputers. Temporal steps of the external loop have to be run sequentially, but computations of the second loop on stock levels can be run concurrently. So, we have split the stock level loop on a set of processors

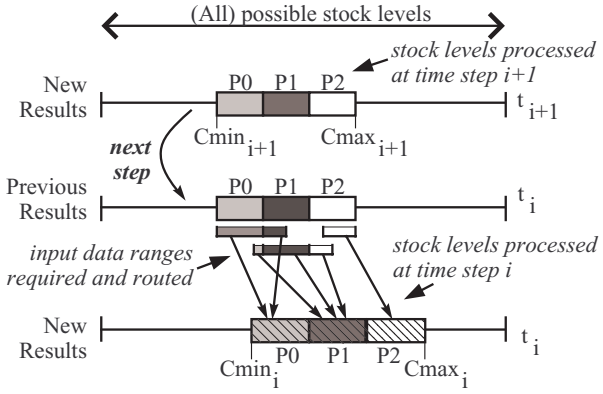

Figure 2. Example of optimized data distribution on three processors.

communicating by message passing. However, the range of stock levels to process changes at each temporal step, and leads to redistribute computations and data at each step.

As illustrated on figure 2, the stock level loop (from $Cmin_{i+1}$ to $Cmax_{i+1}$) is load balanced on processors at step $t = i + 1$, and each processor stores its results. At step $t = i$ (the next step), the new stock level loop (from $Cmin_i$ to $Cmax_i$) is load balanced on processors, and each processor requires a specific range of the previous results as input data. Hence, each processor computes the new input data range required to process each stock level, and deduces the input data range required by each processor to process its new range of stock level (according to the load balancing of the new range of stock levels to process). Then, each processor establishes its routing plan: it points out the range of its previous results to send to each other processor, and the range of previous results to receive from each other processor. Finally, each processor executes its routing plan according to a predetermined message passing scheme.

To store some new input data tables with different sizes at each step, some data tables are allocated and freed at each step on each processor. This memory management strategy introduces some small overhead, but in exchange minimizes the amount of memory used and therefore allows larger problems, requiring more processors, to be treated.

3.2 Main algorithm steps

According to the strategy introduced in the previous subsection, we have designed the distributed algorithm depicted in figure 3. The first step consists in load balancing the computations of the prices at t_n by calling specific initialization routines. Then the algorithm enters a loop of n steps (from t_{n-1} to t_0) which encompasses two main substeps: *data exchange planning and execution*, and *new computation processing*.

The *data exchange planning and execution* sub-step starts computing the new load balancing map and the new input data distribution map on each processor. These computations are simple, and are faster to compute entirely on each processor than to distribute and parallelize. Then each processor builds its routing plan and resizes its local data tables, according to the new input data distribution map. The data exchanges are achieved just after these preliminary operations, and are based on point-to-point communications.

The *new computation processing* sub-step is illustrated at the bottom of the time step loop of figure 3. It consists in a pure local and efficient computation on each processor, according to the stochastic control algorithm of figure 1, and to a fixed price model (see section 1.2).

3.3 MPI based implementations

Our first implementation was based on a Python top-level program calling MPI communication routines through the Python *Pypar* module [7]. This allowed users to easily tune the top-level program and run different distributed computations, but it was limited to the use of `MPI_Bsend()` routine and did not run on Blue Gene/L (which did not support Python), and we limited our experiments to a small 32 PC cluster. Our second implementation has been entirely achieved using MPI and C++ programming tools. Three different versions have been implemented: using the blocking communication routine `MPI_Bsend()`, and the non-blocking routines `MPI_Ibsend()` and `MPI_Issend()`. Non-blocking versions allow each PC to parallelize and overlap its message sending and receiving at each step, and the non-blocking `MPI_Issend()` routine achieves non-blocking handshakes and requires a more complex design but avoids to allocate extra communication buffers and to write out again data. As the required memory is less important, this implementation can reach greater size-up, when distributing large applications. Moreover, its non-blocking handshake has exhibited very limited overheads in our application experiments.

4 Testbed introduction

4.1 Experimental distributed systems

Our distributed algorithms and implementations were assessed on three different testbeds. The first was the "Pentium-4 cluster" of SUPELEC which interconnects 32 PCs across a cheap Gigabit Ethernet network composed of two interconnected 24-port switches. Each PC has a Pentium-4 at 3 GHz and 2 GB of RAM. The second was the "dual Opteron cluster" of the French experimental Grid *Grid'5000*. It is composed of 72 nodes with two single core Opteron processors at 2 GHz and 2 GB of RAM that

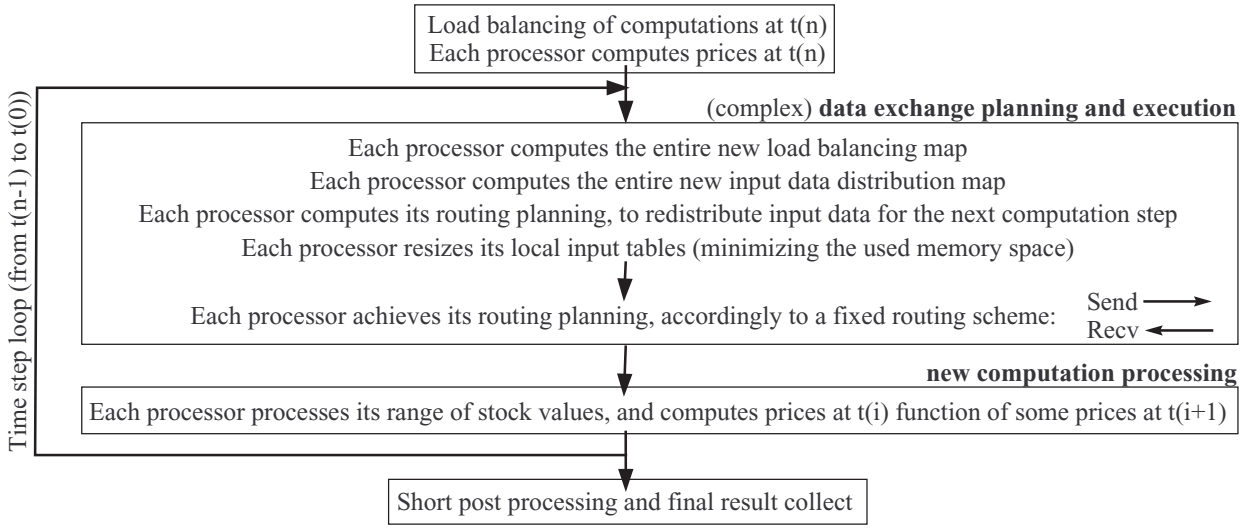

Figure 3. Main steps and sub-steps of our distributed algorithm for stochastic control.

are interconnected across a fast Gigabit Ethernet switch. The third was the "IBM Blue Gene/L supercomputer" of EDF R&D, providing up to 4096 nodes which communicate through proprietary high-speed networks. Each node hosts two processors at 700 MHz which share 1 GB of RAM. These testbeds are various but all have mono-core processors.

4.2 Test application features

For the purpose of testing the application with our different models we consider the following scenario where a gas storage owner wants to valuate his utility which has a capacity of $100,000$ MWh for a use during two years. The injection and withdrawal rates have values ranging between 100 and $1,000$ MWh per day and are highly dependent on the stock level. When the stock level in the cavity is high, the pressure is high too and makes injections more difficult than withdrawals. Conversely, when the stock level is low, injections are easier than withdrawals. The storage is valuated for a use beginning in one year and finishing two years later. The initial stock level is $20,000$ MWh and the final value of the gas storage in three years is set to 0 for simplicity. An annual interest rate of 8% is used as well as the forward prices available at Zeebruge hub at the beginning of 2006. The discretization step of the gas storage is set to 500 MWh.

The three stochastic price models are characterized by a daily short-term volatility equal to 0.014 associated to a daily short-term mean-reverting value of 0.0022 that totally define the first Gaussian model. The two-factor model needs two additional parameters to be defined: the daily long-term volatility set to 0.004, and the daily long-term mean-reverting set to 0.01. As for the Normal Inverse Gaussian model it also needs two more parameters: the first one α is set to 0.5 and is related to the kurtosis of the distribution while the second one β is set to 0 and is associated to the asymmetry of the distribution. The time step used for the three methods is 0.125 day. Such a refined time step is not necessary for the valuation itself, but it is to calculate the optimal command that could be used by a Monte Carlo simulator in order to get, for example, the cash distribution generated each month during the two years. The Normal Inverse Gaussian model also needs a step for the *space* discretization. This step is set to 0.0125.

Using the above configuration, our three models yield respectively $1,355,010$, $1,358,930$ and $1,354,630$ which correspond to the renting price of our fictive gas storage space calculated in euros for a two-year period. In this gas storage test case, it can be noticed that the prices obtained by the three models are nearly identical. Nevertheless, other tests have to be carried out in order to determine whether the sophisticated model can outperform the one-factor Gaussian model for different types of gas storage. As it is shown by the performance results in the following section, such an investigation is made possible by our distributed implementation.

5 Large experiment results

5.1 Gaussian algorithm

When using two processors per node, the Blue Gene performances do not decrease: using P processors on P nodes or $P/2$ nodes leads to the same execution time. At the opposite, the dual Opteron cluster performances fall consid-

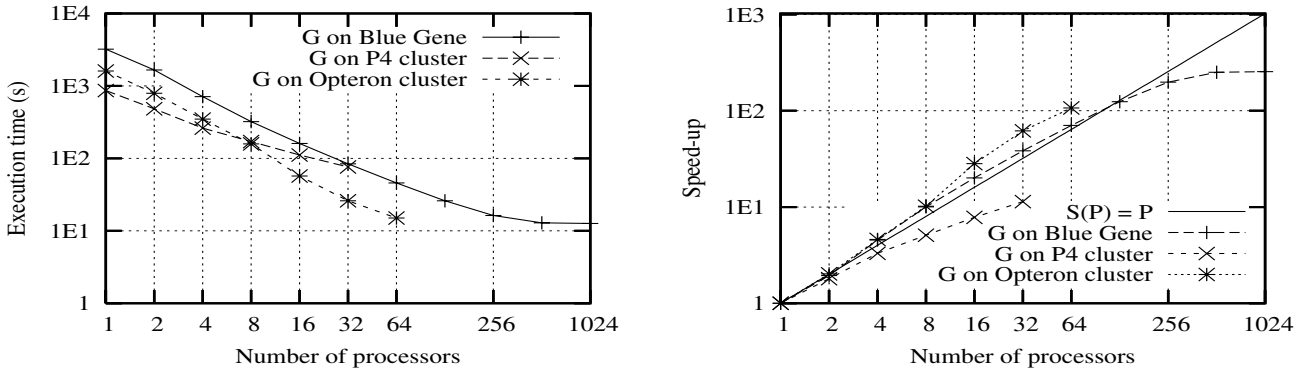

Figure 4. Execution times (in logarithmic scale) and speedups of the Gaussian algorithm on three different distributed architectures, using only one processor per node.

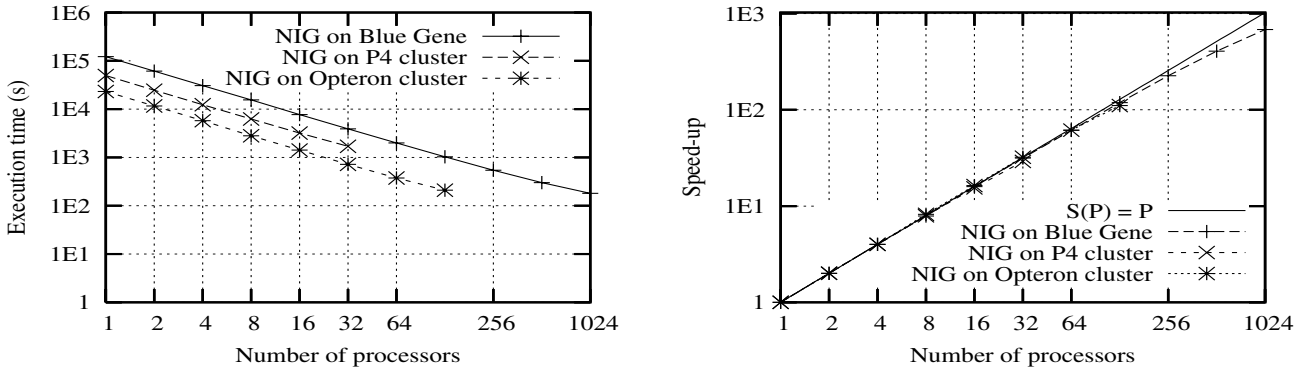

Figure 5. Execution times and speedup of the Normal Inverse Gaussian algorithm on three different distributed architectures, using the maximum number of processors per node.

erably: it is faster to use P processors on P nodes than to attempt to use $2.P$ processors on P nodes! So we use only one processor per node on our PC clusters and two processors per nodes on the Blue Gene to run the distributed *Gaussian* model. Performance measures are summarized on figure 4. When using only one processor per node, the Blue Gene supercomputer and the dual Opteron cluster achieve a superlinear speedup from 4 to 64 processors. This can be explained by the improved cache performance obtained from the smaller memory requirements per node as a result of the distribution of the data. However, this superlinear speedup tends to disappear, and the Blue Gene speedup reaches its maximum at 512 processors. Even on a supercomputer our parallelization of the *Gaussian* algorithm does not scale beyond 512 processors, and its performance surpasses just a little bit the one on our high-end dual Opteron cluster using 64 processors (on 64 nodes). As for the cheap Pentium-4 PC cluster, it does not achieve any superlinear speedup and has a slowly increasing speedup curve. So, a fast interconnection network seems mandatory to achieve good performances on this distributed application, but a medium size monoprocessor PC cluster with a good Gigabit-Ethernet switch can be a sufficient solution to run this distributed *Gaussian* algorithm.

Finally, despite some scalability problems that have been encountered, the best sequential execution time which was close to 15 minutes has been successfully decreased to 13 s - 15 s on a high-end cluster and a Blue Gene supercomputer. This is a real improvement for users, that frequently run this reference algorithm.

5.2 Normal Inverse Gaussian algorithm

Our distributed *Normal Inverse Gaussian* algorithm and implementation have reached very good performances independently of the per-node number of processors that was used. Figure 5 introduces performances achieved using the maximum number of processors per node on the Pentium-4 cluster, the dual Opteron cluster and the Blue Gene supercomputer. The performance curves exhibit an almost per-

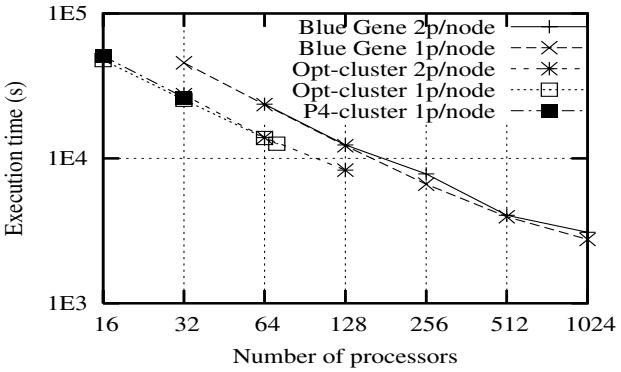

Figure 6. Execution times of 2-factor Gaussian algorithm on 3 different architectures, using 1 or 2 processors per node.

fect parallelization of the *Normal Inverse Gaussian* algorithm on all these architectures, even on the *cheap* Gigabit-Ethernet Pentium-4 cluster. The lowest execution time is achieved by the Blue Gene when using 1024 processors. However, the dual Opteron PC cluster achieves similar performances with only 128 processors. Hence, a large PC cluster with powerful multiprocessor nodes can be an interesting alternative to run our distributed NIG algorithm, and this regardless of its interconnection network.

In the end, our best sequential execution time which nears 6h25 (obtained by an Opteron processor) has been decreased to 3 minutes using 1024 processors of Blue Gene. Thus, our distribution makes it possible for users to use the *Normal Inverse Gaussian* algorithm provided they can mobilize enough computing resources.

5.3 2-factor Gaussian algorithm

The distributed *2-factor Gaussian* algorithm requires both huge amount of CPU and memory. With the current set of parameters (see section 4.4.2) and our implementation which mainly uses two tables - for storing the old and the new results - the application would theoretically require $2 \times 5,895$ MB of memory to execute sequentially. Hence, the application would require $1,474$ MB per node and would easily be run on 8 nodes equipped with 2 GB of RAM. However, due to the nature of the stochastic algorithm and our distribution strategy, the overall memory needed when parallelizing is greater than in the sequential case. Furthermore, the kernel of the host operating system as well as the presence of communications which are handled by MPI contribute to increase the memory use. As a result, in practice, the minimum requirement to run this algorithm without swapping is 10 processors with 2 GB of memory each.

Figure 6 shows the execution times measured. The small 32 Pentium-4 PC cluster has been able to run this benchmark from 16 to 32 processors, with 2 GB of memory per PC (and per processor). On 32 processors the execution time was approximately half of the execution time on 16 processors: the G-2f application seems to scale on this basic cluster. The dual Opteron cluster, which is equipped with 2 GB of memory per PC, could also run our experiments with 16 PCs. Execution times on P processors were approximately the same on P nodes and on $P/2$ nodes, and the benchmark successfully scaled up to 128 processors using 64 nodes and 2 processors per node. Finally, on Blue Gene, with only 1 GB of memory per node, 32 nodes and their memories were required. Execution times were a little bit longer when using P processors on $P/2$ nodes instead of P nodes, but the slow down was not so important. Hence, like on the dual Opteron cluster, the G-2f application has also been run on the Blue Gene using two processors per node. Figure 6 shows that the G-2f application scales very well up to 128 processors on the dual Opteron cluster and up to 1024 processors on the Blue Gene machine. Finally, the Blue Gene machine appears to be the most interesting system to run the long G-2f application.

Our distributed *2-factor Gaussian* algorithm has succeeded in making possible these simulations (using the memory of at least 16 or 32 processors), and has yielded results in 46 minutes on 1024 processors. However, the computation of the usual speedup is impossible since the G-2f benchmark cold not be run on a single processor.

6 Accuracy *vs* execution time

6.1 Interest in constant time computing

All previous calculations have been carried out using a discretization step of the stock level *q-discr* equal to 500 MWh. The *q-discr* factor is not the unique accuracy control parameter of the G-2f simulations, but has a significant impact on this benchmark. When needed, it is important to be able to run these simulations with higher accuracy in close execution time, to get the right results without disturbing the user planning. For example, to maintain the analysis time at a constant value before making a deal or an investment. But it is also important not to mobilize too much computing resources (that could be useful for other urgent computations). So, we have studied the *scalability* of both our application and our distributed systems by trying to identify the exact amount of computing resources to maintain the execution time to $12,000$ s (3h20) when the required accuracy increases.

6.2 Experimental scalability

The number of processors required to run the G-2f benchmark in 12,000 s for different simulation accuracies

Table 1. Processors required to achieve G-2f computations in 12,000 s, function of the accuracy.

q-discr factor:	2000 MWh	1000 MWh	500 MWh	250 MWh
Simulation accuracy:	rough simulations			fine simulations
dual Opteron cluster, 2 proc. per node	18 proc.	38 proc.	88 proc.	-
Blue Gene supercomputer, 1 proc. per node	32 proc.	63 proc.	132 proc.	280 proc.
Blue Gene supercomputer, 2 proc. per node	32 proc.	63 proc.	132 proc.	286 proc.
Simulation result (pricing result in Euros)	11,065 Euros	13,052 Euros	13,589 Euros	13,870 Euros

and on different system configurations, are detailed in table 1. This extensibility experiment shows it is possible to maintain constant the execution time of the G-2f application when increasing its accuracy to get better results. The last row of table 1 shows that the results' values improve and variations minimize when the discretization factor *q-discr* decreases. Table 1 points out that the number of required processors tends to double when the discretization is twice finer. The Blue Gene architecture has been designed to *scale* up to a hundred thousands processors (to reach PetaFlops), and it is easy to mobilize the required number of processors to achieve a simulation with a 250 MWh discretization factor in $12,000$ s, while this was not possible on the dual Opteron cluster. Table 1 shows that 286 Blue Gene processors instead of 280 are required to run a strongly accurate simulation in $12,000$ s when using two processors per node instead of one. As this overhead is small, it is better to use P processors on $P/2$ nodes since it mobilizes less computing resources.

Finally, this scalability benchmark shows that our distributed strategy and implementation *scale* successfully on a Blue Gene as well as on a PC-cluster architecture.

7 Conclusion and perspectives

Our first distribution of a stochastic control algorithm used for gas storage valuation, and the numerous experiments we did on three different distributed systems, have shown it is possible to efficiently speed up and size up stochastic control computations. Thanks to our parallelization strategy which distributes both computations and data, and updates this distribution at each time cycle, we have succeeded (1) in running simulations that required at least 10 processors with 2 GB of memory each, (2) in scaling and (3) in achieving performances on a PC cluster (up to 128 processors) as well as on a Blue Gene supercomputer (up to 1024 processors).

The first uses of our distributed *2-factor Gaussian* algorithm allowed to notice that the choice of the discretization factor is a far more relevant to the result value than the choice of the model. However, in order to be actually used in industrial environments, this research still need to be improved. A N-dimensional version of the stochastic control algorithm, which is used to optimize the global management of several storage devices, still has to be distributed.

During our numerous experiments it appeared that PC clusters are sometimes interesting alternatives to supercomputers, but sometimes they lack of reliability. Several failures, that led to restart the benchmarks, had to be dealt with when using PC clusters, while we never encountered such issues on the IBM Blue Gene/L supercomputer at EDF company. Adding fault tolerance mechanisms in our distributed stochastic control algorithm is another topic of our current research.

Acknowledgment

This research is part of the ANR-CICG GCPMF project, and is supported both by ANR (French National Research Agency) and by Region Lorraine.

References

[1] O. E. Barndorff-Nielsen. Processes of Normal Inverse Gaussian Type. *Finance and stochastics*, 2, 1998.

[2] C. Barrera-Esteve, F. Bergeret, E. Gobet, and al. Numerical methods for the pricing of Swing options: a stochastic approach. *Methodology and Computing in Applied Probability*, 2006.

[3] Z. Chen and P. Forsyth. A semi-lagrangian approach for natural gas storage valuation and optimal operation. Technical report, 2006.

[4] A. V. Gerbessiotis. Trinomial-tree based parallel option price valuations. *International Journal of Parallel, Emergent and Distributed Systems*, 18(4), 2003.

[5] L. Henrio, V. D. Doan, M. Bossy, F. Baude, S. Vialle, V. Galtier, and S. Bezzine. A fault tolerant and multi-paradigm grid architecture for time constrained problems. Application to option pricing in finance. In *e-Science 2006*, Amsterdam, Netherlands, dec 2006. IEEE CS Press.

[6] P. Jaillet, E. Ronn, and S. Tompaidis. Valuation of commodity-based swing options. *Management science*, 50, 2004.

[7] C. Makassikis, X. Warin, and S. Vialle. Distribution of a stochastic control algorithm applied to gas storage valuation. In *The 7th IEEE International Symposium on Signal Processing and Information Technology*, Cairo, Egypt, 2007.