Distribution of a Stochastic Control Algorithm Applied to Gas Storage Valuation

Constantinos Makassikis\textsuperscript{1,2}, Stéphane Vialle\textsuperscript{1}, Xavier Warin\textsuperscript{3}

\texttt{Constantinos.Makassikis@supelec.fr}, \texttt{Stephane.Vialle@supelec.fr}, \texttt{Xavier.Warin@edf.fr}

\textsuperscript{1}SUPELEC, IMS research group, 2 rue Edouard Belin, 57070 Metz, France
\textsuperscript{2}LORIA, ALGORILLE project team, BP 239, 54506 Vandoeuvre-lès-Nancy, France
\textsuperscript{3}EDF - R&D, OSIRIS group, 1 Avenue Charles de Gaulle, 92141 Clamart, France

Abstract

This paper introduces a research project that aims to speed-up and size-up some gas storage valuations, based on a Stochastic Dynamic Programming algorithm. Such valuations are typically needed by investment projects and yield prices of gas storage spaces and facilities. However, they involve computations which require great amounts of CPU power or memory. As a result, their parallelization on PC clusters or supercomputers becomes highly attractive and sometimes unavoidable despite its complexity.

Our parallelization strategy is based on a message passing paradigm, and distributes both computations and data on a cluster, in order to achieve speed-up and size-up. It includes some complex and optimized data exchanges which are dynamically computed, planned and achieved at each computation step. This optimized data distribution and memory management allows to process large problems on a high number of processors. Moreover, our parallel implementation is able to support different price models, and our first experiments on a standard 32 PC cluster show very good performances particularly for complex price models.

1 Introduction and objectives

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 [1], [2], [3] for example). As a result, many different price models can be used to carry out this valorization. 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 [4] and in section 2,
- the second one is based on a two-factor Ornstein-Uhlenbeck process, hence a two-factor model,
- 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 [5].

All valuations are achieved by a stochastic programming approach described in [4] 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.

From a computer science point of view this is a Stochastic Dynamic Programming algorithm which is complex to parallelize. Indeed, despite some natural parallelism (see Stochastic control algorithm of 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 [6] as well as computations requiring many communications between processors [7]. However, gas storage valuation equations need to take into account gas stock levels and lead to different computations and distributions.

Section 2 introduces the mathematical problem and the initial algorithm. Section 3 lists the development tools used by the sequential version and kept in the distributed version. The distributed and optimized algorithm for clusters is described in section 4, and the first experimental results are introduced in section 5. Finally, section 6 summarizes this research and our current results.

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:

\[
\begin{align*}
\text{Injection} & \quad a_{in,s}(I_t), \text{ with cost:} \\
& \quad \phi_{-1}(I_t, I_t) = -S_t a_{in}(I_t) - K_{in}(I_t) \\
\text{Storage} & \quad \text{with cost:} \\
& \quad \phi_{0}(I_t, I_t) = -K_{s}(I_t) \\
\text{Withdrawal} & \quad a_{out,c}(I_t), \text{ with gain:} \\
& \quad \phi_{1}(I_t, I_t) = S_t a_{out}(I_t) - K_{out}(I_t)
\end{align*}
\]

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

\[
\begin{align*}
\frac{dI_t}{dt} &= a_{in,s}(I_t) & \text{in injection} \\
\frac{dI_t}{dt} &= 0 & \text{in storing} \\
\frac{dI_t}{dt} &= -a_{out,c}(I_t) & \text{in withdrawal}
\end{align*}
\]

If 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, and if we suppose that the regime switching can occur at any date, the gain obtained by managing the facility from a date \( t \) until to a date \( T \) with a strategy \( u \) is given by:

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

where:

- \( J(T, S_T, I_T, i_T, u_T) \) a given final value function,
- \( s \) the gas price in \( t \) given by a markovian process,
- \( c \) the stock level in \( t \),
- \( i \) the regime in \( t \).

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

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

2.2 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 with equally spaced levels. Furthermore, the regime switching occurs only at given dates (once a day): so we discretize the time with an equally space step \( \Delta t \). 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.

For \( t := (N - 1) \Delta t \) to 0

For \( s \in \text{all possible stock levels} \)

For \( c \in \text{admissible stock levels} \)

For \( s \in \text{all possible price levels} \)

\[
J^*(s, c) := \max \left\{ -(a_{in,s} + K_{in}) \Delta t + \mathbb{E}(J^*(s_{t+\Delta t}, c + a_{in} \Delta t)|S_t = s), \right. \\
\left. -(a_{out,s} - K_{out}) \Delta t + \mathbb{E}(J^*(s_{t+\Delta t}, c - a_{out} \Delta t)|S_t = s), -K_{s} \Delta t + \mathbb{E}(J^*(s_{t+\Delta t}, c)|S_t = s) \right\}
\]

\[
J^* := \tilde{J}^* \quad // \text{Set } J^* \text{ for the next time step}
\]

Figure 1: Theoretical stochastic control algorithm

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 4).
3 Development tools

The original and sequential application was composed of a Python main program calling some efficient C++ computing functions. This programming strategy has some advantages for researchers and engineers in financial computing. C++ is used to implement the computing library very efficiently, while Python is used to implement non intensive computing parts of the application, and allows to develop very quickly various main programs. In order to develop a distributed application adapted to users, we have decided to maintain this strategy. Therefore, we have designed a distributed application composed of sequential computing routines implemented in C++, a C-MPI library, and a toplevel Python program to deploy processes on the different processors, call the sequential computation routines and manage data exchange between processors. We have chosen the "PyPar-1.9.2" MPI interface module for Python [8], which is an interface to well-known and efficient C-MPI libraries (such as mpich-1.2.7), instead of full MPI re-implementations in Python. In the process, we have improved this Python/C-MPI interface in order to use a "buffered" message send routine (MPI_Bsend) instead of a basic send routine (MPI_Send), and to support modern NumPy arrays in MPI routines.

The implementation of our stochastic algorithms involves many large arrays. They are implemented in the Python code with arrays of the Numpy-1.0.1 package, and in the C++ code with arrays of the blitz-0.9 library. The latter allows to easily allocate, resize and manipulate large arrays. Finally, the interface between Python and C++ is achieved by the convenient Boost.Python-1.33.1 library.

4 Optimized distribution

4.1 Parallelization strategy

To achieve large speed-up 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 communicating by message passing. However, the range of stock levels to process changes at each temporal step, leading to redistribute computations and data at each step.

As illustrated on figure 2, stock level loop (from Cmin_{i+1} to Cmax_{i+1}) is load balanced on processors at step i + 1, and each processor stores its results. At step 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 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. 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. In order not to require too large temporary message buffers on processors, and not to overload the cluster interconnection network, the routing plan is split in (P − 1)/2 steps. Processors are considered on a virtual ring, and at step i each processor exchanges data only with its two neighbors at distance i (see figure 3). This routing scheme could need improvement for the custom interconnection network of a supercomputer, but its low memory and controlled bandwidth consumption makes it suitable for a basic PC cluster with classic gigabit Ethernet links and switches.

In order 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.

4.2 Distributed algorithm design and implementation

According to the strategy introduced in the previous subsection, we have designed the distributed algorithm de-
Figure 3: Routing scheme adopted on PC cluster

(only messages from and to processor #i are printed on this figure)

Figure 4: Distributed algorithm for stochastic control

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 sub-steps: 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. Our current Python-MPI (Pypar) development tool does not support asynchronous communications, and is limited to the basic and implementation dependent MPI_Send routine. We have improved Pypar by adding support for the MPI_Bsend routine. This solution takes time to copy data into buffers, but allows to control the size and the allocation of the communication buffers. This allows us to control the total amount of memory required by our implementation and ensure that memory will not hinder the scalability of the application. The execution of the routing plan ends the data exchange planning and execution sub-step. The different send and receive operations are scheduled to use small and limited communication buffers and not to overload standard cluster communication networks (see previous subsection on parallelization strategy).

The new computation processing sub-step is illustrated
at the bottom of figure 4. It consists in 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).

5 Performance measurements

5.1 Test applications

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 are highly dependent on the stock level and have values ranging between 100 and 1,000 MWh per day:

- 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. The annual interest rate of 8% that we used for the valuation is the one provided by Zeebruge at the beginning of year 2006 (the Stock Exchange of Zeebruge fixes gas prices). 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 $\alpha$ is set to 0.5 and is related to the kurtosis of the distribution while the second one $\beta$ 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 was set to 0.0125. The renting price of our fictive gas storage space is calculated in euros for a two-year period. Our three models yield respectively 1,355,010 and 1,358,930 and 1,354,630.

In this gas storage 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. As it is shown by the performance results in the following section, such an investigation is made possible by our distributed implementation.

5.2 Experimental performances

The experiment testbed was a classic 32 PC cluster with Pentium-IV processors at 3 GHz, 2 GB of main memory and a Gigabit Ethernet interconnection network. No other application was running during the benchmarks.

Table 1 and figure 5 show the execution times and speed-ups achieved for the three benchmarks with our implementation based on Python-C++ and a subset of MPI functionalities included in Pypar. The first benchmark (“G-1fac.”) is associated to the one-factor Gaussian price model and is classically used for gas storage valuation. It includes frequent communications and small amount of computations during each temporal step. Thus, it is not surprising that its speed-up is limited to 10 on 32 processors. However, this model is used as a reference model to evaluate results of more complex models, and is frequently run for long-term simulations. Thus, this speed-up of 10 remains very attractive.

The third benchmark (“NIG”), which is associated to the normal inverse Gaussian model, involves long computations between communication steps. It achieves a speed-up close to 30 on 32 processors, with an efficiency greater than 92%, and an execution time decreased from...
13h43 on a single processor to 0h28 on 32 processors. Obviously, such improvement will facilitate the study of this complex model.

The second benchmark ("G-2fac.") is associated to the two-factor Gaussian model and is an experiment of another complex model requiring both a lot of CPU and a lot of memory. The latter requirement is such that the memory of more than eight nodes of our cluster is necessary to run the benchmark without swapping. Thanks to our data distribution the computation has been successfully achieved. Execution times obtained when using 16 and 32 machines have been reported in table 1. Despite the excessive long duration which is displayed, the execution time is divided by a factor of 1.93 when switching from 16 to 32 processors. This benchmark illustrates both size-up and speed-up abilities of our implementation.

6 Conclusion and perspective

We have designed and implemented a stochastic dynamic programming algorithm for gas storage valuation. This distributed algorithm supports different price models of gas storage spaces and facilities, and includes an optimized dynamic data distribution and memory management in order to achieve both speed-up and size-up. The distributed implementation introduced in this paper is based on a MPI-Python interface (Pyfar), and on C++ computing routines. It achieves very good speed-ups and high efficiency (greater than 92%) on a complex valuation model run on our standard 32 PC cluster. Moreover, our implementation allows to run and speed up another complex model requiring the memory of more than 8 PCs to install all its datastructures.

Finally, the speed-up curve of the Normal Inverse Gaussian model on figure 5 is close to the ideal speed-up, and can continue to increase on larger clusters. This parallelization allows fast investigations of complex and original valuation models for gas storage spaces and facilities, and will help to design better models of gas storage valuation.

In order to track better performances, a pure MPI-C++ version is under development. The current implementation uses the non-blocking MPI_Issend communication routine which does not require any additional buffers, and has already achieved a slight improvement on 32 processors with our least favorable benchmark ("G-1fact."). Memory consumption has been reduced while speed-up has increased from 10 to 11.5. This optimized MPI-C++ version will be tested using complex price models on larger clusters and on an IBM BlueGene/L supercomputer (up to 8000 processors) in a near future. The next step will consist in improving the software architecture in order to be able to plug and experiment new price models very easily.

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


Résumé
Face à la demande croissante en termes de puissance de calcul que connaît le monde des calculs financiers, les solutions fondées sur le parallélisme deviennent incontournables. De nombreuses applications de calculs financiers se décomposent simplement en tâches indépendantes et donnent lieu à du bag of task, mais certaines applications ne se prêtent pas à ce type de décomposition. Dans ce papier, nous présentons une parallélisation originale de l'algorithme d'une telle application de calcul financier appliquée à la valorisation d'un actif de stockage de gaz et réalisée dans le cadre du projet ANR GCPMF. Les expérimentations à large échelle que nous avons menées sur un cluster de PCs de Grid'5000 (jusqu'à 128 processeurs) et sur le supercalculateur Blue Gene/L d'EDF R&D (jusqu'à 1024 processeurs) affichent de très bonnes performances et permettent de comparer ces systèmes non seulement à partir leur temps d'exécution mais aussi en termes de fiabilité expérimentale.

Mots-clés : Contrôle stochastique, distribution large échelle, clusters de PCs, Blue Gene/L

1. Introduction

1.1. Contexte financier
Un actif de stockage de gaz est habituellement constitué d’une cavité destinée à contenir le gaz ainsi que de compresseurs permettant d’effectuer des injections, des sous-tirages ou simplement de maintenir le gaz sous pression dans la cavité. En temps normal, le propriétaire remplit son actif de stockage pendant les périodes où les prix du gaz sont bas pour répondre aux demandes des clients ou pour le revendre ultérieurement sur les marchés. Éventuellement, il peut envisager de louer une partie de sa capacité de stockage à une autre entité. Les prix du gaz connaissent des fluctuations issues principalement de la modification de la demande. À cause de l’inélasticité entre la production et la demande, les prix du gaz sont, par exemple, plus élevés en hiver qu’en été. Le propriétaire d’un actif de stockage de gaz peut alors profiter de cette dynamique des prix en arbitrant entre les différentiels temporels des prix du gaz et ainsi valoriser son actif. La détermination du prix de location tient compte de l’opportunité d’arbitrage mais elle est également soumise à des contraintes. D’une part, il existe des contraintes physiques liées à la cavité et la manière de stocker le gaz. Par exemple, une injection (resp. un

---

*Ce travail est le fruit de la collaboration avec l’équipe Osiris de EDF R&D dans le cadre du projet ANR GCPMF.*
La valorisation fait appel à des algorithmes de contrôle stochastique et à des modèles de prix du gaz. De nombreux travaux de recherche récents dans le domaine de la valorisation des actifs de stockage de gaz (cf. [1, 2]) ont conduit à l’élaboration de ces modèles. Cependant, les besoins de ces modèles en termes de puissance de calcul et de consommation mémoire peuvent aisément surpasser les capacités des machines monoprocesseur actuelles ce qui rend très difficile leur utilisation en environnement industriel où ils sont, de surcroît, soumis à des contraintes de temps. Dès lors, la conception et l’implantation d’algorithmes parallèles faisant intervenir ces modèles devient incontournable. C’est l’un des objectifs du projet ANR GCPMF dans lequel s’inscrivent ces recherches.

1.2. Défis informatiques
D’un point de vue informatique, l’algorithme auquel on s’intéresse est un algorithme itératif de programmation dynamique stochastique qui est complexe à paralléliser et ce, malgré la présence de parallélisme (cf. Section 2.1). En effet, les calculs à chaque étape dépendent des résultats de l’étape précédente, et les données à calculer varient régulièrement. Ces contraintes conduisent à redistribuer les données et les calculs à chaque étape. Par ailleurs, les données requises par chaque processeur doivent être soigneusement identifiées afin de n’avoir à redistribuer et à stocker qu’un minimum de données sur chaque processeur. Cette stratégie est nécessaire pour pouvoir exécuter des problèmes de très grande taille sur de nombreux processeurs. Une diffusion (broadcast) à chaque étape de toutes les données sur tous les processeurs serait facile à implanter mais nécessiterait trop de mémoire sur chaque processeur.

Notre papier est organisé comme suit. Après la section 2 où est présenté en détail l’algorithme distribué que nous avons conçu, la section 3 se concentre sur l’analyse des performances obtenues par l’algorithme. Enfin, la section 4 résume notre travail de recherche et présente des perspectives.

2. Distribution de l’algorithme

2.1. Algorithme séquentiel et difficultés de distribution
La figure 1 présente l’algorithme séquentiel qui est utilisé pour effectuer la valorisation. Cet algorithme consiste à balayer toute la période de valorisation de la date finale à la date initiale et à déterminer, à chaque étape de ce parcours, l’action la plus intéressante à entreprendre : injecter, sous-tirer ou ne rien faire. La décision prise dépend d’une part des niveaux de stocks dont les valeurs sont conditionnées par différentes contraintes (cf. section 1.1), d’autre part de la valeur des prix renseignée par le modèle de prix utilisé et finalement des décisions et résultats du pas de temps précédent qui sont sauvegardés dans la structure de données OldRes à la fin de chaque itération en temps. Une parallélisation au niveau de la boucle temporelle s’avère donc impossible. En revanche, une parallélisation de la boucle sur les niveaux de stocks est possible, mais nécessitera un échange complexe de données entre les processeurs à chaque pas de temps (cf. section 2.2).

2.2. Algorithme distribué
Stratégie de distribution
Afin de réaliser des accélérations importantes et permettre le passage à l’échelle, l’algorithme de la figure 1 a été parallélisé pour des architectures extensibles tels les clusters de PC’s et les
For $t := (N-1)\Delta t$ to 0
For $c \in$ niveaux de stock admissibles
For $s \in$ valeurs de prix possibles
\[ NewRes[s, c] = \text{calcul}(OldRes, s, c) \]
// Recopie pour le pas de temps suivant
OldRes := NewRes

Fig. 1 – Algorithme séquentiel simplifié de contrôle stochastique.

Fig. 2 – Exemple sur trois processeurs d’une redistribution de données optimisée.

supercalculateurs à mémoire partagée.

Comme le montre la figure 2, à l’étape $i+1$, chaque processeur effectue les calculs pour l’ensemble contigu des niveaux de stocks - compris entre $C_{\text{min}}_{i+1}$ à $C_{\text{max}}_{i+1}$ - qui lui ont été attribués. À l’étape $i$ (étape suivante), la nouvelle boucle sur les niveaux de stock (de $C_{\text{min}}_{i}$ à $C_{\text{max}}_{i}$) est partagée entre les processeurs. Pour effectuer sa nouvelle tâche, chaque processeur a besoin de certains résultats de l’itération précédente dont il n’est pas nécessairement en possession. Ainsi, chaque processeur détermine d’une part les données en sa possession dont auront besoin les autres processeurs, et d’autre part les données qui lui manquent et sont détenues par d’autres processeurs ; ces informations sont alors utilisées pour dresser un plan de routage dans lequel figurent toutes les communications qu’il devra effectuer. Ce plan est finalement exécuté selon un schéma de communication préétabli.

Algorithmèse distribué détaillé

En suivant la stratégie décrite dans le paragraphe précédent, nous avons conçu l’algorithme distribué décrit dans la figure 3. La première étape de cet algorithme consiste à partager les calculs à $t_n$ en utilisant des routines d’initialisation spécifiques. On rentre ensuite dans une boucle de $n$ itérations (de $t_{n-1}$ à $t_0$) qui est constituée de deux sous-étapes. La première consiste à planifier et à exécuter l’échange des données. La seconde correspond à la nouvelle phase de calculs.

Pendant la première sous-étape chaque processeur commence par déterminer la totalité du nouveau plan de partage des calculs et la nouvelle distribution des données. Les calculs impliqués par cette étape sont très simples et ne présentent donc aucun intérêt à être distribués. Ensuite chaque processeur établit son plan de routage et redimensionne ses tables locales conformément aux besoins de la nouvelle distribution des données. L’échange effectif des données s’effectue immédiatement après et repose sur des communications point-à-point.

La seconde sous-étape consiste en un calcul local à chaque processeur suivant l’algorithme de contrôle stochastique partiellement décrit dans la figure 1.

2.3. Implantations

L’algorithme présenté dans la section précédente a donné lieu à deux implantations. La première privilégiait la facilité d’utilisation en combinant Python avec du MPI et du C++. L’avantage de Python est de permettre de facilement paramétrer l’application et d’ajouter rapidement des modules tels que des modules de visualisation.

Quant à la deuxième implantation, elle vise la performance pure et a été écrite entièrement en...
C/C++ et MPI. Cette implantation offre le choix entre trois primitives MPI pour effectuer les communications : la primitive bloquante `MPI_Bsend()` ainsi que les primitives non-bloquantes `MPI_Ibsend()` et `MPI_Issend()`. La primitive `MPI_Send()` a rapidement été écartée à cause de son manque de portabilité entre implantations de MPI. Parmi les trois utilisées, les primitives `MPI_Ibsend()` et `MPI_Issend()`, en permettant plusieurs communications en parallèle (lorsque le réseau d’interconnexions le permet), se sont montrées les plus rapides. En échange d’un surcoût lié à la synchronisation inhérente à `MPI_Issend()`, cette dernière ne nécessite aucune allocation ni gestion explicite de buffers. Cette économie la rend très intéressante pour la distribution d’applications gourmandes en mémoire.

3. Expérimentations

3.1. Environnements et modèles de prix

L’évaluation des performances de notre algorithme parallèle a été réalisée sur trois architectures distribuées différentes et ce, en utilisant trois modèles de prix variés.

L’environnement d’expérimentation comprenait deux clusters de PCs et un supercalculateur :
- le cluster de Pentium 4 de SUPELEC qui est composé de 32 PCs reliés par un réseau Gigabit-Ethernet bon marché : chaque PC comporte un processeur Pentium 4 à 3 GHz et 2 Go de mémoire ;
- le cluster de bi-Opteron de l’INRIA qui dispose de 72 PCs reliés par un réseau Gigabit-Ethernet de très bonne qualité : chaque PC comporte deux processeurs Opteron monocores à 2 GHz et 2 Go de mémoire ; ce cluster est situé sur le site de Sophia de la plate-forme française d’expérimentation Grid’5000 ;
- le supercalculateur Blue Gene/L d’EDF R&D doté de 4096 noëuds et d’un réseau Gigabit-Ethernet hyper spécialisé. Sa configuration par noeud est, de loin, la moins puissante parmi les trois architectures considérées : deux processeurs PowerPC à 700 MHz et 1 Go de RAM.

Les modèles de prix utilisés comprenaient :
- un modèle de prix *gaussien à un facteur* : c’est un modèle de référence qui est couramment
utilisé de part sa rapidité d’exécution ;
– un modèle de prix *normal inverse gaussien* qui est beaucoup plus lourd que le modèle gaussien
  en termes de calculs : ceci rend son utilisation moins fréquente ;
– un modèle de prix *gaussien à deux facteurs* : des trois modèles utilisés c’est de loin le modèle
  le plus lourd aussi bien en termes de calculs qu’en termes d’espace mémoire nécessaire à son
  exécution.

Plus de détails sur ces modèles mathématiques sont disponibles dans [3].

Avec la configuration des modèles prise pour les expériences, les zones de recouvrement sont
constantes au sein d’un modèle ce qui conduit à un volume de données échangées constant par
processeur et par itération. Les volumes de données par itération augmentent linéairement par
rapport au nombre de processeurs et nous avons en moyenne sur 32 processeurs 1 Mo de données
échangées pour les modèles gaussien et normal inverse gaussien contre 349 Mo pour le gaussien
à deux facteurs.

3.2. Analyse des performances
Le cluster de bi-Opteron et le supercalculateur Blue Gene offrent la possibilité d’exécuter des
applications selon deux configurations : soit en utilisant un processeur par noeud soit en utilisant
deux processeurs par noeud. Ces deux configurations seront désignées par le suite respectivement
par *mode monopro* et *mode bipro*.

**Modèle gaussien à un facteur**

Sur ce modèle, le supercalculateur Blue Gene se comporte aussi bien dans les deux modes. Ce
n’est pas le cas du cluster de bi-Opteron dont les performances se dégradent considérablement
en mode bipro à partir de 32 processeurs. Des mesures expérimentales ont montré que l’augmen-
tation du nombre de processeurs accentuait de manière significative le temps passé sur les
communications en mode bipro par rapport au mode monopro : il semblerait que l’accès à la
carte réseau constitue une source d’engorgement pour les processeurs d’un même noeud. La suite
se concentre sur l’analyse des performances en mode monopro présentes à la figure 4.

L’observation des courbes d’accélérations de la figure 4 rend compte d’une hyperaccélération
entre 4 et 64 processeurs pour le supercalculateur Blue Gene et pour le cluster bi-Opteron que
nous attribuons à une augmentation simultanée du nombre de processeurs et de la quantité
totale de mémoire disponible. Sachant que les noeuds sur Blue Gene disposent de deux fois
moins de mémoire que les noeuds du cluster bi-Opteron, il n’est pas surprenant de voir ce
dernier hyperaccélérer davantage. Dans tous les cas, l’hyperaccélération tend à disparaître avec
l’augmentation du nombre de processeurs et l’accélération maximale est obtenue par Blue Gene
sur 512 processeurs. Le cluster de Pentium 4 ne réalise aucune hyperaccélération et sa courbe
d’accélération croît lentement. Il s’avère donc qu’un réseau d’interconnexions très rapide est
indispensable pour obtenir de bonnes performances sur ce modèle, mais dès lors un cluster de
taille moyenne suffit pour en accélérer l’exécution (cf. section 3.3).

En dernier lieu, malgré quelques problèmes de passage à l’échelle que nous avons rencontrés, le
meilleur temps séquentiel obtenu, proche de 15 min, a pu être réduit à 13 — 15 s sur un cluster
de PCs et sur un supercalculateur Blue Gene. Il s’agit donc d’une réelle amélioration pour les
utilisateurs étant donné que c’est le modèle le plus utilisé.

**Modèle normal inverse gaussien**

Que ce soit en mode monopro ou bipro les performances obtenues par notre algorithme et son
implantation sur le modèle normal inverse gaussien sont très bonnes. Sur la figure 5 où ont été
Fig. 4 – Temps d’exécution et accélérations (échelle logarithmique) avec le modèle gaussien sur trois architectures distribuées différentes en utilisant un seul processeur par noeud.

Reportées les performances mesurées en utilisant le nombre maximal de processeurs par noeuds (i.e. : mode bipro), nous observons une parallélisation quasi-parfaite et ce, même sur le cluster bas de gamme à base de Pentium 4. Le meilleur temps d’exécution est réalisé sur Blue Gene avec 1024 processeurs. Cependant, le cluster de bi-Opteron réalise une performance similaire avec seulement 128 processeurs. Ainsi, une alternative intéressante pour exécuter le modèle normal inverse gaussien avec notre algorithme est un cluster de PCs de grande taille muni de processeurs puissants sans être nécessairement reliés par un réseau ultra rapide.

Finalement, le meilleur temps séquentiel qui est proche de 6h25 et qui a été obtenu par un processeur Opteron, a pu être réduit à 3 min en utilisant 1024 processeurs de Blue Gene. Il en résulte que notre distribution, moyennant la disponibilité d’un nombre suffisant de processeurs, rend possible une utilisation beaucoup plus courante du modèle normal inverse gaussien.

Fig. 5 – Temps d’exécution (échelle logarithmique) sur trois architectures distribuées différentes. À gauche : avec le modèle normal inverse gaussien en utilisant le nombre maximal de processeurs par noeud. À droite : avec le modèle gaussien à deux facteurs en utilisant le nombre minimal ou maximal de processeurs par noeud.
Modèle gaussien à deux facteurs

Ce modèle nécessite une puissance de calcul et un espace mémoire considérables en vue de s’exécuter dans des temps raisonnables/acceptables. Avec le jeu de paramètres courant et notre implantation qui utilise principalement deux tables - pour stocker les anciens et les nouveaux résultats - l’application aurait théoriquement besoin de \(2 \times 5895\) Mo de mémoire pour s’exécuter en séquentiel. En théorie, une distribution sur 8 noeuds nécessiterait 1474 Mo (i.e. : \((2 \times 5895)/8\)) de RAM par noeud, donc 8 noeuds munis de 2 Go de RAM devraient supporter cette application. Cependant, en pratique, d’autres paramètres entrent en considération. Par exemple, la taille du noyau du système d’exploitation ne peut être négligée. Par ailleurs, à cause de la structuration de notre algorithme parallèle, la quantité de mémoire requise lors de la parallélisation est plus importante que dans le cas séquentiel. En l’occurrence, comme chaque processeur stocke les données influençant ses calculs, une légère réplication des données est possible au niveau des frontières de calculs entre processeurs. Il en résulte, qu’en pratique, cet algorithme nécessite au moins 10 processeurs dotés de 2 Go de mémoire chacun pour s’exécuter sans avoir recours au swap. Ainsi, selon le système, et sachant que l’on effectue nos tests avec des nombres de processeurs qui sont une puissance de 2, les courbes des temps d’exécution de la figure 5 débutent soit à 16 soit à 32 processeurs. Le petit cluster de Pentium-4 avec ses 32 noeuds monoprocesseurs équipés de 2 Go de mémoire a réussi à exécuter ce benchmark sur 16 et 32 processeurs. Avec une réduction du temps sur 32 processeurs d’un facteur deux par rapport au temps sur 16 processeurs, l’application semble réaliser un début de passage à l’échelle sur ce modeste cluster. Le cluster de bi-Opteron, avec 2 Go de mémoire par noeur, réussi également à exécuter le modèle Gaussien à deux facteurs à partir de 16 noeuds en mode monopro. Les temps d’exécution affichés en mode bipro sont très similaires à ceux en mode monopro et le benchmark supporte très bien le passage à l’échelle jusqu’à 128 processeurs sur 64 noeuds et en utilisant deux processeurs par noeur. En ce qui concerne le supercalculateur Blue Gene, avec seulement 1 Go de mémoire par noeur, la mémoire de 32 noeuds est nécessaire. Les temps en mode bipro sont légèrement plus importants que ceux en mode monopro mais le ralentissement observé est relativement faible. Comparé aux clusters de PCs, le supercalculateur Blue Gene qui est doté de la configuration par noeur la moins puissante connaît des temps d’exécution sur 32 processeurs beaucoup plus importants. Cependant grâce à son architecture ultrascallable [4] qui allie un excellent réseau d’intéroconnexions avec de nombreux noeuds, le supercalculateur Blue Gene démontre sa supériorité en finissant loin en tête.

3.3. Vers la meilleure solution

Au cours de nos expérimentations le supercalculateur Blue Gene, avec ses milliers de processeurs, s’est avéré être la solution la plus intéressante pour des applications présentant un fort potentiel de passage à l’échelle. Dans d’autres cas, les clusters de PCs se présentent comme des alternatives très attrayantes étant donné leur bonnes performances et surtout leur prix modique à l’achat. Cependant, les clusters de PCs affichent deux inconvénients majeurs :
- ils ne sont pas fiables : plusieurs pannes sont venues perturber nos expériences sur les clusters de PCs, alors que rien de tel n’est survenu sur Blue Gene ;
- malgré un prix d’achat très faible comparé à un supercalculateur comme Blue Gene, le coût global d’un cluster de PCs peut devenir plus important. En effet, l’absence de fiabilité accroît les ressources humaines nécessaires pour les administrer. Par ailleurs, les applications destinées à y être exécutées ont parfois besoin d’algorithmes adaptés, plus longs à développer, afin de réaliser de bonnes performances sans pour autant disposer de réseaux d’interconnexions très efficaces.
D’un point de vue technique, nos expériences démontrent la faisabilité d’obtenir de bonnes performances pour les modèles gaussien à un facteur et normal inverse gaussien sur des clusters de PCs. Cependant, d’un point de vue industriel, notamment au sein des laboratoires d’EDF R&D, le supercalculateur Blue Gene est apparu comme une solution très performante et fiable pour tous les modèles, et finalement très rentable.

4. Conclusion et perspectives

La distribution d’un algorithme de contrôle stochastique utilisé pour la valorisation d’actifs de stockage de gaz qui a été présentée ainsi que les expériences menées sur trois architectures distribuées différentes montrent qu’il est possible d’accélérer et de passer à l’échelle des calculs de contrôle stochastique. La stratégie de parallélisation adoptée, qui consiste à optimiser aussi bien la taille des données sur chaque processeur que la taille et le nombre de messages au prix de schémas de communication complexes, s’est averée convenir pour l’exécution de modèles aux besoins variés. En particulier, elle facilite l’étude d’un modèle très lourd en termes de calculs et de mémoire en rendant possible son exécution en des temps raisonnables sur des clusters de PCs et sur un supercalculateur Blue Gene. En dernier lieu, les performances obtenues présentent les clusters de PCs comme des alternatives intéressantes aux supercalculateurs sur certaines exécutions mais à condition qu’ils soient pourvus de mécanismes de tolérance aux pannes. C’est un des sujets de recherche que nous étudions actuellement et que nous comptons appliquer à notre algorithme distribué de contrôle stochastique.

Remerciements

Ce travail de recherche fait partie du projet ANR-CICG GCPMF qui est supporté par l’ANR et la Région Lorraine. Nous remercions également Xavier Warin de l’équipe Osiris à EDF R&D pour ses développements et son aide sur le contrôle stochastique, et la réalisation des mesures de performances sur le supercalculateur Blue Gene.

Bibliographie

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:

\[ \frac{dF(t, T)}{F(t, T)} = \sigma_S(t) e^{-\alpha_S(T-t)} dz_t^S \]

where \( z_t^S \) is a brownian motion on a probability space \((\Omega, F, P)\) endowed with a filtration \( \{F_t, t \in [0, T]\}\), \( \sigma_S \) is the short-term volatility, \( \alpha_S \) 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:

\[ \frac{dF(t, T)}{F(t, T)} = \sigma_S(t) e^{-\alpha_S(T-t)} dz_t^S + \sigma_L(t) e^{-\alpha_L(T-t)} dz_t^L \]

where \( z_t^L \), \( \sigma_L(t) \), and \( \alpha_L \) 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]:

\[ F(t, T) = F(t, 0) e^{M(t, T) + \int_0^t \sigma_S(u) e^{-\alpha_S(T-u)} dL_u} \]

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

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

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:

\[
\begin{align*}
\text{Injection} & : a_{in}(I_t), \text{with cost:} & \phi_{-1}(S_t, I_t) = -S_t a_{in}(I_t) - K_{in}(I_t) \\
\text{Storage} & : \text{with cost:} & \phi_0(S_t, I_t) = -K_s(I_t) \\
\text{Withdrawal} & : a_{out}(I_t), \text{with gain:} & \phi_1(S_t, I_t) = S_t a_{out}(I_t) - K_{out}(I_t)
\end{align*}
\]
The instantaneous evolution of the stock \( I_t \) depends on the regime of the facility:

\[
\begin{align*}
&dI_t = a_{in,s}(I_t)dt & \text{in injection} \\
&dI_t = 0 & \text{in storing} \\
&dI_t = -a_{out,c}(I_t)dt & \text{in withdrawal}
\end{align*}
\]

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:

\[
J(t, s, c, i, u) = \mathbb{E} \left( \int_t^T \phi_u(r, S_r, I_r) dr + J(T, S_T, I_T, i_T, u_T) \right| S_t = s, I_t = c, u_t = i)
\]

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 \( I^*_{target} \);
- \( s \) 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 \( \Delta t \). 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

\[
\begin{align*}
J^*(s, c) := & \max \left( -a_{in,s} + K_{in}\Delta t + \mathbb{E} \left( J^*(S_{t+\Delta t}, c + a_{in}\Delta t) \right| S_t = s) , \\
& (a_{out}s - K_{out})\Delta t + \mathbb{E} \left( J^*(S_{t+\Delta t}, c - a_{out}\Delta t) \right| S_t = s) , \\
& -K_s\Delta t + \mathbb{E} \left( J^*(S_{t+\Delta t}, c) \right| S_t = s) \right)
\end{align*}
\]

\[
J^* := \bar{J}^* \quad \text{for the next time step}
\]

#### 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:

\[
\frac{\partial f}{\partial t} - \int_{\mathbb{R}} \left( f(x + y) - f(x) - \frac{\partial f}{\partial x}(x)y \right) K^{NIG}_{\alpha,\beta,\delta}(y)dy - \frac{\sigma^2}{2} \frac{\partial^2 f}{\partial x^2} S(x) = 0
\]

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
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 \(C_{\text{min}i+1}\) to \(C_{\text{max}i+1}\)) 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 \(C_{\text{min}i}\) to \(C_{\text{max}i}\)) 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 sub-steps: 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_Isend(). Non-blocking versions allow each PC to parallelize and overlap its message sending and receiving at each step, and the non-blocking MPI_Isend() 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 valued 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 $\alpha$ is set to 0.5 and is related to the kurtosis of the distribution while the second one $\beta$ 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-
erably: it is faster to use $P$ processors on $P$ nodes than to attempt to use $2P$ 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-
f ect 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 is near 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 could 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.

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

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 GCMPF project, and is supported both by ANR (French National Research Agency) and by Region Lorraine.

References