Design and Experimentation of a Large Scale Distributed Stochastic Control Algorithm Applied to Energy Management Problems

Xavier Warin
EDF
France

Stephane Vialle
SUPELEC & AlGorille INRIA Project Team
France

Abstract

The Stochastic Dynamic Programming method often used to solve some stochastic optimization problems is only usable in low dimension, being plagued by the curse of dimensionality. In this article, we explain how to postpone this limit by using *High Performance Computing*: parallel and distributed algorithms design, optimized implementations and usage of large scale distributed architectures (PC clusters and Blue Gene/P).

1. Introduction and objectives

Stochastic optimization is used in many industries to take decisions facing some uncertainties in the future. The asset to optimize can be a network (railway, telecommunication [Charalambous et al. (2005)]), some exotic financial options of american type [Hull (2008)]. In the energy industry, a gaz company may want to optimize the use of a gaz storage [Chen and Forsyth (2009)], [Ludkovski and Carmona (2010, to appear)]. An electricity company may want to optimize the value of a powerplant [Porchet et al. (2009)] facing a price signal and dealing with operational contraints : ramp contraints, minimum on-off times, maximum number of start up during a period.
An integrated energy company may want to maximize an expected revenue coming from many decisions take :

- which thermal assets to use ?
- how should be managed the hydraulic reservoirs ?
- which customers options to exercice ?
- how should the physical portfolio be hedged with future contracts ?

In the previous example, due to the structure of the energy market with limited liquidity, the management of a future position on the market can be seen as the management of a stock of energy available at a given price. So the problem can be seen as an optimization problem with many stocks to deal with. This example will be taken as a test case for our performance studies.

In order to solve stochastic optimization problems, some methods have be developed in the case of convex continuous optimization : The *Stochastic Dual Dynamic Programming* method [Rotting and Gjelsvik (1992)] is widely used for compagnies having large stocks of water to manage. When the company portfolio is composed of many stocks of water and many power plants a decomposition method can be used [Culioli and Cohen (1990)] and the bundle method may be used for coordination [Bacaud et al. (2001)]. The uncertainty is usually modeled with trees [Heitsch and Romisch (2003)].
In realistic modelization of the previous problem, the convexity is not assured. The constraints may be non linear as for gaz storage for example where injection and withdrawal capacities depend on the position in the stock (and for thermodynamic reason depends on the past controls in accurate model). Most of the time, the problem is not continuous and is in fact a mixed integer stochastic problem : the commands associated to a stock of water can only take some discrete values due to the fact that a turbine has only on-off positions, financial positions are taken for discrete number of stocks... If the constraints and the objectif function are linearized, the stochastic problem can be discretized on the tree and a mixed integer programming solver can be used. In order to be able to use this kind of modelization a non recombining tree has to be build. The explosion of the number of leaves of the tree leads to a huge mixed integer problem to solve.
Therefore when the constraints are non linear or when the problem is non convex, the dynamic programming method developed in 1957 [Bellman (1957)] may be the most attractive. This simple approach faces one flaw: it is an enumerative method and the computational cost goes up exponentially with the number of state variable to manage. This approach is currently used for a number of state variable below 5 or 6. This article introduces the parallelization scheme developed to implement the dynamic programming method, details some improvements required to run large benchmarks on large scale architectures, and presents the serial optimizations achieved to efficiently run on each node of a PC cluster and an IBM Blue Gene/P supercomputer. This approach allows us to tackle a simplified problem with 3 random factors to face and 7 stocks to manage.

2. Stochastic control optimization and simulation

We give a simplified view of a stochastic control optimization problem. Supposing that the problem we propose to solve can be set as :

$$\min_{nc_t} \quad \mathbb{E}\left(\sum_{t=1}^{N} \phi(t, \xi_t, nc_t)\right) \tag{1}$$

where ϕ is a cost function depending on time, the state variable ξ_t (stock and uncertainty,) and depending on the command nc_t realized at date t. For simplicity, we suppose that the control only acts on the deterministic stocks and that the uncertainties are uncontrolled. Some additional constraints are added defining at date t the possible commands nc_t depending on ξ_t.
The software used to manage the energy assets are usually separated into two parts. A first software, an optimization solver is used to calculate the so-called Bellman value until maturity T. The second one will test the Bellman values calculated during the first software run on some scenarios.

2.1 Optimization part

In our implementation of the Bellman method, we store the Bellman values J at a given time step t, for a given uncertainty factor occurring at time t and for some stocks levels. These Bellman values represent the expected gains remaining for the optimal asset management from the date t until the date T starting optimization with a given state. Instead of using usual non recombining trees, we have chosen to use Monte Carlo scenarios to achieve our optimization following [Longstaff and Schwartz (2001)] methodology. The uncertainties are here simplified so that the model is Markovian. The number of scenarios used during this part is rather small (less than a thousand). This part is by far the most time consuming. The algorithm 1 gives the Bellman values J for each time step t calculated by backward recursion. In the algorithm 1, due to the Markovian property of the uncertainty, $s^* = f(s,w)$ is a realization at date $t + \Delta t$ of an uncertainty whose value is equal to s at date t, where w is a random factor independent on s.

```
For $t := (nbstep - 1)\Delta t$ to $0$
    For $c \in$ admissible stock levels ($nbstate$ levels)
        For $s \in$ all uncertainty ($nbtrajectory$)
            $\tilde{J}^*(s,c) = \infty$
            For $nc \in$ all possible commands for stocks ($nbcommand$)
                $\tilde{J}^*(s,c) = \min(\tilde{J}^*(s,c), \phi(nc) + \mathbb{E}(J(t+\Delta t, s^*, c+nc)|s))$
    $J^*(t,:,:) := \tilde{J}^*$
```

Figure 1: Bellman algorithm, with a backward computation loop.

In our modelization, uncertainties are driven by brownian motions and conditional expectation in the algorithm 1 are calculated by regression methods as explained in [Longstaff and Schwartz (2001)]. Using Monte Carlo, it could have been possible to use Malliavin methods [Bouchard et al. (2004)], or it could have been possible to use a recombining quantization tree [Bally et al. (2005)].

2.2 Simulation part

A second software called a simulator is then used to accurately compute some financial indicators (VaR, EEaR, expected gains on some given periods). The optimization part only gives the Bellman values in each possible state of the system. In the simulation part, the uncertainties are accurately described with using many scenarios (many tens of thousand) to accurately test the previously calculated Bellman values. Besides, the modelization in the optimizer is often a simplified one so that calculation are made possible by a reduction in the number of state variable. In the simulator it is often much more easier to deal with far more complicated constraints so that the modelization is more realistic. In the simulator, all the simulations can be achieved in parallel, so we could think that this part is embarrassingly parallel as shown by algorithm 2. However, we will see in the sequel that the parallelization scheme used during the optimization will bring some difficulties during simulations that will lead to some parallelization task to achieve.

3. Distributed algorithm

Our goal was to develop a distributed application efficiently running both on large PC cluster (using Linux and classic NFS) and on IBM Blue Gene supercomputers. To achieve this goal, we have designed some main mechanisms and sub-algorithms to manage data distribution,load

```
stock(1:$nbtrajectory$) = initialStock
For $t := 0$ $to$ $(nbstep - 1)\Delta t$
    For $s \in$ all uncertainty ($nbtrajectory$)
        Gain = - $\infty$
        For $nc \in$ all possible commands for stocks ($nbcommand$)
                $GainA = phi(nc) + \mathbb{E}\left(J^*(t + \Delta t, s^*, stock(s) + nc)|s\right)$
                if $GainA > Gain$
                    com = nc
                    Gain = GainA
        stock(s) += com
```

Figure 2: Simulation on some scenarios.

balancing, data routage planning, data routage execution, and file accesses. Next sections introduce our parallelization strategy, detail the most important issues and describe our global distributed algorithm.

3.1 Parallelization overview of the optimization part

As explained in section 2 we use a backward loop to achieve the optimization part of our stochastic control application. This backward loop is applied to calculate the Bellman values at discrete points belonging to a set of N stocks, which form some N-dimensional cube of data, or *data N-cubes*.

Considering one stock X, its stock levels at t_n and t_{n+1} are linked by the equation:

$$X_{n+1} = X_n + Command_n + Supply_n \tag{2}$$

Where:

- X_n and X_{n+1} are possible levels of the X stock, and belong to intervals of possible values ($[X_n^{min}; X_n^{max}]$ and $[X_{n+1}^{min}; X_{n+1}^{max}]$), function of scenarios and physical constraints.
- The *Command* is the change of stock level due to the execution of a *command* on the stock X between t_n and t_{n+1}. It belongs to an interval of values: $[C_n^{min}; C_n^{max}]$, function of scenarios and physical constraints.
- The $Supply_n$ is the change of stock level due to an external supply (in our test case with hydraulic energy stocks, snow melting and rain represent this supply). Again, it belongs to an interval of values: $[S_n^{min}; S_n^{max}]$, function of scenarios and physical constraints.

Considering the equation 2, the backward loop algorithm introduced in section 2, a set of scenarios and physical constraints, and N stocks, the following 6 sub-steps algorithm is run on each computing node at each time step :

1. When finishing the t_{n+1} computing step and entering t_n one (backward loop), minimal and maximal stock levels of all stocks are computed on each computing node, according to scenarios and physical constraints on each stock. So, each node easily computes N minimal and maximal stock levels that defines the minimal and maximal vertexes of the *N-cube* of points where the Bellman values have to be calculated at date t_n.
2. Each node runs its splitting algorithm of the t_n *N-cube* to distribute the t_n Bellman values that will be to computed at step t_n on $P = 2^{d_p}$ computing nodes. Each node computes

the entire map of this distribution: the t_n *data map*. See section 3.4 for details about the splitting algorithm.

3. Using scenarios and physical constraints set for the application, each node computes the *Commands* and *Supplies* to apply to each stock of each t_n N-subcube of the t_n *data map*. Using equation 2 each node computes the t_{n+1} N-subcube of points where the t_{n+1} Bellman values are required by each node to process the calculation of the Bellman values at the stocks points belonging to its t_n N-subcube. So, each node easily computes the coordinates of the P' t_{n+1} *data influence areas* or t_n *shadow regions*, and builds the entire t_n *shadow region map* without any communication with others nodes. This solution has appeared faster than to compute only local data partition and local shadow region on each node and to exchange messages on the communication network to gather the complete maps on each node.
4. Now each node has the entire t_n *shadow region map* computed at the previous sub-step, and the entire t_{n+1} *data map* computed at the previous time step of the backward loop. Some basic computations of N-cube intersections allow each node to compute the P N-subcubes of points associated to the Bellman values to receive from others nodes and from itself, and the P N-subcubes of points associated to the Bellman values to send to other nodes. Some of these N-subcubes can be empty and have a null size, when some nodes have no N-subcubes of data at time step t_n or t_{n+1}. So, each node builds its t_n *routing plan*, still without any communications with other nodes. See section 3.5 for details about the computation of this routing plan.
5. Using MPI communication routines, each node executes its t_n *routing plan* and brings back the Bellman values associated to points belonging to its t_{n+1} *shadow region* in its local memory. Function of the underlying interconnection network and the machine size, it can be interesting to overlap all communications, or it can be necessary to spread the numerous communications and to achieve several communication sub-steps. See section 3.6 for details about the routing plan execution.
6. Using the t_{n+1} Bellman brought back in its memory, each node can achieve the computation of the optimal commands for all stock points (according to the stochastic control algorithm) and calculate its t_n Bellman value.
7. Then, each node save on disk the t_n Bellman values and some others step results that will be used in the *simulation part* of the application. They are temporary results stored on local disks when exist, or in global storage area, depending of the underlying parallel architecture. Finally, each node cancels its t_{n+1} *data map*, t_n *shadow region map* and t_n *routing plan*. Only its t_n *data map* and t_n *data N-subcube* have to remain to process the following time step.

This time step algorithm is repeated in the backward loop up to time step 0. Then some global results are saved, and the simulation part of the application is run.

3.2 Parallelization overview of the simulation part

In usual sequential software, simulations is achieved scenario by scenario: the stock levels and the commands are calculated from date 0 to date T for each scenario sequentially. This approach is obviously easy to parallelize when the Bellman values are shared by each node. In our case, doing so will mean a lot of time spent in IO. In the algorithm 2, it has been chosen to advance time step by time step and to do the calculation at each time step for all simulations.

So Bellman temporary files stored in the optimization part are opened and closed only once by time step to read Bellman values of the next time step.
Similarly to the optimization part, at each time step t_n the following algorithm is achieved by each computing node:

1. Each computing node reads some temporary files of optimization results: the t_{n+1} *data map* and the t_{n+1} *data* (Bellman values). All these reading operations are achieved in parallel from the P computing nodes.
2. For each trajectory (up to the number of trajectories managed by each node):
    (a) Each node simulates the hazard trajectory from time step t_n to time step t_{n+1}.
    (b) From the current N dimensional stock point SP_n, using equation 2, each node computes the t_{n+1} N-subcube of points where the t_{n+1} Bellman values are required to process the calculation of the optimal command at SP_n: the t_n *shadow region* coordinates of the current trajectory.
    (c) All nodes exchange their t_n *shadow region* coordinates using MPI communication routines and achieving a *all_gather* communication scheme. So, each node can build a complete t_{n+1} *shadow region map* in its local memory. In the *optimization* part each node could compute the entire t_{n+1} *shadow region map*, but in the *simulation* part inter-node communications are mandatory.
    (d) Each node computes its *routing plan*, computing N-subcubes intersections of t_{n+1} *data map* and t_{n+1} *shadow region map*. We apply again the 2-step algorithm described on figure 6 and used in the *optimization* part.
    (e) Each node executes its *routing plan* using MPI communication routines, and and brings back the Bellman values associated to points belonging to its t_{n+1} *shadow region* in its local memory. Like in the *optimization* part, depending on the underlying interconnection network and the machine size, it can be interesting to overlap all communications, or it can be necessary to spread the numerous communications and to achieve several communication sub-steps (see section 3.6).
    (f) Using the t_{n+1} Bellman value brought back in its memory, each node can compute the optimal command according to algorithm introduced on figure 2.
3. If required by user, data of the current time step are gathered on computing node 0, and written on disk (see section 3.7).

Finally, some complete results are computed and saved by node 0, like the global gain computed by the entire application.

3.3 Global distributed algorithm

Figure 3 summarizes the main three parts of our complete algorithm to compute optimal commands of a N dimensional optimization problem and to test them in simulation. The first part is the reading of input data files according to the IO strategy introduced in section 3.7. The second part is the *optimization solver* execution, computing some Bellman values in a backward loop (see sections 1 and 3.1). At each step, a N-cube of Bellman values to compute is split on an hypercube of computing nodes to load balance the computations, a shadow region is identified and gathered on each node, some multithreaded local computations of optimal commands are achieved for each point of the N-cube (see section 4.3), and some temporary results are stored on disk. Then, the third part tests the previously computed commands.

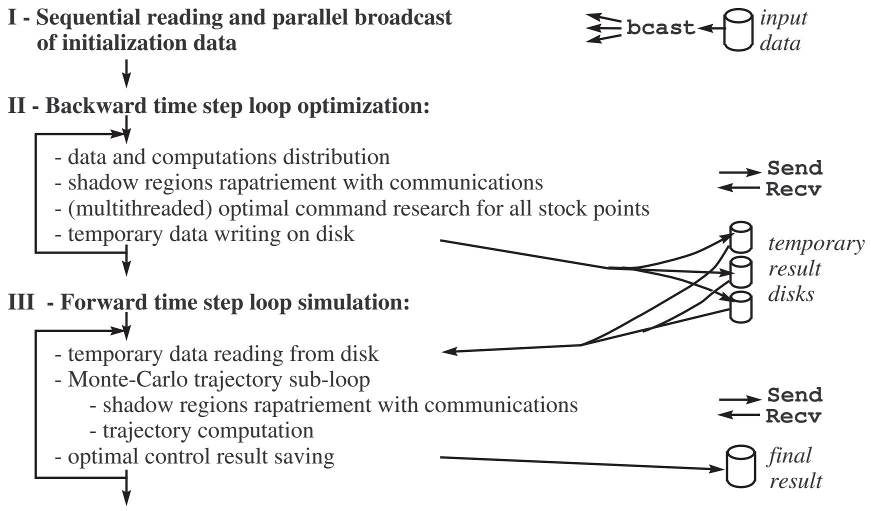
I - Sequential reading and parallel broadcast of initialization data
bcast
input data
II - Backward time step loop optimization:
- data and computations distribution
- shadow regions rapatriement with communications
- (multithreaded) optimal command research for all stock points
- temporary data writing on disk
Send
Recv
temporary result disks
III - Forward time step loop simulation:
- temporary data reading from disk
- Monte-Carlo trajectory sub-loop
- shadow regions rapatriement with communications
- trajectory computation
- optimal control result saving
Send
Recv
final result

Figure 3: Main steps of the complete application algorithm.

This *simulation* part runs a forward time step loop (see sections 1 and 3.2) and a Monte-Carlo trajectory sub-loop (see section 4.3), and uses the same previous mechanisms than the second part. At each time step, each node reads some temporary results it has stored during the *optimization* part, gather some shadow regions, achieves some local computations, and stores the final results on disk.

3.4 Data N-cube splitting and data map setting

During the backward computation loop of the Bellman algorithm (see figure 1) of the *optimization part* of our application, we need to compute a N-dimensional cube of Bellman values at each time step. This computation is long and requires a large amount of memory to store the N-cube data. So we have to *split* this N-cube data on a set of computing nodes both to speedup (using more processors) and to size up (using more memory). Moreover, each dimension of this N-cube represents the *stock levels* of one *stock* that can change from time step t_{n+1} to t_n. Each stock level range can be translated, and/or enlarged or shrunk. So, we have to redistribute our problem at each time step: we have to *split* a new N-cube of stock point when entering a new time step. Our N-cube splitting algorithm is a critical component of our distributed application that must run quickly. During the forward loop of the *simulation part* we reread on disk the maps stored during optimization.

The computation of one Bellman value at one point of the t_n N-cube requires the *influence area* of this value given by equation 2 : the t_{n+1} Bellman values at stocks points belonging to a small sub-cube of the t_{n+1} N-cube. Computation of the entire t_n N-sub-cube attached to one computing node requires an *influence area* that can be a large *shadow region*, leading to MPI communication of Bellman values stored on many other computing nodes (see figure 4). To minimize the size of this N-dimensional *shadow region* we favor *cubic* N-sub-cubes in place of *flat* ones. So, we aim to achieve *cubic* split of the N-cube data at each time step.

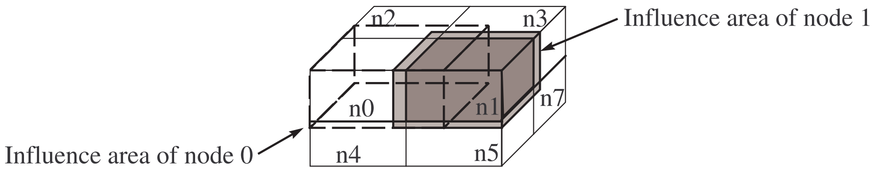

Figure 4: Example of *cubic* split of the N-Cube of data and computations

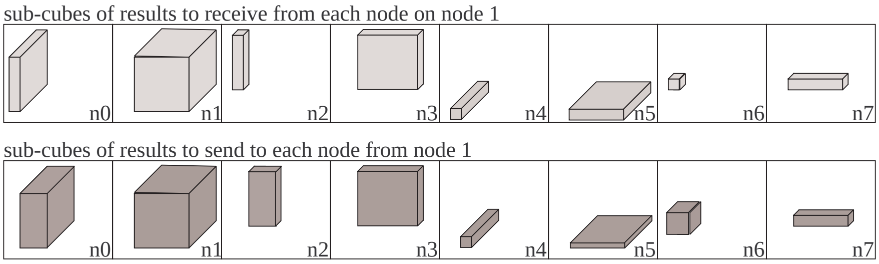
sub-cubes of results to receive from each node on node 1
n0
n1
n2
n3
n4
n5
n6
n7
sub-cubes of results to send to each node from node 1
n0
n1
n2
n3
n4
n5
n6
n7

Figure 5: Example of routing plan established on node 1

We decided to split our N-cube data on $P_{max} = 2^{d_{max}}$ computing nodes. We successively split in two equal parts some dimensions of the N-cube, up to obtain $2^{d_{max}}$ sub-cubes, or to have reach the limits of the division of the N-cube. Our algorithm includes 3 sub-steps:

1. We split the dimensions of the N-cube in order to obtain sub-cubes with close dimension sizes. We start to sort the N' divisible dimensions in decreasing order, and attempt to split the first one in 2 equal parts with sizes close to size of the second dimension. Then we attempt to split again the size of the 2 first dimensions to reduce their sizes close to the size of the third one. This splitting operation fails if it leads to a sub-cube dimension size smaller than a *minimal size*, set to avoid to process too small data sub-cubes. The splitting operation is repeated up to achieve d_{max} splits, or up to reduce the sizes of the $N' - 1$ first dimensions close to the size of the smallest one. Then, if we have not obtained $2^{d_{max}}$ sub-cubes we run the second sub-step.
2. Previously we have obtained sub-cubes with N' close dimension sizes. Now we sort these N' divisible dimensions in decreasing order, considering their split dimension sizes. We attempt to split again in 2 equal parts each divisible dimension in a round robin way, up to achieve d_{max} splits, or up to reach the limit of the *minimal size* for each divisible dimension. Then, if we have not obtained $2^{d_{max}}$ sub-cubes we run the third sub-step.
3. If it is specified to *exceed* the *minimal size* limit, then we split again in 2 equal parts each divisible dimension in a round robin way, up to achieve d_{max} splits, or up to reach dimension sizes equal to 1. In our application, the *minimal size* value is set before to split a N-cube, and a command line option allows the user to respect or to exceed this limit. So, when processing small problems on large numbers of computing nodes, some

```
// Input variable datatypes and declarations on node Me
N-cube-coord_t $DataMap_{t_{n+1}}[P]$
N-cube-coord_t $ShadowRegion_{t_n}[P]$
// Output variable datatypes and declarations on node Me
N-cube-coord_t $LocalRoutingPlan_{t_n}^{Recv}[P]$
N-cube-coord_t $LocalRoutingPlan_{t_n}^{Send}[P]$

// Coordinates computation of the N-subcubes to receive on node $Me$ from all nodes
For $i := 0$ to $(P-1)$
    $LocalRoutingPlan_{t_n}^{Recv}[i] := DataMap_{t_{n+1}}[i] \cap ShadowRegionMap_{t_n}[Me]$

// Coordinates computation of the N-subcubes to send to all nodes from node $Me$
For $i := 0$ to $(P-1)$
    $LocalRoutingPlan_{t_n}^{Send}[i] := DataMap_{t_{n+1}}[Me] \cap ShadowRegionMap_{t_n}[i]$
```

Figure 6: Computation of t_n routing plan on computing node Me ($0 \leq Me < P$)

experiments are required and can be rapidly conducted to point out the right tuning of our splitting algorithm.

Finally, after running our splitting algorithm at time step t_n we obtain 2^d sub-cubes, and we can give data and work up to $P = 2^d$ computing nodes. When processing small problems on large parallel machines, it is possible not all computing nodes will have some computations to achieve at time step t_n ($P < P_{max}$) (a too fine grained data distribution would lead to inefficient parallelization). This splitting algorithm is run on each computing node at the beginning of time step t_n. They all compute the same N-cube splitting and deduce the same number of provisioned nodes P and the same *data map*.

3.5 Shadow region map and routing plan computations

Different data N-subcubes located on different nodes, or existing at different time steps, can have *shadow regions* with different sizes. Moreover, depending on the time step, the problem size and the number of used computing nodes, the *shadow region* N-subcube of one computing node can reach only its direct neighbors or can encompass these nodes. So, the exact routing plan of each node has to be dynamically established at each time step before to retrieve data from other nodes.

As explained in section 3.1, each node computes the entire *shadow region map*: a table of P coordinates of N-subcubes. In our application these entire maps can be deduced from t_n and t_{n+1} *data maps*, and from scenarios and physical constraints on commands and supplies of each stock. For example, node 1 on figure 4 knows its *shadow region* (light gray cube) in this 3-cube, the *shadow region* of node 0 (dotted line) and of nodes 2 to 7 (not drawn on figure 4).

Then, using both its t_n *shadow region map* and its t_{n+1} *data maps*, each computing node can easily compute its local t_n *routing plan* in two sub-steps:

1. Each node computes the N-subcubes of Bellman values it has to receive from other nodes: the *receive part* of its local *routing plan*. The intersection of the t_n *shadow region* N-subcube of node Me with the t_{n+1} N-subcube of another node gives the t_{n+1} N-subcube of Bellman values the node Me has to receive from this node. So, each node achieve the first loop of the algorithm described on figure 6, and computes P intersections of N-subcubes coordinates, to get the coordinates of the P N-subcube of Bellman values it

has to receive. When the *shadow regions* are not too large, many of these P N-subcubes are empty.

2. Each node computes the N-subcubes of Bellman values it has to send to other nodes: the *send part* of its local *routing plan*. The intersection of the t_{n+1} N-subcube of node *Me* with the t_n *shadow region* N-subcube of another node gives the t_{n+1} N-subcube of Bellman values the node *Me* has to send to this node. So, each node achieve the second loop of the algorithm described on figure 6, and computes P intersections of N-subcubes coordinates, to get the coordinates of the P N-subcube of Bellman values it has to send. Again, many of these N-subcubes are empty when the *shadow regions* are not too large.

Figure 5 shows an example of local *routing plan* computed on node 1, considering the data distribution partially illustrated on figure 4. This entire routing plan computation consists in $2.P$ intersections of N-subcube coordinates. Finally, this is a very fast integer computation, run at each time step.

3.6 Routing plan execution

Node communications are implemented with non-blocking communications and are overlapped, in order to use the maximal abilities of the interconnection network. However, for large number of nodes we can get small sub-cubes of data on each node, and the influence areas can reach many nodes (not only direct neighbor nodes). Then, the routing plan execution achieves a huge number of communications, and some node interconnexion network could saturate and slow down. So, we have parameterized the routing plan execution with the number of nodes that a node can attempt to contact simultaneously. This mechanism spreads the execution of the communication plan, and the spreading out is controlled by two application options (specified on the command line): one for the *optimization* part, and one for the *simulation* part.
When running our benchmark (see section 5) on our 256 dual-core PC cluster it has been faster not to spread these communications, but on our 8192 quad-core Blue Gene/P it has been really faster to spread the communications of the *simulation* part. Each Blue Gene node has to contact only 128 or 256 other nodes at the same time, to prevent the simulation time to double. When running larger benchmarks (closer to future real case experiments), the size of the data and of the *shadow regions* could increase. Moreover, each *shadow region* could spread on a little bit more nodes. So, the total size and number of communications could increase, and it seems necessary to be able to temporally spread both communications of *optimization* and *simulation* parts, on both our PC-cluster and our Blue Gene/P supercomputer.
So, we have maintained our communication spreading strategy. When running the application, an option on the command line allows to limit the number of simultaneous asynchronous communications a computing node can start. If a saturation of the communication system appears, it is possible to use it sparingly, spreading the communications.

3.7 File IO constraints and adopted solutions

Our application deals with input data files, temporary output and input files, and final result files. These files can be large, and our main target systems have very different file access mechanisms. Computing nodes of IBM Blue Gene supercomputers do not have local disks, but an efficient parallel file system and hardware allows all nodes to concurrently access a global remote disk storage. At the opposite, nodes of our Linux PC cluster have local disks but use basic Linux NFS mechanisms to access global remote disks. All nodes of our cluster

can not make their disk accesses at the same time. When increasing the number of used nodes, IO execution times become longer, and finally they freeze.
Temporary files are written and read at each time step. However, each temporary result file is written during the *optimization* part by only one node, and is read during the *simulation* part by only the same node. These files do not require concurrent accesses and their management is easy. Depending on their path specified on the command line when running the application, they are stored on local disks (fastest solution on PC cluster), or on a remote global disk (IBM Blue Gene solution). When using a unique global disk it is possible to store some temporary index files only once, to reduce the total amount of data stored.
Input data files are read only once at the beginning, but have to be read by each computing node. Final result files are written at each time step of the *simulation* part and have to store data from each computing node. In both cases, we have favored the genericity of the file access mechanism: node 0 opens, accesses and closes files, and sends data to or receives data from other nodes across the interconnection network (using MPI communication routines). This IO strategy is an old one and is not always the most efficient, but is highly portable. It has been implemented in the first version of our distributed application. A new strategy, relying on a *Parallel File System* and an efficient hardware, will be designed in future versions.

4. Parallel and distributed implementation issues

4.1 N-cube implementation

Our implementation includes 3 main kinds of arrays: MPI communication buffers, *N-cube data maps* and *N-cube data*. We have used classic dynamic C arrays to implement the first kind, and the `blitz++` generic C++ library [Veldhuizen (2001)] to implement the second and third kinds. However, in order to compile the same source code independently of the number of energy stocks to process, we have flattened the N-cubes required by our algorithms. Any N-dimensional array of stock point values becomes a one dimensional array of values.
Our implementation includes the following kind of variables:

- A *stock level range* is a one dimensional array of 2 values, implemented with a `blitz::TinyVector` of 2 integer values.
- The coordinates of a *N-cube of stock points* is an array of *N stock level ranges*, implemented with a one dimensional `blitz::Array` of *N* `blitz::TinyVector` of 2 integer values.
- A *map* of *N-cube data* is implemented with a two dimensional array of $P \times N$ *stock level ranges*. It is implemented with a two dimensional `blitz::Array` of `blitz::TinyVector`).
- A *Bellman value* is depending on the *stock point* considered and on the *alea* considered. Our *N-cube data* are arrays of Bellman values function of different *aleas* in a N-cube of *stock points*. A *N-cube data* is implemented with a two dimensional `blitz::Array` of `double`: the first dimension index is the flattened *N* dimensional coordinate of the stock point, and the second dimension index is the *alea* index.
- Some one dimensional arrays of `double` are used to store data to send to or to receive from another node, and some two dimensional arrays of `double` are used to store data to send to or to receive from all computing nodes. Communications are implemented with the MPI library and its C API, that was available on all our testbed architectures. This API requires addresses of contiguous memory areas, to read data to send or to store received data. So, classic C dynamic arrays appeared a nice solution to implement communication buffers with sizes updated at each time step.

Finally, `blitz` access mechanism to `blitz` array elements appeared slow. So, inside the computing loop we prefer to get the address of the first element to access using a `blitz` function, and to access the next elements incrementing a pointer like it is possible for a classic C array.

4.2 MPI communications

Our distributed application consists in loops of local computations and internode communications, and communications have to be achieved before to run the next local computations. So, we do not attempt to overlap computations and communications. However, in a communication step each node can exchange messages with many others, so it is important to attempt to overlap all message exchanges and to avoid to serialize these exchanges.
When routing the Bellman values of the *shadow region* the communication schemes can be different on each node and at each time step (see sub-steps 5 of section 3.1 and 2.e of section 3.2), and data to send is not contiguous in memory. So, we have not used collective communications (easier to use with regular communication schemes), but asynchronous MPI point-to-point communication routines. Our communication sub-algorithm is the following:

- compute the size of each message to send or to receive,
- allocate message buffers, for messages to send and to receive,
- make local copy of data to send in the corresponding send buffers,
- start all asynchronous MPI point-to-point *receive* and *send* operations,
- wait until all *receive* operations have completed (synchronization operation),
- store received data in the corresponding application variables (`blitz++` arrays),
- wait until all *send* operations have completed (synchronization operation),
- delete all communication buffers.

As we have chosen to fill *explicit communication buffers* to store data to exchange, we have used *in place* asynchronous communication routines to exchange these buffers (avoiding to re-copy data in other buffers with buffered communications). We have used `MPI_Irecv`, and `MPI_Isend` or `MPI_Issend`, depending on the architecture and MPI library used. The `MPI_Isend` routines is usually faster but has a non standard behavior, function of the MPI library and architecture used. The `MPI_Issend` is a little bit longer but has a standardized behavior. On Linux PC clusters where different MPI libraries are installed, we use `MPI_Issend / MPI_Irecv` routines. On IBM Blue Gene supercomputer, with an IBM MPI library, we successfully experimented `MPI_Isend / MPI_Irecv` routines.
Internode communications required in IO operations to send initial data to each node, or to save final results on disk in each time step of *simulation part* (see sub-step 7 of section 3.1), are implemented with some collective MPI communications: `MPI_Bcast`, `MPI_Gather`.
Exchange of *local shadow region coordinates* in each time step of the *simulation part* (see sub-step 2.c of section 3.2) is implemented with a collective `MPI_Allgather` operation. All these communication have very regular schemes and can be efficiently implemented with MPI collective communication routines.

4.3 Nested loops multithreading

In order to take advantage of multi-core processors we have multithreaded, in order to create only one MPI process per node and one thread per core in place of one MPI process per core. Depending on the application and the computations achieved, this strategy can be more or less efficient. We will see in section 5.4 it leads to serious performance increase of our application.

To achieve multithreading we have split some nested loops using OpenMP standard tool or the Intel Thread Building Block library (TBB). We maintain these two multithreaded implementations to improve the portability of our code. For example, in the past we encountered some problems at execution time using OpenMP with ICC compiler, and TBB was not available on Blue Gene supercomputers. Using OpenMP or Intel TBB, we have adopted an incremental and pragmatic approach to identify the nested loops to parallelize. First, we have multithreaded the *optimization* part of our application (the most time consuming), and second we attempted to multithread the *simulation* part.
In the *optimization* part of our application we have easily multithreaded two nested loops: the first prepares data and the second computes the Bellman values (see section 2). However, only the second has a significant execution time and leads to an efficient multithreaded parallelization. A computing loop in the routing plan execution, packing some data to prepare messages, could be parallelized too. But, it would lead to seriously more complex code while this loop is only $0.15 - 0.20\%$ of the execution time on a 256 dual-core PC cluster and on several thousand nodes of a Blue Gene/P. So, we have not multithreaded this loop.
In the *simulation* part each node processes some independent Monte-Carlo trajectories, and parallelization with multithreading has to be achieved while testing the commands in the algorithm 2. But this application part is not bounded by the amount of computations, but by the amount of data to get back from other nodes and to store in the node memory, because each MC trajectory follows an unpredictable path and requires a specific *shadow region*. So, the impact of multithreading will be limited on the *simulation* part until we improve this part (see section 6).

4.4 Serial optimizations

Beyond the parallel aspects the serial optimization is a critical point to tackle the current and coming processor complexity as well as to exploit the entirely capabilities of the compilers. Three types of serial optimization were carried out to match the processor architecture and to simplify the language complexity, in order to help the compiler to generate the best binary:

1. Substitution or coupling of the main computing parts including blitz++ classes by standard C operations or basic C functions.
2. Loops unrolling with backward technique to ease SIMD or SSE (Streaming SIMD Extension for x86 processor architecture) instructions generation and optimization by the compiler while reducing the number of branches.
3. Moving local data allocations outside the parallel multithreaded sections, to minimize memory fragmentation, to reduce C++ constructor/destructor classes overhead and to control data alignment (to optimize memory bandwidth depending on the memory architecture).

Most of the data are stored and computed within `blitz++` classes. The `blitz++` streamlines the overall implementation by providing arrays operations whatever the data type. Overloading operator is one of the main inhibitor for the compilers to generate an optimal binary. To get round this inhibitor the operations including blitz classes were replaced by standard C pointers and C operations for the most time consuming routines. C pointers and operators of code C are very simple to couple with `blitz++` arrays, and whatever the processor architecture we have got a significant speedup greater than a factor 3 with this technique. See [Vezolle et al. (2009)] for more details about these optimizations.

With the current and future processors it is compulsory to generate vector instructions to reach a good ratio of the serial peak performance. $30-40\%$ of the total elapsed time of our software is spent in `while` loops including a `break` test. For a medium case the minimum number of iterations is around 100. A simple look at the assembler code shows that, whatever the level of the compiler optimization, the structure of the loop and the break test do not allow to unroll techniques and therefore to generate vector instructions. So, we have explicitly loop unrolled these `while`-and-`break` loops, with extra post-computing iterations then unrolling back to get the break point. This method enables vector instructions while reducing the number of branches.
In the shared memory parallel implementation (with Intel TBB library or OpenMP directives) each thread independently allocates local `blitz++` classes (arrays or vectors). The memory allocations are requested concurrently in the heap zone and can generate memory fragmentation as well as potential bank conflicts. In order to reduce the overhead due to memory management between the threads the main local arrays were moved outside the parallel session and indexed per the thread numbers. This optimization decreases the number of memory allocations while allowing a better control of the array alignment between the threads. Moreover, a singleton C++ class was added to `blitz++` library to synchronize the thread memory constructors/destructors and therefore minimize memory fragmentation (this feature can be deactivated depending on the operating system).

5. Experimental performances

5.1 User case introduction

We consider the situation of a power utility that has to satisfy customer load, using the power plants and one reservoir to manage. The utility equally disposes of a trading entity being able to take positions on both the spot market and futures market. We do neither consider the market complete, nor that market-depth is infinite.

5.1.1 Load and price model

The price model will be a two factor model [Clewlow and Strickland (2000)] driven by two brownian motions, and we will use a one factor model for load. In this modelization, the price future $\tilde{F}(t,T)$ corresponding to the price of one MWh seen at date t for delivery at date T evolves around an initial forward curve $\tilde{F}(T_0,T)$ and the load $D(t)$ corresponding to the demand at date t randomly evolves around an average load $D_0(t)$ depending on time. The following SDE describes our uncertainty model for the forward curve $\tilde{F}(t,T)$:

$$\frac{d\tilde{F}(t,T)}{\tilde{F}(t,T)} = \sigma_S(t)e^{-a_S(T-t)}dz_t^S + \sigma_L(t)dz_t^L, \tag{3}$$

with z_t^S and z_t^L two brownian motions, σ_i some volatility parameters.

With the following notations:

$$\begin{aligned} V(t1,t2,t3) &= \int_{t_1}^{t_2} \sigma_S(u)^2 e^{-2a_S(t_3-u)} + \sigma_L(u)^2 + 2\rho\sigma_S(u)e^{-a_S(t_3-u)}\sigma_L(u)du, \\ W_S(t_0,t) &= \int_{t_0}^{t} \sigma_S(u)e^{-a_S(t-u)}dz_u^S, \\ W_L(t_0,t) &= \int_{t_0}^{t} \sigma_L(u)dz_u^L, \end{aligned} \tag{4}$$

the integration of the previous equation gives :

$$\tilde{F}(t,T) = \tilde{F}(t_0,T)e^{-\frac{1}{2}V(t_0,t,T) + e^{-a_S(T-t)}W_S(t_0,t) + W_L(t_0,t)}. \tag{5}$$

Noting z_t^D a third brownian motion correlated to z_t^S and z_t^L , σ_D the volatility, and noting

$$\begin{aligned} V_D(t1,t2) &= \int_{t_1}^{t_2} \sigma_D(u)^2 e^{-2a_D(t_2-u)} du, \\ W_D(t_0,t) &= \int_{t_0}^{t} \sigma_D(u) e^{-a_D(t-u)} dz_u^D, \end{aligned} \tag{6}$$

the load curve follows the following equation:

$$D(t) = D_0(t)e^{-\frac{1}{2}V_D(t_0,t) + W_D(t0,t)}. \tag{7}$$

With this modelization, the spot price is defined as the limit of the future price :

$$S(t) = \lim_{T \downarrow t} \tilde{F}(t,T) \tag{8}$$

The dynamic of a financial product p for a delivery period of one month $[t_b(p), t_e(p)]$ can be approximated by :

$$\frac{dF(t,p)}{F(t,p)} = \widetilde{\sigma}_S(t,p)e^{-a_S(t_b(p)-t)} dz_t^S + \sigma_L(t) dz_t^L, \tag{9}$$

where :

$$\widetilde{\sigma}_S(t,p) = \sigma_S(t) \frac{\sum\limits_{t_i \in [t_b(p),t_e(p)]} e^{-a_S(t_i - t_b(p))}}{\sum\limits_{t_i \in [t_b(p),t_e(p)]} 1} \tag{10}$$

5.1.2 Test case

We first introduce some notation for our market products:

$\mathcal{P}(t) = \{p : t < t_b(p)\}$	all futures with delivery after t,
$L(t,p) = \{\tau : \tau < t, p \in \mathcal{P}(\tau)\}$	all time steps τ before t for which the futures product p is available on the market,
$\mathcal{P}^t = \{p : t \in [t_b(p), t_e(p)]\}$	all products in delivery at t,
$\mathcal{P} = \cup_{t\in[0,T]} \mathcal{P}(t)$	all futures products considered.

Now we can write the problem to be solved:

$$
\begin{aligned}
\min \quad & \mathbb{E}\left(\sum_{t=0}^{T}[\sum_{i=1}^{npal} c_{i,t}u_{i,t} - v_t S_t + \sum_{p\in\mathcal{P}(t)} (t_e(p)-t_b(p))(q(t,p)F(t,p)+|q(t,p)|\mathcal{B}_t)]\right) \\
s.t. \quad & D_t = \sum_{i=1}^{npal} u_{i,t} - v_t + w_t + \sum_{p\in\mathcal{P}^t}\sum_{s\in L(t,p)} q(s,p) \\
& R_{t+1} = R_t + \Delta t(-w_t + A_t) \qquad (11) \\
& R_{min} \le R_t \le R_{max} \\
& q_{p,min} \le q(s,p) \le q_{p,max} \ \ \forall s\in[0,T]\ \forall p\in\mathcal{P} \\
& y_{p,min} \le \sum_{s=0}^{\tau} q(s,p) \le y_{p,max} \ \ \forall \tau < t_b(p)\ \ \forall p\in\mathcal{P} \\
& v_{t,min} \le v_t \le v_{t,max} \\
& 0 \le u_{i,t} \le u_{i,t,max}, \qquad (12)
\end{aligned}
$$

where

- D_t is the customer Load at time t in MW
- $u_{i,t}$ is the production of unit i at time t in MW
- v_t is spot transactions in MW (counted positive for sales)
- $q(t,p)$ is the power of the futures product p bought at time t in MW
- $\mathcal{B}_t$ is the spread bid-ask in euros/MWh taking into account the illiquidity of the market: its double value is the price gap purchase/sale of one MWh
- R_t is the level of the reservoir at time t in MWh
- $S_t = F(t,t)$ is the spot price in euros/Mwh
- $F(t,p)$ is the futures price of the product p at time t in euros/MWh
- w_t is the production of the reservoir at time t in MW
- A_t are the reservoir inflows in MW
- Δt the time step in hours
- $q_{p,min}$, $q_{p,max}$ are bounds on what can be bought and sold per time step on the futures market in MW
- $y_{p,min}$, $y_{p,max}$ are the bounds on the size of the portfolio for futures product p
- R_{min}, R_{max} are (natural) bounds on the energy the reservoir can contain.

Some additional values for the initial stocks are also given, and some final values are set for the reservoir stock remaining at date T.

5.1.3 Numerical data

We consider at the begin of a month a four months optimization, where the operator can take position in the future market twice a month using month ahead futures peak and offpeak, two month ahead futures peak and off peak, and three month ahead futures base and peak. So the user has at date 0 6 future products at disposal. The number of trajectories for optimization is 400. The depth of the market for the 6 future products is set to 2000 MW for purchase and sales ($y_{p,min} = -2000$, $y_{p,max} = 2000$). Every two weeks, the company is allowed to change its position in the futures market within the limits of 1000 MW ($q_{p,min} = -1000$, $q_{p,max} = 1000$). All the commands for the futures stocks are tested from -1000 MW to 1000 MW with a step of 1000 MW. The hydraulic command is tested with a step of 1000MW. All the stocks of futures are discretized with a 1000MW step, the hydraulic reservoir is discretized with 225 steps leading to a maximum of $225 * 5^6$ points to explore for the stock state variables. The maximum number of commands tested is $5 * 3^6$ at day 30 for each point stock. This discretization is a very accurate one leading to a huge problem to solve. Notice that the number of stocks is decreasing with time. After two months, the two first future delivery periods are past so the problem becomes a 5 stocks problem. After three months , we are left with a three stocks problems and no command to test (delivery of the two last future contracts has begun). The global problem is solved with 6 steps per days, defining the reservoir strategy, and the future commands are tested every two weeks.

5.2 Testbeds introduction

We used two different parallel machines to test our application and measure its performances : a PC cluster and a supercomputer.

- Our PC cluster was a 256-node cluster of SUPELEC (from CARRI Systems company) with a total of 512 cores. Each node hosts one dual-core processor: INTEL Xeon-3075 at 2.66 GHz, with a front side bus at 1333 MHz. The two cores of each processor share 4 GB of RAM, and the interconnection network is a Gigabit Ethernet network built around a large and fast CISCO 6509 switch.
- Our supercomputer was the IBM Blue Gene/P supercomputer of EDF R&D. It provides up to 8192 nodes and a total of 32768 cores, which communicate through proprietary high-speed networks. Each node hosts one quad-core PowerPC 450 processor at 850 MHz, and the 4 cores share 2 GB of RAM.

5.3 Experimental provisioning of the computing nodes

Figure 7 shows the evolution of the number of stock points of our benchmark application, and the evolution of the number of available nodes that have some work to achieve: the number of provisioned nodes. The number of stock points defines the problem size. It can evolve at each time step of the *optimization* part and the splitting algorithm that distributes the N-cube data and the associated work has to be run at the beginning of each time step (see section 3.1). This algorithm determines the number of available nodes to use at the current time step. The number of stock points of this benchmark increases up to 3 515 625, and we can see on figure 7 the evolution of their distribution on a 256-nodes PC cluster, and on 4096 and 8192 nodes of a Blue Gene supercomputer. Excepted at time step 0 that has only one stock point, it has been possible to use the 256 nodes of our PC cluster at each time step. But it has not been possible to achieve this efficiency on the Blue Gene. We succeeded to use up to 8192 nodes of this architecture, but sometimes we used only 2048 or 512 nodes.

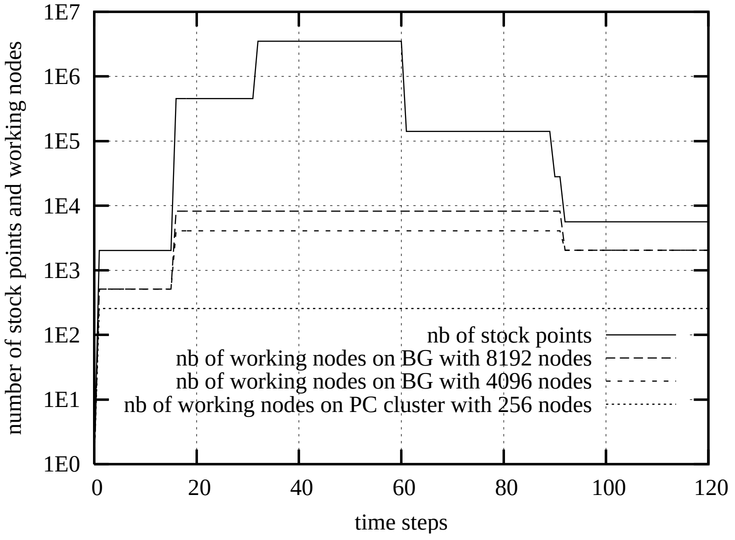

Figure 7: Evolution of the number of stock points (problem size) and of the number of working nodes (useful size of the machine)

However, section 5.4 will introduce the good scalability achieved by the *optimization* part of our application, both on our 256-nodes PC cluster and our 8192-nodes Blue Gene. In fact, time steps with small numbers of stock points are not the most time consuming. They do not make up a significant part of the execution time, and to use a limited number of nodes to process these time steps does not limit the performances. But it is critical to be able to use a large number of nodes to process time steps with a great amount of stock points. This dynamic load balancing and adaptation of the number of working nodes is achieved by our *splitting algorithm*, as illustrated by figure 7.
Section 3.4 introduces our splitting strategy, aiming at creating and distribute *cubic* subcubes and avoiding *flat* ones. When the backward loop of the *optimization* part leaves step 61 and enters step 60 the cube of stock points increases a lot (from 140 625 to 3 515 625 stock points) because dimensions two and five enlarge from 1 to 5 stock levels. In both steps the cube is split in 8192 subcubes, but this division evolves to take advantage of the enlargement of dimensions two and five. The following equations resume this evolution:

$$\begin{aligned} \textit{step } 61 : 140625 \textit{ stock points} &= 225 \times 1 \times 5 \times 5 \times 1 \times 5 \times 5 \textit{ stock points} && (13) \\ \textit{step } 60 : 3515625 \textit{ stock points} &= 225 \times 5 \times 5 \times 5 \times 5 \times 5 \times 5 \textit{ stock points} && (14) \end{aligned}$$

$$\begin{aligned} \textit{step } 61 : 8192 \textit{ subcubes} &= 128 \times 1 \times 4 \times 4 \times 1 \times 2 \times 2 \textit{ subcubes} && (15) \\ \textit{step } 60 : 8192 \textit{ subcubes} &= 128 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \textit{ subcubes} && (16) \end{aligned}$$

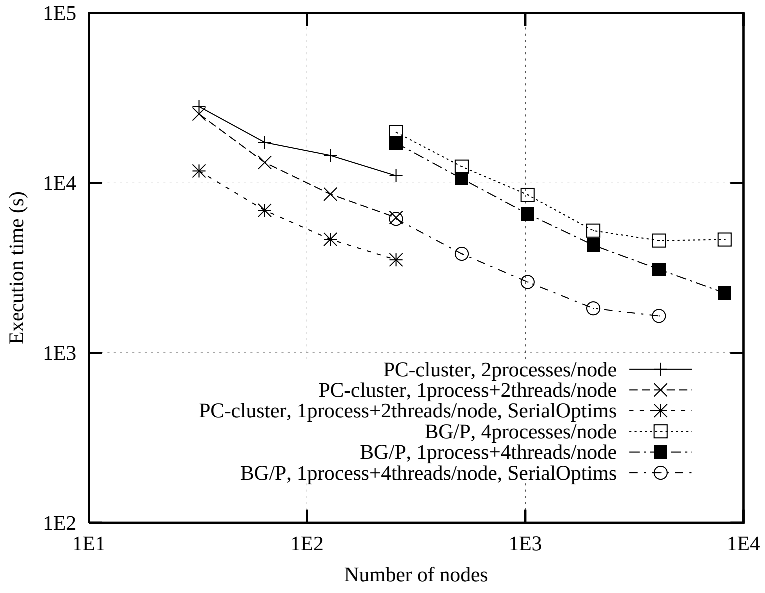

Figure 8: Total execution times function of the deployment and optimization mechanisms

$$\begin{aligned}
subcube\ sizes &= \begin{bmatrix} min\ nb\ of\ stock\ levels \\ max\ nb\ of\ stock\ levels \end{bmatrix}_{dim1} \times \begin{bmatrix} \\ \end{bmatrix}_{dim2} \dots \\
step\ 61 : subcube\ sizes &= \begin{bmatrix} 1 \\ 2 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad (17) \\
step\ 60 : subcube\ sizes &= \begin{bmatrix} 1 \\ 2 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad (18)
\end{aligned}$$

At time step 61, equations 13 and 15 show the dimension one has a stock level range of size 225 split in 128 subranges. This leads to subcubes with 1 (min) or 2 (max) stock levels in dimension one on the different nodes, as summarized by equation 17. Similarly, the dimension two has a stock level range of size 1 split in 1 subrange of size 1, the dimension three has a stock level range of 5 split in 4 subranges of size 1 or 2. . . At time step 60, equations 14 and 16 show the range of dimensions two and five enlarge from 1 to 5 stock levels and their division increases from 1 to 2 subparts, while the division of dimensions three and four decreases from 4 to 2 subparts. Finally, equation 18 shows the 8192 subcubes are more *cubic*: they have similar minimal and maximal sizes in their last six dimensions and only their first dimension can have a smaller size. This kind of data re-distribution can happen each time the global N-cube of data evolves, even if the number of provisioned nodes remains unchanged, in order to optimize the computation load balancing and the communication amount.

5.4 Performances function of deployment and optimization mechanisms

Figure 8 shows the different total execution times on the two testbeds introduced in section 5.2 for the following parallelizations:

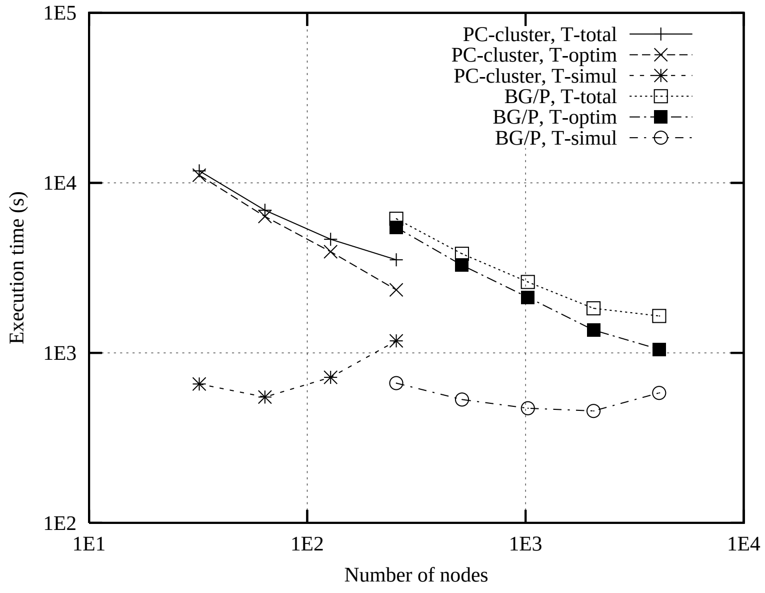

Figure 9: Details of the best execution times of the application

- implementing no serial optimization and using no thread but running several MPI processes per node (one MPI process per core),
- implementing no serial optimization but using multithreading (one MPI process per node and one thread per core),
- implementing serial optimizations and multithreading (one MPI process per node and one thread per core).

Without multithreading the execution time decreases slowly on the PC-cluster or reaches an asymptote on the Blue Gene/P. When using multithreading the execution time is smaller and decreases regularly up to 256 nodes and 512 cores on PC cluster, and up to 8192 nodes and 32768 cores on Blue Gene/P. So, the *deployment* strategy has a large impact on performances of our application. Performance curves of figure 8 show we have to deploy only one MPI process per node and to run threads to efficiently use the different cores of each node. The multithreading development introduced in section 4.3 has been easy to achieve (parallelizing only some nested loops), and has reduced the execution time and extended the scalability of the application.
These results confirm some previous experiments achieved on our PC cluster and on the Blue Gene/L of EDF without serial optimizations. Multithreading was not available on the Blue Gene/L. Using all cores of each nodes decreased the execution time but did not allowed to reach a good scalability on our Blue Gene/L [(Vialle et al.; 2008)].
Serial optimizations introduced in section 4.4 have also an important impact on the performances. We can see on figure 8 they divide the execution time by a factor 1.63 to 2.14 on the PC cluster of SUPELEC, and by a factor 1.88 to 2.79 on the Blue Gene/P supercomputer of EDF (depending on the number of used nodes). Moreover, they lead to reach the scalability

limit of our distributed application: the execution time decreases but reaches a new asymptote when using 4096 nodes and 16384 cores on our Blue Gene/P. We can not speedup more this benchmark application with our current algorithms and implementation.
These experiments have allowed to identify the right deployment strategy (running one MPI process per node and multithreading) and the right implementation (using all our serial optimizations). We analyze our best performances in the next section.

5.5 Detailed best performances of the application and its subparts

Figure 9 shows the details of the best execution times (using multithreading and implementing serial optimizations). First, we can observe the *optimization* part of our application scales while the *simulation* part does not speedup and limits the global performances and scaling of the application. So, our N-cube distribution strategy, our *shadow region map* and *routing plan* computations, and our *routing plan* executions appear to be efficient and not to penalize the speedup of the *optimization part*. But our distribution strategy of Monte carlo trajectories in the *simulation part* does not speedup, and limits the performances of the entire application.
Second, we observe on figure 9 our distributed and parallel algorithm, serial optimizations and portable implementation allow to run our complete application on a 7-stocks and 10-state-variables in less than 1*h* on our PC-cluster with 256 nodes and 512 cores, and in less than 30*mn* on our Blue Gene/P supercomputer used with 4096 nodes and 16384 cores. These performances allow to plan some computations we could not run before.
Finally, considering some real and industrial use cases, with bigger data set, the *optimization* part will increase more than the *simulation* part, and our implementation should scale both on our PC cluster and our Blue Gene/P. Our current distributed and parallel implementation is operational to process many of our real problems.

6. Conclusion and perspectives

Our parallel algorithm, serial optimizations and portable implementation allow to run our complete application on a 7-stocks and 10-state-variables in less than 1*h* on our PC-cluster with 256 nodes and 512 cores, and in less than 30*mn* on our Blue Gene/P supercomputer used with 4096 nodes and 16384 cores. On both testbeds, the interest of multithreading and serial optimizations have been measured and emphasized. Then, a detailed analysis has shown the *optimization* part scales while the *simulation part* reaches its limits. These current performances promise high performances for future industrial use cases where the *optimization* part will increase (achieving more computations in one time step) and will become a more significant part of the application.
However, for some high dimension problems, the communications during the *simulation* part could become predominant. We plan to modify this part by reorganizing trajectories so that trajectories with similar stocks levels are treated by the same processor. This will allow us to identify and to bring back the *shadow region* only once per processor at each time step and to decrease the number of communication needed.
Previously our paradigm has been successfully tested too on a smaller case for gaz storage [Makassikis et al. (2008)]. Currently it is used to valuate power plants facing the market prices and for different problems of asset liability management. In order to make easier the development of new stochastic control applications, we aim to develop a generic library to rapidly and efficiently distribute N dimensional cubes of data on large size architectures.

Acknowledgment

Authors thank Pascal Vezolle from IBM Deep Computing Europe for serial optimizations and fine tuning of the code, achieving sensitive speed improvement.
This research has been part of the ANR-CICG GCPMF project, and has been supported both by ANR (French National Research Agency) and by Region Lorraine.

7. References

Bacaud, L., Lemarechal, C., Renaud, A. and Sagastizabal, C. (2001). Bundle methods in stochastic optimal power management: A disaggregated approach using preconditioner, *Computational Optimization and Applications* **20**(3).

Bally, V., Pags, G. and Printems, J. (2005). A quantization method for pricing and hedging multi-dimensional american style options, *Mathematical Finance* **15**(1).

Bellman, R. E. (1957). *Dynamic Programming*, Princeton University Press, Princeton.

Bouchard, B., Ekeland, I. and Touzi, N. (2004). On the malliavin approach to monte carlo approximation of conditional expectations, *Finance and Stochastics* **8**(1): 45–71.

Charalambous, C., Djouadi, S. and Denic, S. Z. (2005). Stochastic power control for wireless networks via sde's: Probabilistic qos measures, *IEEE Transactions on Information Theory* **51**(2): 4396–4401.

Chen, Z. and Forsyth, P. (2009). Implications of a regime switching model on natural gas storage valuation and optimal operation, *Quantitative Finance* **10**: 159–176.

Clewlow, L. and Strickland, C. (2000). *Energy derivatives: Pricing and risk management*, Lacima.

Culioli, J. C. and Cohen, G. (1990). Decomposition-coordination algorithms in stochastic optimization, *SIAM Journal of Control and Optimization* **28**(6).

Heitsch, H. and Romisch, W. (2003). Scenario reduction algorithms in stochastic programming, *Computational Optimization and Applications* **24**.

Hull, J. (2008). *Options, Futures, and Other Derivatives, 7th Economy Edition*, Prentice Hall.

Longstaff, F. and Schwartz, E. (2001). Valuing american options by simulation : A simple least-squares, *Review of Financial Studies* **14**(1).

Ludkovski, M. and Carmona, R. (2010, to appear). Gas storage and supply: An optimal switching approach, *Quantitative Finance* .

Makassikis, C., Vialle, S. and Warin, X. (2008). Large scale distribution of stochastic control algorithms for financial applications, *The First International Workshop on Parallel and Distributed Computing in Finance (PdCoF08)*, Miami, USA.

Porchet, A., Touzi, N. and Warin, X. (2009). Valuation of a powerplant under production constraints and markets incompleteness, *Mathematical Methods of Operations research* **70**(1): 47–75.

Rotting, T. A. and Gjelsvik, A. (1992). Stochastic dual dynamic programming for seasonal scheduling in the norwegian power system, *Transactions on power system* **7**(1).

Veldhuizen, T. (2001). Blitz++ User's Guide, Version 1.2, http://www.oonumerics.org/blitz/manual/blitz.html.

Vezolle, P., Vialle, S. and Warin, X. (2009). Large scale experiment and optimization of a distributed stochastic control algorithm. application to energy management problems, *Workshop on Large-Scale Parallel Processing (LSPP 2009)*, Roma, Italy.

Vialle, S., Warin, X. and Mercier, P. (2008). A N-dimensional stochastic control algorithm for electricity asset management on PC cluster and Blue Gene supercomputer, *9th*

International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA08), NTNU, Trondheim, Norway.