Large Scale Experiment and Optimization of a Distributed Stochastic Control Algorithm. Application to Energy Management Problems.

Pascal. Vezolle*, Stephane Vialle†‡, Xavier Warin§
*IBM Deep Computing Europe, 34060 Montpellier, FRANCE
†SUPELEC, IMS group, 2 rue Edouard Belin, 57070 Metz, France
‡AlGorille INRIA Project Team, 615, rue du Jardin Botanique 54600 Villers-les-Nancy France, France
§EDF - R&D, OSIRIS group, 92141 Clamart, France

Abstract

Asset management for the electricity industry leads to very large stochastic optimization problem. We explain in this article how to efficiently distribute the Bellman algorithm used, re-distributing data and computations at each time step, and we examine the parallelization of a simulation algorithm usually used after this optimization part. We focus on distributed architectures with shared memory multi-core nodes, and we design a multiparadigm parallel algorithm, implemented with both MPI and multithreading mechanisms. Then we lay emphasis on the serial optimizations carried out to achieve high performances both on a dual-core PC cluster and a Blue Gene/P IBM supercomputer with quad-core nodes.

Finally, we introduce experimental results achieved on two large testbeds, running a 7-stocks and 10-state-variables benchmark, and we show the impact of multithreading and serial optimizations on our distributed application.

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 (telecommunication, railway), some exotic financial options or some power plants for some energy company. In the case of the energy industry, we try to maximize an expected revenue coming from decisions to run the assets, use some stocks of water, use some customers options in the portfolio, and buy or sell futures on the energy market, while satisfying the load curve. 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 [1]. So the problem can be seen as an optimization problem with many stocks to deal with. The *Stochastic Dual Dynamic Programming* method [2] 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 [3] and the bundle method may be used for coordination [4]. The uncertainty is usually modeled with trees [5]. In the case where we have binary decision to take and the constraints are linearized, the stochastic problem can be discretized on the tree and a mixed integer programming solver can be used.

When the problem is continuous and convex some very efficient methods can be used, but when the contraints are not linear and the problem is not convex, the dynamic programming method developed by Bellman in 1957 [6] may be the most attractive. This simple approach faces one flaw: the computational cost goes up exponentially with the number of stocks to manage. This article introduces the parallelization scheme developed to implement the dynamic programming method, and then 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.

2. Stochastic control optimization and simulation

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. These Bellman J values at a given time step t, for a given uncertainty factor occurring at time t (demand, fuel and electricity prices, precipitations, power plants outages), and for some stocks levels, represent the expected gains remaining for the optimal asset management from the date t until the date T. We have chosen to use Monte Carlo scenarios to achieve our optimization following [7] 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.

A second software called a simulator is then used to accurately compute some financial indicators (VaR, EEaR, expected gains on some given periods). The uncertainties

For $t := (nbstep - 1)\Delta t$ *to* 0
  For $c \in$ admissible stock levels ($nbstate$ levels)
    For $s \in$ all possible price and demand scenario ($nbtrajectory$)
      $\tilde{J}^*(s,c) = \infty$
      For $nc \in$ all possible commands for stocks ($nbcommand$)
        $\tilde{J}^*(s,c) = \min(\tilde{J}^*(s,c), \phi(nc) + \mathbb{E}\left(J^*(s^*, c + nc)|s\right))$
  $J^* := \tilde{J}^*$ *//Set J^* for the next time step*

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

are here accurately described with many scenarios (many tens of thousand) to accurately test the previously calculated commands. All the simulations can be achieved in parallel, so we could think that this part is embarrassingly parallel. 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. Parallel and distributed algorithm

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

3.1. File IO mechanisms

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 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 NFS mechanisms to access global remote disks. All nodes can not make their disk accesses at the same time.

Input data files are read only once at the beginning. Their access time is not critic and we have favored the genericity of the access mechanism. Node 0 reads these data and broadcasts their values to other nodes using MPI communication routines. Some temporary files are written and read frequently by each computing node. Depending on their path, 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. The final results are reduced on node 0, using MPI communication routines, and written by node 0 on a global and remote disk (archiving all computation results).

3.2. Data N-cube splitting

The application data is a N-dimensional cube of value vectors (considering N energy stocks). To load balance the computations on thousands of nodes (and ten thousands of cores), we split the N-cube of data onto an hypercube of 2^d nodes, as illustrated on figure 2. During the backward computation loop of the Bellman algorithm (see figure 1) each node processing a sub-cube of data at time step t_i requires a larger sub-cube of t_{i+1} results: its *influence area*. The N-cube split is achieved in order to get *cubic* sub-cubes of data on each node, and *cubic* influence area. This splitting strategy limits the amount of influence area stored on neighbor nodes (i.e. the number of nodes covered by one influence area), and reduces the amount of communications.

In order to implement a generic code, independent of the number of stocks, this N-cube of data is stored in a large one-dimensional Blitz++ array of value vectors, with devoted access routines converting N-dimensional coordinates into one-dimension index.

3.3. Routing plan computation and parameterized execution

Depending on the algorithm step and problem values, a sub-cube *influence area* can reach only the direct neighbor nodes or can encompass these nodes and reach more nodes. Moreover, different sub-cubes can have influence areas with different sizes. So, the exact routing plan of each node has to be dynamically established at each loop iteration before to retrieve data from the influence area and to achieve the local computations.

We adopt the following strategy: (1) each node computes and stores the complete distribution maps and its sub-cubes of data at t_{i+1} and t_i, (2) each node computes the influence area at time t_i of all nodes, (3) each node computes the intersection of its influence area at time t_i with the sub-cubes of data of other nodes at t_{i+1} and deduces the data to receive from any node, and (4) each node computes the intersection of the influence area of any node at time t_i with its sub-cube of data at time t_{i+1} and deduces the data to send to any node at t_i. Figure 3 shows an example of possible routing plan computed on node 1, according to our example of N-cube splitting (see figure 2). Due to the nature of our

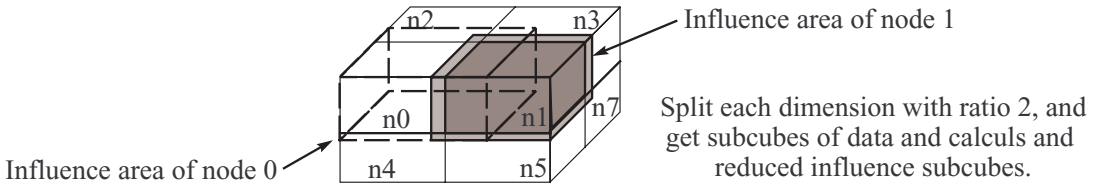

Figure 2. Example of *cubic* split of the N-Cube of data and computations

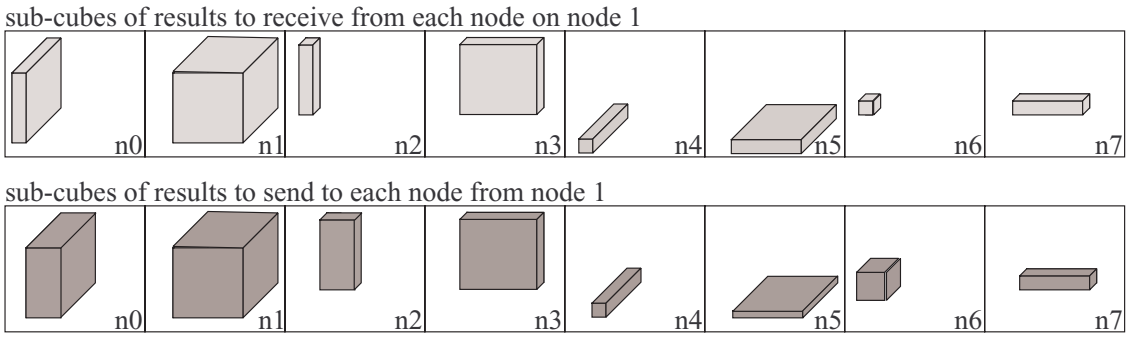

Figure 3. Example of routing plan established on node 1

problem, nodes can compute their routing plans without any communications. This approach minimizes data exchange between nodes.

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 favorite benchmark on our 256 dual-core PC cluster it is faster not to spread these communications, but on our 8192 quad-core Blue Gene/P it is 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) it appears necessary to spread both communications of *optimization* and *simulation* parts, on both our PC-cluster and our Blue Gene/P supercomputer. So, our communication spreading strategy is essential to run our application on large scale architectures.

3.4. Intra-node computation scheduling and parallelization

In order to take advantage of multi-core processors we have multithreaded and split some nested loops using OpenMP standard tool and the Intel Thread Building Block library (TBB). We maintain two multithreaded implementations to improve the portability of our code. For example, we have encountered some problems at execution time using OpenMP with ICC compiler, and TBB is currently not available on Blue Gene supercomputers. In both cases, we have adopted an incremental and pragmatic approach to identify the nested loops to parallelize.

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 theoretically it exists

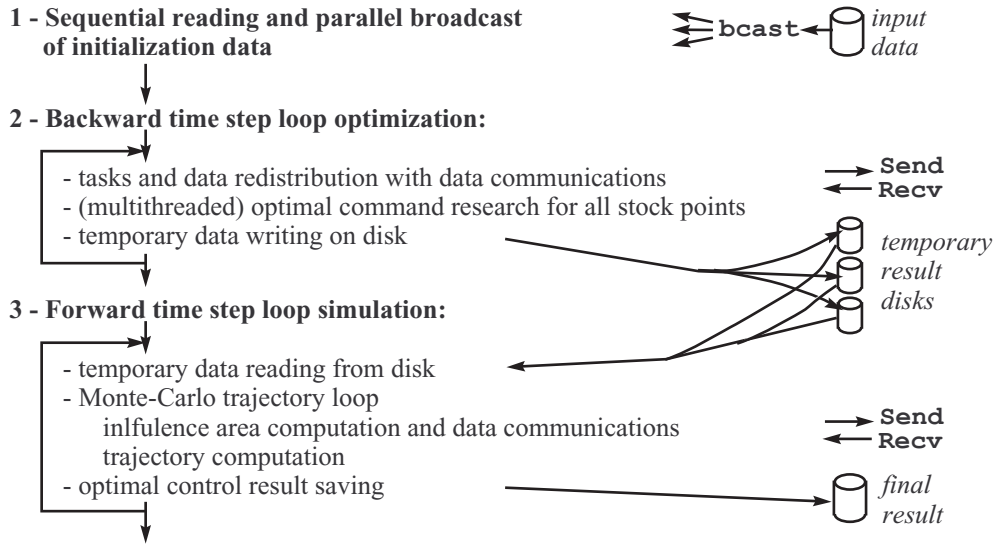

Figure 4. Main steps of the complete application algorithm.

some computation loops to parallelize. 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 *influence area*. When processing realistic large use cases, a *simulation* time step has to enter a sub-loop to get back one influence area and to process only one trajectory at each sub-step. So, we can not easily parallelize the processing of the MC trajectories of each node, and this part of the application is still monothreaded.

3.5. Global distributed algorithm

Figure 4 shows the main three parts of our complete algorithm to compute optimal commands on a set of N energy stocks, in order to optimize their global management considering the energy market and the customer demands. The first part is the reading of input data files according to the IO strategy introduced in section 3.1. The second part is the *optimization solver* execution, computing some Bellman values in a backward loop (see section 1). At each step, a N-cube of data is split on an hypercube of computing nodes to load balance the computations (see section 3.2), a devoted routing plan is computed and executed on each node (see section 3.3), some multithreaded local computations are achieved (see section 3.4), and some temporary results are stored on disk (see section 3.1). Then, the third part computes some financial indicators to test the previously computed commands. This *simulation* part runs a forward time step loop (see section 1) and a Monte-Carlo trajectory loop (see section 3.4), and uses the same previous mechanisms. At each time step, each node reads some temporary results it has stored during the *optimization* part, computes some N-dimensional sub-cube intersections to establish and execute some routing plans, achieves some local computations, and stores the final results on disk.

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).

(a) blitz++ arrays and operators: 0.082s on a BG/P core

```
array<double,1> tab0(n);
array<double,2> tab1(n,n);
array<double,1> tab2(n);

for (i=0; i<n1; i++) {
  tab0 = (double) i;
  tab1(i,range::all()) += cst;
  tab2 += cst*tab1(i,range::all());
}
```

(b) blitz++ arrays and C operators through library: 0.024s on a BG/P core

```
array<double,1> tab0(n);
array<double,2> tab1(n,n);
array<double,1> tab2(n);

for (i=0; i<n1; ++i) {
  copy_scalar_c(n,&tab0(0),i);
  add_scalar_c(n,&tab1(i,0),cst);
  daxpy(n,&tab2(0),&tab1(i,0),cst);
}
```

(c) blitz++ arrays, C pointers and operators: 0.023s on a BG/P core

```
array<double,1> tab0(n);
array<double,2> tab1(n,n);
array<double,1> tab2(n);
double *ptab0,*ptab1,*ptab2;

ptab0 = &tab0(0);
ptab2 = &tab2(0);
for (i=0; i<n1; ++i) {
  for (j=0; j<n; j++)
    ptab0[j] = (double) i;
  ptab1 = &tab1(i,0);
  for(j=0; j<n; j++)
    ptab1[j] += cst;
  for(j=0; j<n; j++)
    ptab2[j] += cst*ptab1[j];
}
```

(d) C arrays and operators: 0.0185s on a BG/P core

```
double tab0[n];
double tab1[n][n];
double tab2[n];


for (i=0; i<n1; i++) {
  for(j=0; j<n; j++)
    tab0[j] = (double) i;
  for(j=0; j<n; j++)
    tab1[i][j] += cst;
  for(j=0;j<n;j++)
    tab2[j] += cst*tab1[i][j];
}
```

Figure 5. Optimizations of 'blitz++ classes' by coupling standard C operations or basic C functions

4.1. Coupling blitz++ classes with standard C operations

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 operations for the most time consuming routines. Figure 5 presents several implementations of a simple original example using blitz++ arrays, as well as the corresponding performances for a small input on one IBM Blue Gene/P core. Compared to the original function fully implemented with blitz classes (code 5-a) a standard C function (code 5-d) is more than 4 times faster.

However, the dynamic allocation of local arrays implemented in code 5-d is not supported by all the compilers. So, we decided to use blitz++ arrays (like in code 5-a) but to replace the operations of the blitz++ classes by either external functions (code 5-b) or C pointers (code 5-c). Performances of these two solutions are very close on our problem, and the C pointers and operators of code 5-c are very simple to couple with blitz++ arrays for standard C types and do not require any library. So, we have chosen the solution 5-c, and whatever the processor architecture we have got a significant speedup greater than a factor 3 with this technique.

4.2. Loops unrolling with backward technique some 'while' loops

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 illustrated on figure 6-a. 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. An optimization consists in post-computing iterations then

(a) Original function

```
ib = 0;
while(ib < nopal) {
  tLoad -= Energie(..,ib);
  if (tLoad <= 0.0) {
    i_0 = ib;
    break;
  }
  gain(ib) -= Energie(..,ib)*Cout_Prop(..,ib);
  ++ib;
}
```

(b) Unrolling on K iterations + backward checking

```
ib = 0;
while(ib < nopal-K) {
  tLoad -= Energie(..,ib) + Energie(..,ib+1) + ..
            .. + Energie(...,ib+K-1);
  gain(ib+0) -= Energie(..,ib)*Cout_Prop(..,ib);
  gain(ib+1) -= Energie(..,ib+1)*Cout_Prop(..,ib+1);
  ...
  gain(ib+K-1) -= Energie(..,ib+K)*Cout_Prop(..,ib+K-1);
  if (tLoad <= 0.0) {
    for (ik=K-1; ik>=0; ik--) {
      tLoad += Energie(..,ib+ik);
      gain(ib+ik) +=Energie(..,ib+ik)*Cout_Prop(..,ib+ik);
      if (tLoad > 0.0)  break;
    }
    tLoad -= Energie(..,ib+ik);
    i_0 = ib+ik;
    break;
  }
  ib +=K;
}
// Process remaining iterations from ib to nopal if ib >= nopal-K
...
```

Figure 6. Optimization of a 'while' loop using loop unrolling with back technique

unrolling back to get the break point. In the example of figure 6-b the while loop is unrolled on K iterations. This method enables vector instructions while reducing the number of branches by a factor around K. The overhead for the backward unrolling is negligible and can be automatically adapted from the maximum size of the loop. In the code this optimization is added to the previous one (blitz++ arrays plus C pointers, figure 5-c).

4.3. Moving local data allocations outside the multithreaded sections

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.

In a near future we aim to consider some systems like IBM Blue Gene have memories based on several physical modules, or *banks*, having their own read/writes queues. The maximum memory bandwidth is achieved by accessing simultaneously all the banks, but it requires to control the data array alignment in memory and between the threads. In fact, the memory lines are allocated across the banks in a round robin fashion, and assuming the size of the arrays per thread is a multiple of size line all the threads will potentially access the same banks. So, allocating the main local arrays outside the parallel region will allow to ease control by adding an adaptive offset values to align arrays in

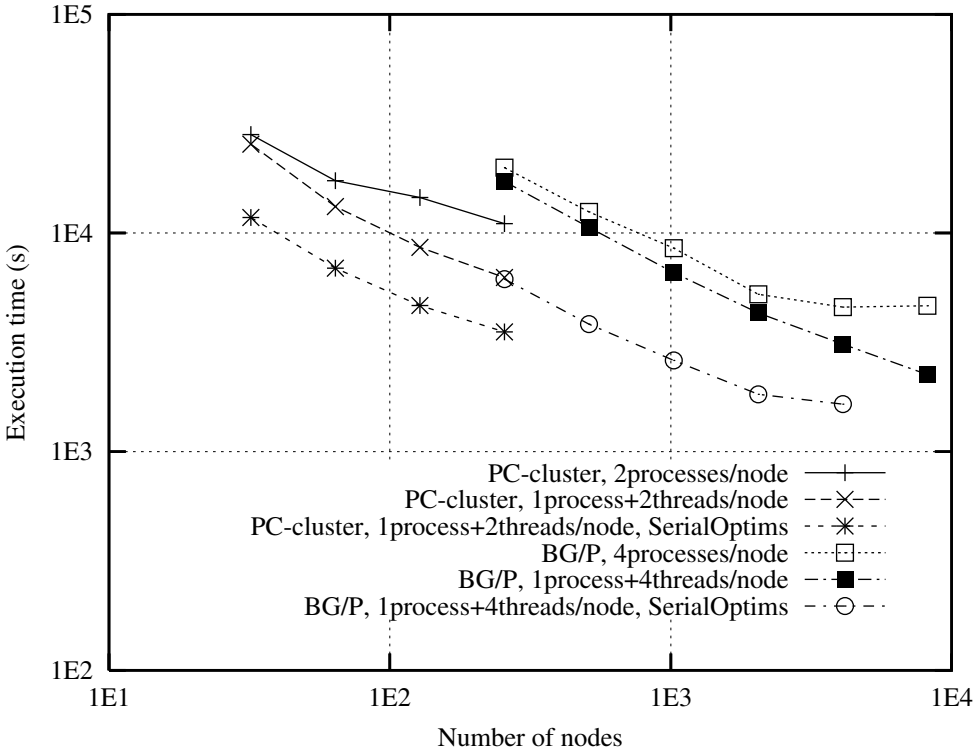

Figure 7. Total execution times

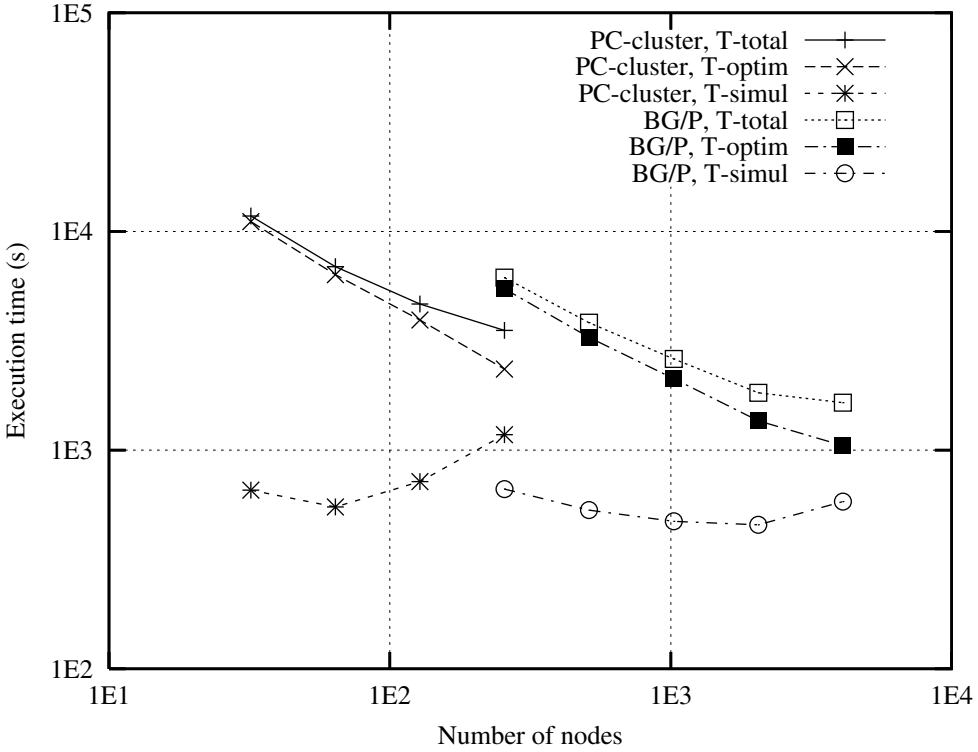

Figure 8. Details of the best execution times

different banks. This technique can improve the performance by 20% on Blue Gene architecture [8].

5. Experimental performances

To evaluate our parallelization and optimizations we run a 7-stocks and 10-state variables benchmark on two testbeds:

- The first was a 256-PC 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.
- The second 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.

Some previous performances on other testbeds, like a Blue Gene/L of EDF, without serial optimizations can be found in [9]. Figure 7 shows the different total execution times on the two testbeds introduced above for the following parallelizations:

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

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 a smaller and deceases regularly up to 256 nodes and 512 cores on PC cluster, and up to 8192 nodes and 32768 cores on Blue Gene/P. When adding serial optimizations the execution time is divided by a factor 1.63 to 2.14 on the PC cluster, and by a factor 1.88 to 2.79 on the Blue Gene/P supercomputer (depending on the number of used nodes). However, the scalability of this benchmark becomes limited: the execution time decrease tends to stop on large numbers of nodes.

Figure 8 shows the details of the best execution times (using multithreading and implementing serial optimizations). We can observe the *optimization* part of our applications scales while the *simulation* part does not speedup and limits the global performances and scaling of the application. However, considering 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.

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 $1h$ on our PC-cluster with 256 nodes and 512 cores, and in less than $30mn$ 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), promising high performances for future industrial use cases where the *optimization* part will increase and will become a more significant part of the application.

Acknowledgment

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

References

[1] X. Warin and W. van Ackooij, "Electricity asset management with future hedging," EDF, Tech. Rep. H-R33-2006-04103-FR, 2008.

[2] T. A. Rotting and A. Gjelsvik, "Stochastic dual dynamic programming for seasonal scheduling in the norwegian power system," *Transactions on power system*, vol. 7, no. 1, 1992.

[3] J. C. Culioli and G. Cohen, "Decomposition-coordination algorithms in stochastic optimization," *SIAM Journal of Control and Optimization*, vol. 28, no. 6, 1990.

[4] L. Bacaud, C. Lemarechal, A. Renaud, and C. Sagastizabal, "Bundle methods in stochastic optimal power management: A disaggregated approach using preconditioner," *Computational Optimization and Applications*, vol. 20, no. 3, 2001.

[5] H. Heitsch and W. Romisch, "Scenario reduction algorithms in stochastic programming," *Computational Optimization and Applications*, vol. 24, 2003.

[6] R. E. Bellman, *Dynamic Programming*. Princeton University Press, Princeton, 1957.

[7] F. Longstaff and E. Schwartz, "Valuing american options by simulation : A simple least-squares," *Review of Financial Studies*, vol. 14, no. 1, Spring 2001.

[8] O. Lascu, N. Allsopp, P. Vezolle, J. Follows, M. Hennecke, F. Ishibashi, M. Paolini, and al., *Unfolding the IBM eServer Blue Gene Solution*, ser. IBM Redbooks. IBM, September 2005.

[9] S. Vialle, X. Warin, and P. Mercier, "A n-dimensional stochastic control algorithm for electricity asset management on pc cluster and blue gene supercomputer," in *9th International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA08)*, NTNU, Trondheim, Norway, May 13-16 2008.