Chapter 1

Optimizing computing and energy performances in heterogeneous clusters of CPUs and GPUs

Stephane Vialle

SUPELEC - UMI GT-CNRS 2958 & AlGorille INRIA Project Team, France

Sylvain Contassot-Vivier

Lorraine University - Loria / AlGorille INRIA Project Team, France

Thomas Jost

ALICE and AlGorille INRIA Project Teams, France

1.1 Introduction

Today multicore CPU clusters and GPU clusters are cheap and extensible parallel architectures, achieving high performances with a wide range of scientific applications. However, depending on the parallel algorithm used to solve the addressed problem and on the available features of the hardware, relative computing and energy performances of the clusters may vary. In fact, modern clusters cumulate several levels of parallelism. Current cluster nodes commonly have several CPU cores, each core supplying SSE units (Streaming SIMD Extension: small vector computing units sharing the CPU memory), and it is easy to install one or several GPU cards in each node (Graphics Processing Unit: large vector computing units with their own memory). So, different kinds of computing *kernels* can be developed to achieve computations on a same node, some for the CPU cores, some for the SSE units and some others for the GPUs. And several combinations of those kernels can be used considering:

1. a cluster of multicore CPUs
2. a cluster of GPUs
3. a cluster of multicore CPUs with SSE units
4. a hybrid cluster of both GPUs and multicore CPUs
5. a hybrid cluster of both GPUs and multicore CPUs with SSE units

Each solution exploits a specific hardware configuration and requires a specific programming, and the different solutions lead to different execution times and energy consumptions. Moreover, the optimal combination of kernel and hardware configuration also depends on the problem and its data size.

Another aspect which impacts the performances lies in the communications. According to the algorithm used and to the chosen implementation, communications and computations of the distributed application can overlap or can be serialized. Overlapping communications and computations is a strategy that is not adapted to every parallel algorithm nor to every hardware, but it is a well-known strategy that can sometimes lead to serious performance improvements. In that context, *asynchronous parallel algorithms* are known to be very well suited. Asynchronous schemes present the great advantage over their synchronous counterparts to perform an implicit overlapping of communications by computations, leading to a better robustness to the interconnection network performances fluctuations and, in some contexts, to better performances [4]. Moreover, although a bit more restrictive conditions apply on their use, a wide family of scientific problems support them.

So, some problems can be solved on current distributed architectures using different *computing kernels* (to exploit the different available computing hardware), with *synchronous* or *asynchronous* management of the distributed computations, and with *overlapped* or *serialized* computations and communications. These different solutions lead to various computing and energy performances according to the hardware, the cluster size and the data size. The optimal solution can change with these parameters, and applications users should not have to deal with these parallel computing issues.

The main goal in this field is to develop auto-adaptive multi-algorithms and multi-kernels applications, in order to achieve optimal runs according to a user-defined criterion (minimize the execution time, the energy consumption, or minimize the energy-delay product...). A *multi-target* and *multi-code* program should include several solutions (implementations) and should be able to automatically select the right one, according to a criterion based on execution speed, on energy consumption or on a speed-energy trade-off. In order to implement this kind of auto-selection of a hardware configuration to exploit and code to run, it is necessary to design a computing and energy performance model.

However, the development of this kind of auto-adaptive solutions remains a real challenge as it requires an *a priori* knowledge of the behavior of each software solution on each class of hardware. Obviously, it is not possible to collect such information for every possible combination. Nonetheless, the design of a model considering heterogeneous distributed architectures, computing performances and energy performances could fill that gap. Such a design is usually achieved by a theoretical analysis of the behaviors of the main classes of parallel algorithms. Then, the result is commonly a model having several parameters depending on the problem and algorithm (nature and data size) but also on the hardware features. Those last information are obtained by the use of generic benchmarks on the target systems.

In the following section, we present the feedback we got from our previous experiments on clusters of CPUs and GPUs and we identify some pertinent benchmarks and optimization rules which can be respectively used to feed a model and to enhance the overall performances. Then, our methodology and metrics for performance evaluation are presented in Section 1.4. Based on the experience gained in our previous works together with a theoretical analysis, a model of computing and energy performance is proposed in Section 1.5 and Section 1.6 and is validated in Section 1.8. All the tests are done with a representative example of scientific computing algorithms, which is a PDE solver detailed in Section 1.7. Finally, we conclude over the current degree of development of fully auto-adaptive algorithms and the short and middle term achievements that can be expected.

1.2 Related works

Many researches have been achieved to design energy performance models of GPUs. They usually focus on modeling one GPU chip, or one GPU card.

In 2008, Rofouei *et al.*[21] introduce a new hardware and software monitoring solution (called LEAP-server), to achieve real-time measurement of the energy consumption of the CPU chip, the GPU card and the PC motherboard. Then the authors run some benchmarks and attempt to link computing performances (execution times and speedup) to energy performances, and they achieve fine measures exhibiting different energy consumptions in function of the GPU memory used. GPUs have different memories, with different performances in term of speed and energy consumption (see [17]). Finally, their measurement solution and performance modeling aims at deciding whether the computations has to be run on the CPU or on the GPU to optimize the performances. In 2009, Ma *et al.*[20] use conventional hardware, classical electrical power measurement mechanisms and functions of the NVIDIA toolkit to measure the workload of the different components of the GPU. It leads to fine measurement and complex workload and electrical power profiles of different benchmark applications. Then, the authors run a statistical analysis, with a *Support Vector Regression* model (SVR) trained on the benchmark application profiles. The resulting SVR is used to predict the energy performances of new applications in function of their workload profiles, in order to automatically select CPU or GPU computing kernels.

SPRAT is a language designed by Takizawa *et al.*[22] to process some *streams* on CPU or on GPU. It aims at reducing the energy consumption without degrading the computing performances. A performance model is introduced by the authors to take into account the execution times on CPU and on GPU, the data transfer time between CPU and GPU, and the energy consumption. The authors focus on the cost of data transfers, which can be excessive and then lead to longer execution times on the GPU than on the CPU. Some *credits* are introduced in SPRAT runtime. The amount of credits increases when the GPU kernel execution appears to be efficient, and they allow to take the risk of running another GPU kernel (instead of a CPU one). This model and language seems especially suited to applications that require frequent data transfers between CPU and GPU.

All these models and choice strategies between a CPU and a GPU kernel have been designed to optimize the usage of one GPU card and one CPU motherboard. However, our goal is to use a *cluster of PCs with CPU and GPU on each node*. So, we decided to monitor the energy consumed by each node (*i.e.* by each complete PC), and to optimize the global energy consumption of our clusters (*i.e.* the energy required by the nodes and the interconnection network). Moreover, we are not only interested in choosing the right kernel (CPU or GPU kernel) run on each node, but also in determining the right operating mode (typically between synchronous and asynchronous versions of a parallel algorithm). The next section introduces our first parallel performance model and experimental measures.

The choice between computing and energy performances can be relevant when the fastest solution is not the less energy-consuming. A global performance criterion can be required to achieve the right choices of computing kernels and operating mode. An interesting global criterion is the *Energy Delay Product* (EDP) introduced in [16] in 1996. It is the product of the energy consumption and the execution time: the two parameters we want to decrease. Looking for the solution that minimizes their product can lead to a good compromise between computing and energy performances. We plan to include the EDP in our future model, in order to track this compromise.

1.3 First experiments and basic model

In this section we consider 3 benchmarks: 3 applications distributed on computing clusters with CPU or GPU nodes. These applications are classical intensive computations: (1) a *European option pricer* corresponding to embarrassingly parallel computations, (2) a *PDE solver* corresponding to an iterative algorithm including a large amount of computations and several communications at each iteration, and (3) a *Jacobi relaxation* corresponding to an iterative algorithm with a huge number of iterations and a small amount of computations and communications at each iteration. Each benchmark implementation has been optimized both on CPUs and GPUs (especially the memory accesses), and has been experimented on a CPU cluster and a GPU cluster. Execution times and consumed energies have been measured and are introduced, analyzed and modeled in this section.

1.3.1 Testbed introduction and measurement methodology

Our first testbed is a cluster of 16 nodes. Each node is a PC composed of an Intel Nehalem CPU with 4 hyperthreaded cores at 2.67GHz, 4GB of RAM, and a NVIDIA GTX285 GPU with 1GB of memory. This cluster has a Gigabit Ethernet interconnection network built around a small DELL Power Object 5324 switch (with 24 ports). The energy consumption of each node is monitored by a Raritan DPXS20A-16 device, that continuously measures the electric power dissipation (in Watts) and can monitor up to 20 nodes. This device hosts a SNMP server that a client can question to get the instantaneous power consumption of each node.

In order to achieve both computing and energy performance measurements of our parallel application, we run a shell script that: (1) runs a Perl SNMP client sending requests to the SNMP server of the Raritan monitor device to sample the electric power consumption, and storing data in a log file, (2) runs the parallel application on a cluster (executing a `mpirun` command), (3) extracts the right data from the log file and computes the energy consumption when the parallel application has finished. The sampling period of the power dissipation on any node of the cluster is approximately $300ms$, and the consumed energy is computed as a definite integral using the trapezoidal rule. The measurement resolution of our complete system appears to be close to $6Watts$, *i.e.* approximately 4% of the measured values.

We only consider the energy consumption of the nodes that are actually used during the computation, as it is easy to remotely switch off unused nodes of our GPU cluster. However it is not possible to switch off the GPU of one node when using only its CPU, and we have not yet tried to reduce the frequency and the energy consumption of the CPU when using mainly the GPU. We also consider the energy consumption of the cluster interconnection switch, that appears to be very stable, independently of the communications achieved across the cluster.

1.3.2 Observation of experimental performances

European option pricer benchmark: Pricing is a very classic and frequently run computation in the financial industry. Many banks aim at speeding up and improving its energy consumption using different parallel architectures, including GPUs. This pricer of European options is based on independent Monte Carlo simulations, and from a pure parallel algorithmic point of view, this is an *embarrassingly parallel* algorithm: each computing node processes some independent Monte Carlo trajectories, and communications are limited to data distribution at the beginning of the application, and to the gathering of the results of each computing node at the end. However, we have been very careful about the parallelization of the random number generator (RNG), to be able to generate uncorrelated random numbers from thousands of threads spread on different nodes [1] without decreasing the pricing accuracy.

The implementation on multicore CPU clusters has been achieved using both MPI, to create one process per node and to insure the few inter-nodes communications, and OpenMP to create several threads per core and take advantage of each available core. The OpenMP parallelization has been optimized to create the required threads only once (inside a large parallel region), and to balance the work among these threads. Moreover, inside each thread, data storage and data accesses are implemented in order to optimize cache memory usage. The implementation on GPU clusters uses the same MPI based computation distribution and internodes communication, while CUDA is used to send data and Monte Carlo trajectory computations on the GPU of each node. In order to avoid frequent data transfers between CPU and GPU, we have ported our RNG to the GPU and all node computations are executed on the GPU. Moreover, we have optimized the GPU memory accesses, mini-

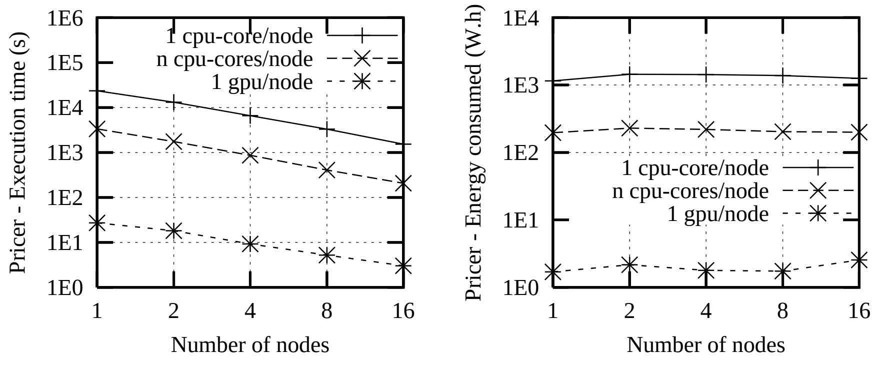

FIGURE 1.1: First benchmark: European option pricer

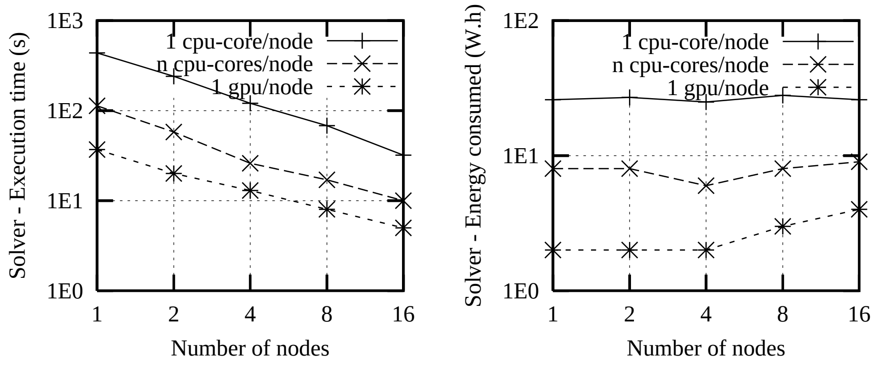

FIGURE 1.2: Second benchmark: PDE solver

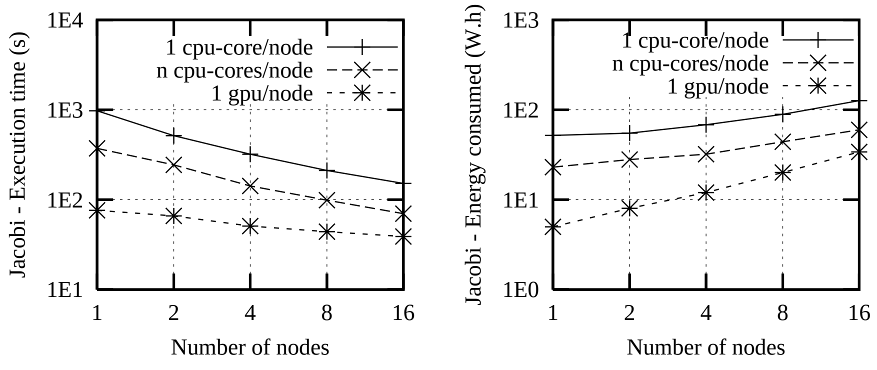

FIGURE 1.3: Third benchmark: Jacobi relaxation

mizing the usage of the (slow) global memory of the GPU, and using mostly (fast) GPU registers.

Figure 1.1 shows quasi-ideal execution time decreases on CPU and GPU clusters, while the energy consumptions remains approximately constant. Finally, the execution on the GPU cluster seems really more efficient than the execution on the multicore CPU cluster.

PDE solver benchmark: This application performs the resolution of partial differential equations (PDEs) using the multisplitting-Newton algorithm and an efficient linear solver using the biconjugate gradient algorithm. The solver is applied to the resolution of a 3D transport model, which simulates chemical species in shallow waters. That application is fully detailed in Section 1.7.

Figure 1.2 exhibits very good decreases of the execution times, and using a GPU cluster seems more interesting than using a CPU cluster. However, the consumed energy does not remain constant and increases significantly when using more than 4 GPUs.

Jacobi relaxation benchmark: This application solves the Poisson's equation on a 2D grid using a Jacobi algorithm. The values on the border of the grid are fixed. The goal is to compute the values inside the grid. In these implementations, at each iteration, a new grid is computed such that the value at each point becomes the mean value of its four neighbors in the previous grid state. Hence, for each point, only three additions and one division by four are required, and with a naive implementation, five memory accesses would be required (four reads and one write). But on modern architectures, memory accesses are much more expensive than computations [23], and this application is *memory bound*. This is the reason why all our optimizations on CPU and on GPU aim at reducing bandwidth consumption and at using the memory bus efficiently.

The CPU implementation has been designed to access to contiguous data elements in order to use the cache memory efficiently. For the considered grid sizes, several rows fit in the cache memory. Hence data elements will be computed row by row. This way, the number of cache misses is minimized and as a consequence the number of memory accesses is minimized as well. A blocked version computing 8×8 blocks has been tried but it showed less interesting performances. The only two effective improvements we have found are loop-unrolling and padding to grid sizes that are multiple of 16 elements, to avoid accesses separated by the *critical stride* as explained in [14]. The GPU implementation has been designed for GPU without generic cache mechanisms. It aims at optimizing the usage of the small *shared* memories available on-chip, that can be used as a *software-managed cache memory*. However, the shared memory is too small to contain several rows of the grid. As a consequence, data partitioning techniques have been used inside each computing node to process the Jacobi Grid per blocks. The size of these blocks has been optimized to get *coalesced* memory accesses.

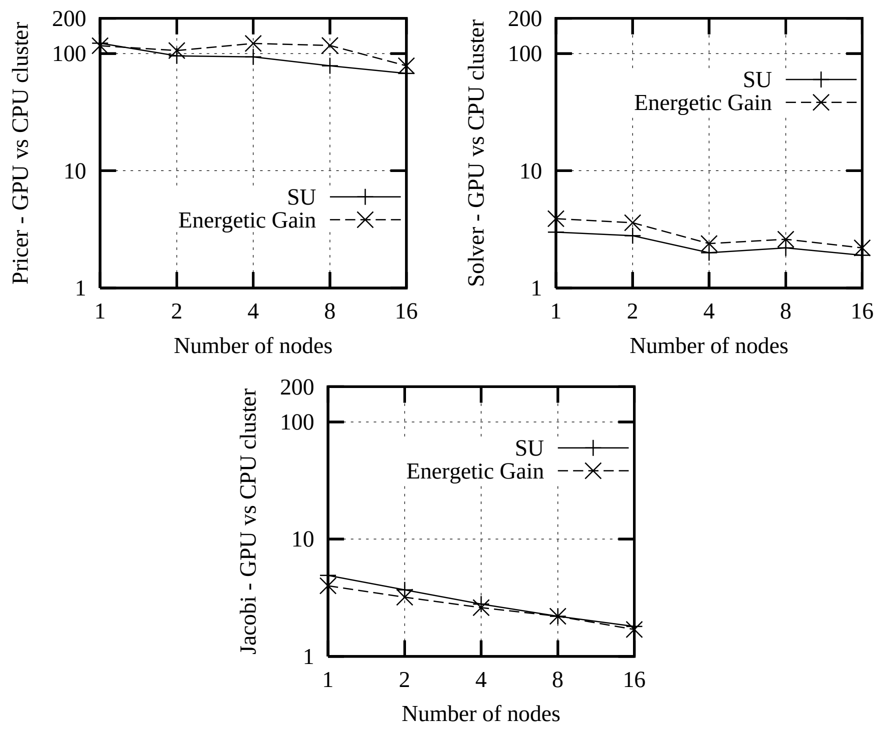

FIGURE 1.4: GPU cluster *vs* CPU cluster relative performances

Moreover, as we use clusters, inter-nodes communications are required at each iteration. But they exchange data between CPU memories or between GPU memories (on GPU clusters), that can be long compared to the computation speed of each node. So we have optimized these communications, implementing overlapped asynchronous communications, and overlapping with CPU-to-GPU and GPU-to-CPU data transfers when possible.

Figure 1.3 shows better performances when using the GPU cluster instead of the CPU one, but the speedup is poor and the energy consumption is clearly increasing with the number of used nodes. Finally, the GPU cluster appears always more efficient than the CPU cluster. But the computing and energy performance profiles of these 3 benchmarks are different, and the superiority of the GPU cluster seems to evolve with the number of used nodes. This issue is investigated in the next section.

1.3.3 Relative performances of the CPU and GPU clusters

In order to analyze the interest to use a GPU cluster in place of a CPU cluster, we have computed the relative speedup (SU) and the relative energy gain (EG) of the GPU cluster compared to the CPU one. Figure 1.4 illustrates these computing and energy relative performances for the 3 benchmarks introduced in the previous section. We can observe that:

- The *option pricer* with embarrassingly parallel computations achieves a speedup and an energy gain close to 100 on a GPU cluster compared to a multicore CPU cluster, while the *PDE solver* and the *Jacobi relaxation*, including computations and communications, reach speedup and energy gain in the range from 1.6 to 10.

- Speedup and energy gain decrease when the number of used nodes increases. This phenomena is limited with the *option pricer*, but is stronger with the *PDE solver* and very clear with the *Jacobi relaxation*. On larger clusters, performances could become higher on multicore CPU clusters.

- Speedup and energy gain curves exhibit very similar profiles. They seem to have the same behavior.

In fact, when the interconnection network is the same, the communications are a little bit longer on a GPU cluster. As we need to exchange some data located in GPU global memories, data have to be transferred from GPU memories to CPU ones. Then, internodes communications must be performed (using MPI for example) and, finally, the received data must be transferred into the global GPU memories. On the opposite, computations are faster on GPUs. So, the ratio of communication time in the total execution time significantly increases when using a GPU cluster. That leads to a smaller performance increase when using more nodes, and finally the multicore CPU cluster can become more efficient.

The next section establishes a first performance and energy consumption model on two different clusters. This model is limited to *scalability areas* of the performance curves, and is applied to the cases of a multicore CPU cluster and a GPU cluster. It aims at predicting the most efficient cluster as a function of the number of used nodes, and at helping us to always choose the best solution.

1.3.4 Basic modeling of CPU and GPU cluster performances

Execution time and energy consumption on one cluster: Considering A: a cluster, N: the number of used nodes, $T(A,N)$: the execution time of an application, and $E(A,N)$: the consumed energy, in the ideal case we have: $T(A,N) = T(A,1)/N$, and $E(A,N) = E(A,1)$ is constant. Our first benchmark (European option pricing achieving embarrassingly parallel computations) exhibits performances close to this ideal case (see Fig. 1.1). But the other experiments do not exhibit these ideal performances. However, lines appear approximately straight when using logarithmic scales, meaning the parallelization *scales*, so we can write:

$$T(A,N) = \quad T(A,1)/N^{\sigma_T^A}, \quad 0 \le \sigma_T^A \le 1 \tag{1.1}$$

$$E(A,N) = \quad E(A,1) \cdot N^{\sigma_E^A}, \quad 0 \le \sigma_E^A \le 1 \tag{1.2}$$

Where σ_T^A and σ_E^A are the slopes of the straight lines of the execution time and energy consumption on curves drawn with logarithmic scales. In the ideal case, these parameters values would be 1, but in practice they are less than 1 (see Fig. 1.2 and Fig. 1.3).

So, when the parallelization scales, the speedup and energy gain of cluster A compared to a sequential run on one of its nodes (*level 1* gains) are:

$$SU_1(A,N) = \frac{T(A,1)}{T(A,N)} = \quad N^{+\sigma_T^A} \quad (\ge 1) \tag{1.3}$$

$$EG_1(A,N) = \frac{E(A,1)}{E(A,N)} = \quad N^{-\sigma_E^A} \quad (\le 1) \tag{1.4}$$

In this basic model we make the following assumptions:

1. The electric power dissipated by one node of cluster A during a sequential computation remains constant and is equal to $P(A,1) = E(A,1)/T(A,1)$.
2. The electric power dissipated by any used node of cluster A during a parallel computation is independent of the number of used nodes, and is equal to $P(A,1)$.
3. The electric power dissipated by the network switch of the cluster remains constant and is equal to $P_{switch}(A)$.

Hypothesis 3 matches all our observations on different clusters. But we will see in Section 1.5 that hypotheses 1 and 2 are approximations. However, assuming these hypotheses, the energy consumed on one node during a sequential run is:

$$E(A,1) \quad = \quad P(A,1) \cdot T(A,1) + \frac{P_{switch}(A)}{N_{max}} \cdot T(A,1) \tag{1.5}$$

where N_{max} is the maximal number of available nodes in the cluster. And the energy consumed on N nodes during a parallel run is:

$$E(A,N) \quad = \quad P(A,1) \cdot T(A,N) \cdot N + \frac{P_{switch}(A) \cdot N}{N_{max}} \cdot T(A,N) \tag{1.6}$$

$$E(A,N) \quad = \quad \frac{T(A,N)}{T(A,1)} \cdot N \cdot \left(P(A,1) \cdot T(A,1) + \frac{P_{switch}(A)}{N_{max}} \cdot T(A,1) \right)$$

Using equations 1.1 and 1.5, we get:

$$E(A,N) \quad = \quad N^{1-\sigma_T^A} \cdot E(A,1) \tag{1.7}$$

and using equation 1.2 we can deduce the relation between the execution time and energy consumption formulas on one cluster, in the scalability area of the performance curves:

$$\sigma_E^A = 1 - \sigma_T^A \tag{1.8}$$

Relative speedup and energy gain between two clusters: To identify the most efficient parallelization on two different clusters A and B, we compute the respective gains on cluster A compared to the gains on cluster B (level 2 gains): $SU_2^{A/B}(N)$ and $EG_2^{A/B}(N)$. This leads to:

$$\begin{aligned} SU_2^{A/B}(N) &= \frac{T(B,N)}{T(A,N)} \\ &= \frac{T(B,1)}{T(A,1)} \cdot N^{\sigma_T^A - \sigma_T^B} \\ &= SU_2^{A/B}(1) \cdot N^{\sigma_T^A - \sigma_T^B} \end{aligned} \tag{1.9}$$

and to:

$$\begin{aligned} EG_2^{A/B}(N) &= \frac{E(B,N)}{E(A,N)} \\ &= \frac{E(B,1)}{E(A,1)} \cdot N^{\sigma_E^B - \sigma_E^A} \\ &= EG_2^{A/B}(1) \cdot N^{\sigma_T^A - \sigma_T^B} \end{aligned} \tag{1.10}$$

These two relative gains have different initial values (on 1 node) but similar evolutions (same $\sigma_T^A - \sigma_T^B$ exponent in the gain expressions). This is the reason why the speedup and energy gain curves on Fig. 1.4 are nearly parallel.

As an example, let's consider that cluster A has faster nodes than cluster B. Depending on the relative values of σ_T^A and σ_T^B, it is possible that the relative gain $SU_2^{A/B}$ becomes smaller than 1 inside the scalability area (the validity domain of equation (1.9)) beyond a threshold number of nodes $N_{A/B,T}$. Then, cluster B runs faster than cluster A beyond this threshold, and similarly, cluster B can be less energy-consuming beyond a threshold $N_{A/B,E}$. Using equations (1.9) and (1.10) we can determine these two thresholds:

$$SU_2^{A/B}(N_{A/B,T}) = 1 \iff N_{A/B,T} = \left(SU_2^{A/B}(1)\right)^{\frac{-1}{\sigma_T^A - \sigma_T^B}} \tag{1.11}$$

$$EG_2^{A/B}(N_{A/B,E}) = 1 \iff N_{A/B,E} = \left(EG_2^{A/B}(1)\right)^{\frac{-1}{\sigma_T^A - \sigma_T^B}} \tag{1.12}$$

Application of the model to a CPU and a GPU clusters: When using one GPU, the computation time is smaller than on one CPU (or we do not use the GPU), and $SU_2^{g/c}(1) > 1$. But communication times are longer on a

GPU cluster than on a CPU cluster with the same interconnection network, because they require the same CPU communications plus some CPU-GPU data transfers. So, the scalability is *weaker* on the GPU cluster and the σ_T parameter is smaller: $0 \leq \sigma_T^{gpu} \leq \sigma_T^{cpu} \leq 1$. Moreover, although the dissipated electric power of a single GPU node is larger than the one of a CPU node, the overall energy consumption of the GPU cluster is usually lower than the CPU cluster: $EG_2^{g/c}(1) > 1$. These hypotheses are verified on our three benchmarks (see Fig. 1.1, Fig. 1.2 and Fig. 1.3). When these hypotheses are true, the two threshold numbers of nodes (equations (1.11) and (1.12)) exist and are greater than 1. Then we define:

$$N_{g/c,min} = \min(N_{g/c,T}, N_{g/c,E}) \quad (1.13)$$
$$N_{g/c,max} = \max(N_{g/c,T}, N_{g/c,E}) \quad (1.14)$$

When $N < N_{g/c,min}$ the GPU cluster solution is faster and less energy-consuming, when $N_{g/c,max} < N$ the multicore GPU cluster is slower and more energy-consuming, and when the number of nodes is in the range $[N_{g/c,min}; N_{g/c,max}]$ the GPU cluster is either faster or less energy-consuming.

So, assuming our different hypotheses are true (mainly expressed by the existence of a scalability area), three different *execution configurations* exist:

- $N < N_{g/c,min}$: the GPU cluster is more interesting.
- $N_{g/c,min} \leq N \leq N_{g/c,max}$: the GPU cluster is more or less interesting than the CPU cluster, depending on the relative importance of the computation speed and the energy consumption.
- $N_{g/c,max} < N$: the CPU cluster is more interesting.

1.3.5 Need for predicting the best operating mode

Using a GPU cluster to run an embarrassingly parallel program (without internode communications) and with all computations running on the GPU, such as our European option pricer, can be very efficient (see Section 1.3.2 and Fig. 1.1). The speedup and energy gain of one GPU node compared to one multicore CPU node are close to 100 and remain close to 100 when the number of nodes increases. Such a case corresponds to the first category described above.

When all computations are not run on the GPU and frequent data transfer are required between CPU and GPU on each node, and/or frequent internode communications are required, the speedup and energy gain on one node are more limited and decrease when the number of nodes increases. Our PDE solver and Jacobi relaxation benchmarks belong to this second category (see Section 1.3.2, Fig. 1.2 and Fig. 1.3).

Finally, most parallel programs include internode communications and data transfers between CPUs and GPUs, and do not perform all their computations on the GPUs. So, the three *execution configurations* introduced at the end of section Section 1.3.4 exist, and using a GPU cluster with a large number of nodes may lead to an important gain or to an important waste of time and energy.

Usually, the end user is not a computer scientist. Although he is capable of specifying if he wishes to decrease the execution time or the energy consumption, he is not able to choose whether or not to use GPUs, depending on the number of available and used nodes. A heuristic has to be designed and implemented in order *to achieve an automatic choice* of the right *computing kernel* to use in *execution configurations* 1 and 3, and to respect the user objective in *execution configuration* 2.

1.3.6 Interests and limits of the basic model, and need for a new one

The heuristic we introduced in the previous section needs a model to predict performances and choose a *computing kernel*. Our previous model, defined in section 1.3.4, does not make any assumption on the internal architecture of the clusters A and B. It is based on the observation of a scalability area and on measurement of computing and energy performances of clusters A and B when using two different numbers of nodes. This leads to achieve at least four benchmarks of the target application, and to establish some kind of experimental reference curves. This approach is acceptable before to enter an *exploitation mode*, running many times and frequently the application. Then, achieving four extra-runs of the application code in order to improve all others, seems the right solution. Of course it is not always possible to achieve some runs on only one node, depending on the problem size, but the model equations could be adapted to not require experiments on one node.

However, many researchers have to run their parallel application in *experimentation mode*: they achieve only few runs (but large runs) before to upgrade the application. Then, execution of four benchmarks before to optimize the execution of each new version of the application can be prohibitive. To optimize the execution of a parallel application in *experimentation mode* we need a more accurate model of our architecture, requiring only *elementary benchmarks* to be calibrated for our machine. Such a model will be designed in sections 1.5 and 1.6.

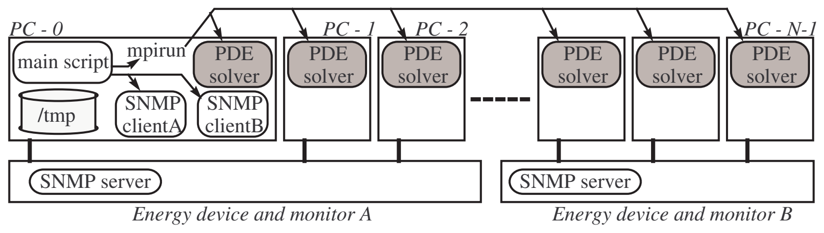

FIGURE 1.5: Hardware and software architecture of the test platform.

1.4 Measurement issues and alternative modeling approach

1.4.1 Measurement methodology

Figure 1.5 illustrates the hardware and software configuration of our CPU+GPU cluster PC, the energy devices and monitors as well as the benchmarking mechanism. As already mentioned in Section 1.3.1, each PC is electrically connected to an output port of an energy device (a Raritan DPXS20A-16), that measures the instantaneous electrical power dissipation of all its outputs. Those measures are collected via SNMP requests.

In order to measure both computing and energy performances of our applications our benchmarking methodology is the following:

1. We allocate some nodes on our cluster, through the OAR cluster management environment, and run a main shell script on the first node we get (nodes are sorted by alphabetic order).

2. The main shell script starts a SNMP client (Perl script) for each SNMP server (*i.e.* for each electric device and monitor to use).

3. Each SNMP client sends requests to a SNMP server to sample the electrical power dissipated by each of its outputs. The sampling interval is close to $300ms$ in our experiments, but this can be tuned. Each SNMP client stores the acquired data in log files on the local disk of the first PC. Each line of a log file stores the sample number, the sample time and the electrical power dissipation of each output port of the electrical device at the sample time.

4. The main shell script waits for the different log files to be created (meaning the energy monitoring mechanism is running). Then, it reads these files and gets the current sample number from each file. The main shell

script runs the parallel application on the allocated nodes, executing a `mpirun` command, while the electrical power dissipation sampling continues.

5. The execution time of the MPI parallel program is measured internally using `gettimeofday`.

6. When the `mpirun` command finishes, the main shell script reads the log files, and get the new current sample numbers. Then it stops the SNMP clients and computes the energy consumed by all the PC nodes involved in the parallel run. It computes a definite integral of the instantaneous power measures between the sample numbers surrounding the `mpirun` execution, using the trapezoidal rule and considering time measures stored in each line of the log files.

Finally, each benchmark is repeated 5 times, then the average values of execution time and energy consumption are computed.

1.4.2 Technical limitations of the measures

A first point that may induce small perturbations in the performance measurements is the concurrent execution of several SNMP clients on the first allocated node. These processes run concurrently of the parallel application and might disturb it. Nonetheless, during our benchmarks we did not observe any significant impact on the execution time measures when running the energy consumption measurement. But users must be aware that it could happen on a larger system, using many energy devices and monitors and running many SNMP clients.

The first real limitation of our benchmarking system is about the uncertainties of the measures. For the whole set of experiments involving several nodes of the cluster (with or without GPU, in synchronous or asynchronous mode), the average variations of the execution times are close to 7%, and the average variations of energy consumptions are around 7.5% and are a bit less stable. The good point is that those variations seem to be strongly correlated. Although they may be partially explained by the sensor quality concerning the energy consumption measures, it remains quite difficult to precisely identify their exact sources.

We also observed four other sources of measurement uncertainties. But those ones can be analyzed and taken into account in our model:

1. There are serious variations in the power consumptions of the different PCs in a homogeneous cluster. Current PCs with identical external features have very close computing performances, but their energy consumptions can vary a lot. So, we decided to consider the energy consumption of each node in our model rather than considering a global average energy consumption of the cluster.

2. We observe some steps in electrical power dissipation of each cluster node. When running computations on the GPU, the dissipated electrical power increases, stabilizes and then increases again before stabilizing again. We observe the symmetric phenomena when computations end. This is mostly due to the GPU fans start/stop cycles. They do not start and stop exactly when computations begin and end, but a little bit later when the GPU temperature increases or decreases and crosses thresholds.

3. We do not observe an instantaneous power dissipation decrease when stopping computations to enter an inter-node communication sub-step of the application. First, sensors do not detect an immediate decrease of power dissipation of the node when intensive computations end. Second, according to the previous point, the fans continue to run up to one minute before stopping. As a consequence, entering a communication sub-step leads to a slow decrease of the energy consumption during this sub-step. In particular, when the sub-step is short, the energy decrease is likely to be unnoticeable. However, for sufficiently long sub-steps, the decrease should reach the idle power level.

4. Performing communications has a negligible impact over the energy consumption. Hence, overlapping computations and communications, like in our asynchronous algorithms, does not lead to any additional energy consumption.

Our model has been designed to take into account these general features of parallel systems, and then to minimize their associated uncertainties.

1.5 Node level model

In this section and the following one, we present a theoretical model linking together the energy and computing performances of two sets of nodes executing a same application. That model is decomposed in two nested levels. The inner one is the node level, described in this section, which consists in a single machine. The second one is the cluster level, described in Section 1.6.

In the scope of this study, dealing with the comparison between CPU and GPU clusters, we focus on intensive computing applications in which other activities (especially disk accesses) are negligible with respect to the computations. Nonetheless, if necessary, such additional activities could be quite easily added to the model presented throughout the two following sections. They would be inserted into the model in a similar way as the computing activities.

Concerning the node level, we link the energy and computing performances of two node configurations. In the following, we do not make any distinction between the denominations *"the two node configurations"* and *"the two nodes"*, but the reader must keep in mind that those *two nodes* may correspond to the same physical node used in two different configurations (use of different numbers of CPU cores and/or additional accelerators: SSE, GPU, FPGA: Field-Programmable Gate Array,...).

1.5.1 Complete model

The hypotheses made over the hardware configuration of a node are very general as a node contains:

- at least one and possibly several CPU cores
- 0 or more additional accelerator cards (GPU and/or FPGA)

In fact, on one node, the power can be divided into several parts: P_I, the power consumed by the system node in *idle* state, which is assumed to be constant during the execution of the application, and a series of U powers $P_C(u_i, t), 1 \leq i \leq U$, each one corresponding to the additional power (in addition to P_I) used by computing unit u_i at time t if it is active (i.e. making computations).

If we assume that:

- the powers of the computing units are cumulative when used together
- for every computing unit u_i, its additional power $P_c(u_i, t)$ is either null (not active) or maximal (active)

then the total power used at each time t on one node is $\sum_{i=1}^{U} P_C(u_i, t) + P_I$, and the total amount of energy consumed during a period T is given by:

$$E = \int_0^T \left(\sum_{i=1}^{U} P_C(u_i, t) + P_I \right) dt = T \cdot P_I + \int_0^T \sum_{i=1}^{U} P_C(u_i, t) dt \qquad (1.15)$$

Then, if we denote by $\beta(u_i)$ $(0 \leq \beta(u_i) \leq 1)$ the ratio of the total execution time of the application during which unit u_i is active, and by $P_C(u_i)$ the additional power used by unit u_i (assumed constant during u_i activity), (1.15) can be reformulated as:

$$E = T \cdot \left(\sum_{i=1}^{U} \beta(u_i) P_C(u_i) + P_I \right) \qquad (1.16)$$

The transition from (1.15) to (1.16) is mathematically valid because the functions $P_C(u_i)$ are piecewise continuous over the integration interval (duration T).

Although this formulation is very interesting, it requires a very accurate knowledge on the behavior of the studied application, which is not always available in practice. Indeed, it is necessary to know the respective periods of use of every computing unit during the execution of the application. In some simple cases, where just a few different types of units are used, it may be possible to get reasonable estimations of those information. However, in most cases, some benchmarks of the application itself are mandatory to get accurate evaluations in its context of use.

1.5.2 Discussion on the importance to avoid benchmarking the target application

An important aspect that has to be taken into account is the way to deduce the different computation ratios. Those ratios can be deduced either by a theoretical analysis or via a series of executions of the target application on the target cluster.

The first solution is not possible when the algorithm and source code of the considered application are not available. Even in the opposite case, it may be a hard task to theoretically deduce the ratios, especially in algorithms where the execution path is irregular. And finally, the time to lead such a detailed study is not always available.

The second solution only depends on the availability of the target cluster, but most of the times this is not an obstacle. Nevertheless, executing the target application several times to deduce its behavior is kind of a nonsense in many situations, especially if we want to minimize the overall energy expenses (which include the benchmarks required for the application setting).

This is why we are concerned with minimizing application-dependent benchmarks by using as much as possible small generic benchmarks. And when benchmarks of the application are mandatory, we suggest to use reduced configurations (problem size,...) to get minimal execution times and energy expenses. The information retrieved from such executions may not be completely representative of the exploitation case, but they should be sufficiently accurate in most cases to allow the model to produce estimations of acceptable quality. Moreover, the experience of the user may help to deduce more accurate parameters from the measured ones.

1.5.3 Simplified model

As we focus on the practical usability of the energy model, we propose to make some approximations over the additional power used during the computations (as opposed to idle times). Thus, we consider only one $P_C(u,t)$, which corresponds to the maximal additional power used by the entire set of computing units u (typically CPU core(s) and/or GPU(s)) used to perform the computations. However, as one may be confronted to the comparison of nodes with different hardware configurations (different numbers of CPU cores, presence or absence of a GPU, model of the GPU if present,...) it is useful to make the distinction between hardware configurations in the model. So, we respectively denote by $P_I(X)$ the idle power of a node with configuration X (for example, all the nodes of type X in a cluster), and $P_C(X, u, t)$ the additional power consumed during the computations performed on a node of type X, with units u at time t.

So, at each time t, the total power used on one node of type A is $P_C(A, u, t) + P_I(A)$ and the total amount of energy consumed during a period T is:

$$\int_0^T (P_C(A,u,t) + P_I(A))dt = T \cdot P_I(A) + \int_0^T P_C(A,u,t)dt \tag{1.17}$$

Now, let's consider two distinct executions of a same application on two nodes A and B. The computing units used on node A for the execution are u_A (CPU core(s) and/or GPU(s)) and the ones used on node B are u_B (CPU core(s) and/or GPU(s)). Also, we denote by $T_A(u_A)$ the total execution time on node A using computing units u_A, and $T_B(u_B)$ the total execution time on node B using units u_B. Then, we will have a lower energy consumption on node A when:

$$T_A(u_A) \cdot P_I(A) + \int_0^{T_A(u_A)} P_C(A, u_A, t)dt \le T_B(u_B) \cdot P_I(B) + \int_0^{T_B(u_B)} P_C(B, u_B, t)dt \tag{1.18}$$

which is equivalent to

$$\int_0^{T_A(u_A)} P_C(A, u_A, t)dt \le T_B(u_B) \cdot P_I(B) - T_A(u_A) \cdot P_I(A) + \int_0^{T_B(u_B)} P_C(B, u_B, t)dt \tag{1.19}$$

Under the same assumptions as in Section 1.5.1, powers at *full load* (i.e. during intensive computations) are considered constant and we have $P_C(A, u_A, t) = P_C(A, u_A)$ and $P_C(B, u_B, t) = P_C(B, u_B)$. Moreover, in the context of a scientific application running on a single node, we consider that idle times are negligible, and then $\beta(u_A)$ and $\beta(u_B)$ are both equal to 1. Then, the constraint becomes:

$$T_A(u_A) \cdot P_C(A, u_A) \le T_B(u_B) \cdot P_I(B) - T_A(u_A) \cdot P_I(A) + T_B(u_B) \cdot P_C(B, u_B) \tag{1.20}$$

which leads to

$$\begin{aligned} P_C(A, u_A) &\le \frac{T_B(u_B)}{T_A(u_A)} \cdot P_I(B) - P_I(A) + \frac{T_B(u_B)}{T_A(u_A)} \cdot P_C(B, u_B) \\ &\le \frac{T_B(u_B)}{T_A(u_A)} \cdot (P_I(B) + P_C(B, u_B)) - P_I(A) \end{aligned} \tag{1.21}$$

inducing a constraint linking together $P_C(A, u_A)$, $P_I(A)$, $P_C(B, u_B)$ and $P_I(B)$.

Let's denote $\alpha = \frac{T_B(u_B)}{T_A(u_A)}$ the speedup of the execution time on node A using computing units u_A with respect to the one on node B using units u_B, then we can rewrite (1.21) as:

$$P_C(A, u_A) \leq \alpha \cdot (P_I(B) + P_C(B, u_B)) - P_I(A) \tag{1.22}$$

Finally, if we denote the respective powers at *full load* by

$$P_F(A, u_A) = P_C(A, u_A) + P_I(A)$$

and

$$P_F(B, u_B) = P_C(B, u_B) + P_I(B),$$

we obtain:

$$P_F(A, u_A) \leq \alpha \cdot P_F(B, u_B) \tag{1.23}$$

which finally gives:

$$\alpha \geq \frac{P_F(A, u_A)}{P_F(B, u_B)} \tag{1.24}$$

which expresses a simple constraint over α, only in terms of the powers at full load for the executions on node A and B with respective units u_A and u_B. It can be noticed that the speedup α can be estimated by experimental measures either with small generic benchmarks and the use of a performance model taking the used computing units into account, or by actual executions of the considered application with small instances of the problem.

For example, a good approximation of a performance model for multi-core nodes is to measure the speedup α_1 which is the ratio of the execution time with 1 core of node A over the one with one core of node B. Then, we deduce α with n_A and n_B cores respectively by $\alpha = \alpha_1 \frac{n_B}{n_A}$.

The interest of that general formulation is that it can be used to compare executions on any couple of nodes. For example, it can be used to compare two nodes with different CPUs, or different GPUs, as well as the same node with and without using a GPU.

In that last case, we have $A = B$, $u_A = GPU$ (precisely 1 CPU core and 1 GPU) and $u_B = CPU$ (1 CPU core), and we can rewrite (1.24) as the simplified formulation:

$$\alpha \geq \frac{P_F(GPU)}{P_F(CPU)} \tag{1.25}$$

1.6 Cluster level model

In this section, we present the cluster level of the simplified model introduced in the previous section.

Let's consider that we want to choose between two configurations of clusters (A and B), respectively having N_A and N_B nodes, to run a given application while minimizing the energy consumption. As in the node level section, we specify the computing units respectively used on each node of the two clusters. For the sake of simplicity, and because it is commonly done in practice, we suppose that all the nodes of a given cluster are used in the same way. This means that the same kind of computing units are used. So, in cluster A, units u_A are used (either CPU core(s) or GPU(s)), and in cluster B, units u_B are used. It must be noticed that the two clusters A and B may be the same physical system.

Also, as it has been observed in practice that there is no significant energy overhead in network switches whether there is some traffic or not, we consider in our model that their energy consumption P_{sw} is constant.

Finally, as we perform a comparison between two clusters for a same application, the parameters of that application (size, initial values,...) are the same on the two clusters. As a first order approximation in our modeling, we consider that the application has the same behavior on all the nodes of the used cluster (sequences of idle/communication and full load periods are identical).

However, we have to make a distinction between a synchronous and an asynchronous application as their computational sequences are quite different, and so are their energy signatures. For clarity sake, the asynchronous version is presented first as it is simpler.

1.6.1 Asynchronous application

In the case of an asynchronous application, the computations are performed uninterruptedly during the execution of the application, in parallel of the communications. This is a rather simple case of parallel execution as it is very similar to the single node model.

In fact, the energy models on clusters A and B take the following forms:

$$\begin{aligned} E_A(u_A) &= T_A(u_A) \cdot (\textstyle\sum_{i=1}^{N_A} P_F^i(A, u_A) + P_{sw}) \\ E_B(u_B) &= T_B(u_B) \cdot (\textstyle\sum_{i=1}^{N_B} P_F^i(B, u_B) + P_{sw}) \end{aligned} \tag{1.26}$$

And the execution on cluster A is more energy interesting than the one on cluster B as soon as $E_A(u_A) < E_B(u_B)$, leading to:

$$T_A(u_A) \cdot \left(\sum_{i=1}^{N_A} P_F^i(A, u_A) + P_{sw} \right) < T_B(u_B) \cdot \left(\sum_{i=1}^{N_B} P_F^i(B, u_B) + P_{sw} \right) \tag{1.27}$$

and if we denote, similarly to the node level part, the relative speedup of the execution on cluster A according to the one on cluster B by $\alpha = \frac{T_B(u_B)}{T_A(u_A)}$, then we obtain:

$$\alpha > \frac{\sum_{i=1}^{N_A} P_F^i(A, u_A) + P_{sw}}{\sum_{i=1}^{N_B} P_F^i(B, u_B) + P_{sw}} \tag{1.28}$$

So, by only knowing the number of used nodes, the powers at full load when using either computing units u_A or u_B, and the power of the network switch(es), we can decide which cluster is the most interesting to use. An advantage of that formulation is that the information required to take that decision can be retrieved by small generic benchmarks. The knowledge of the powers of every node in the considered clusters is actually useful even for homogeneous clusters. Indeed, we have observed in practice that there may be significant differences between the powers of similar nodes (see Table 1.1 in Section 1.8).

In a symmetrical way, (1.28) also specifies a constraint over the minimal relative speedup of cluster A relatively to cluster B in order to get an energy gain. As in the node level context, the speedup α can be estimated without requiring executions of the considered application with full size instances of problem.

Now, in the specific case of deciding whether or not to use the GPUs in a single cluster (implying $A = B$, $N_A = N_B = N$, $P_F^i(A, u_A) = P_F^i(GPU)$ and $P_F^i(B, u_B) = P_F^i(CPU)$), we obtain the following version:

$$\alpha > \frac{\sum_{i=1}^{N} P_F^i(GPU) + P_{sw}}{\sum_{i=1}^{N} P_F^i(CPU) + P_{sw}} \tag{1.29}$$

which can be even more simplified when the power dissipation between the identical nodes is negligible ($P_F^i(X) = P_F^j(X) = P_F(X), \forall i, j \in \{1..N\}$):

$$\alpha > \frac{P_F(GPU) + \frac{P_{sw}}{N}}{P_F(CPU) + \frac{P_{sw}}{N}} \tag{1.30}$$

1.6.2 Synchronous application

In the synchronous case, it is commonly assumed that there are, at least partially, some distinct parts of computation and communication phases. From the energy point of view, the main difference between those two parts comes from the fact that during computations the full power of the node is used (P_F), whereas during synchronous blocking communications, the nodes can be considered in idle state (P_I). Also, during the potential sequences where communications are overlapped with computations, we consider the energy consumption of the nodes at full load (P_F).

The difficult part with synchronous algorithms lies in the necessity to know, at least approximately, the percentage of non-overlapped communications performed during the execution of the algorithm according to the overall execution time. By symmetry, this is equivalent to knowing the percentage of computations ($\beta(u)$ in the complete model in Section 1.5.1). This comes from the distinction in (1.17) of the two different possible phases (computations or communications) during the execution of the application. And, since the application is assumed to have the same behavior on every node of the cluster, as discussed at the beginning of this section, we can rewrite (1.17) for every used node of the cluster, as follows:

$$E_A(u_A) = \int_0^T P_C(A, u_A, t)dt + \int_0^T P_I(A)dt \tag{1.31}$$

And since by definition $P_C()$ is constant (non null) during computations and null otherwise, the previous equation can be reformulated as:

$$E_A(u_A) = T \cdot \beta_A(u_A) \cdot P_C(A, u_A) + T \cdot P_I(A) \tag{1.32}$$

where $\beta_A(u_A)$ is similar to the $\beta(u_i)$ in Section 1.5.1, with the additional precision of the cluster context (configuration A). In practice, a more convenient expression of (1.32) is obtained by replacing $P_C()$ by $P_F()$, the total power at full load:

$$E_A(u_A) = T \cdot (\beta_A(u_A) \cdot P_F(A, u_A) + (1 - \beta_A(u_A)) \cdot P_I(A)) \tag{1.33}$$

Unfortunately, as mentioned at the end of Section 1.5.1, the acquisition of the percentage $\beta_A(u_A)$ requires at least one benchmark of the target application. Moreover, it generally varies in function of the problem size and of the number of nodes in addition to the computing units used.

As a first approximation we consider that for a fixed number of nodes, the computation ratio in synchronous applications slightly changes but does not follow too large variations according to the problem size. In fact, this is rather closely confirmed experimentally, as depicted in Fig. 1.6 for one of the most difficult cases of application which is an iterative PDE solver (see Section 1.7 for further details). It can be seen that for the CPU version, there are significant variations for the very small problem sizes, then the behavior becomes stable. And for the GPU version, the variations are even more limited. Thus, the assumption of a constant computation ratio is reasonably acceptable for most of the applications. So, we propose to approximate that ratio from a single execution of the considered application. The principle is to choose a problem size as small as possible although not too small to avoid non representative ratios. In particular cases where larger variability may be expected, a few executions with slightly different problem sizes could be considered to obtain a representative average ratio.

Then, we obtain the following total energy consumption when running the considered application on cluster A with computing units u_A:

$$E_A(u_A) = T_A(u_A) \cdot \left(\beta_A(u_A) \cdot \sum_{i=1}^{N_A} P_F^i(A, u_A) + (1 - \beta_A(u_A)) \cdot \sum_{i=1}^{N_A} P_I^i(A) + P_{sw} \right) \tag{1.34}$$

So, when comparing two clusters A and B, cluster A becomes more interesting than cluster B according to the energy aspect as soon as $E_A(u_A) < E_B(u_B)$, leading to:

$$\begin{aligned} T_A(u_A) \cdot (\beta_A(u_A) \cdot \textstyle\sum_{i=1}^{N_A} P_F^i(A, u_A) + (1 - \beta_A(u_A)) \cdot \sum_{i=1}^{N_A} P_I^i(A) + P_{sw}) \\ < T_B(u_B) \cdot (\beta_B(u_B) \cdot \textstyle\sum_{i=1}^{N_B} P_F^i(B, u_B) + (1 - \beta_B(u_B)) \cdot \sum_{i=1}^{N_B} P_I^i(B) + P_{sw}) \end{aligned} \tag{1.35}$$

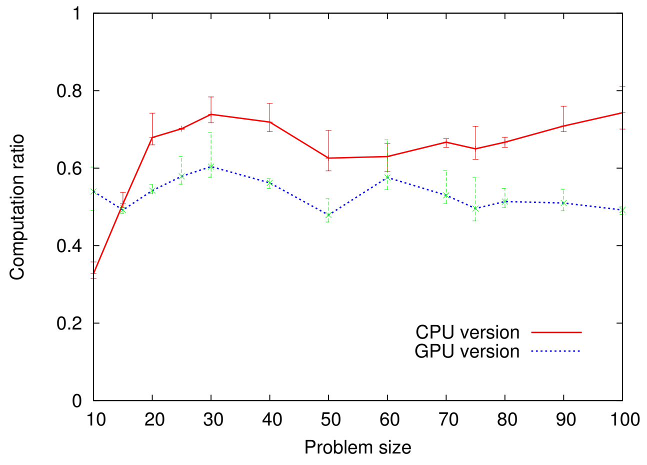

FIGURE 1.6: Computation percentage of the total execution time in function of the problem size for a 17 nodes cluster. Results are statistics from 5 executions.

which finally gives

$$\alpha > \frac{\beta_A(u_A) \cdot \sum_{i=1}^{N_A} P_F^i(A, u_A) + (1 - \beta_A(u_A)) \cdot \sum_{i=1}^{N_A} P_I^i(A) + P_{sw}}{\beta_B(u_B) \cdot \sum_{i=1}^{N_B} P_F^i(B, u_B) + (1 - \beta_B(u_B)) \cdot \sum_{i=1}^{N_B} P_I^i(B) + P_{sw}} \tag{1.36}$$

And we can express the constraint of the energy frontier between clusters A and B only in terms of: (1) the number of used nodes, (2) the powers of the nodes (with respective computing units u_A and u_B) at full load and in idle/communication state, and (3) the power of the network switch(es). Most of those information can be retrieved by small generic benchmarks, except for the $\beta()$ values which require at least one execution of the application, as discussed above.

Now, if we consider a single cluster and we just want to establish whether it is interesting to use its embedded GPUs or not, we can simplify the general notations with $N_A = N_B = N$, $P_F^i(A, u_A) = P_F^i(GPU)$, $P_F^i(B, u_B) = P_F^i(CPU)$, $P_I(A) = P_I(B) = P_I$, $\beta_A(u_A) = \beta(GPU)$ and $\beta_B(u_B) = \beta(CPU)$. According to (1.36), we obtain:

$$\alpha > \frac{\beta(GPU) \cdot \sum_{i=1}^{N} P_F^i(GPU) + (1 - \beta(GPU)) \cdot \sum_{i=1}^{N} P_I^i + P_{sw}}{\beta(CPU) \cdot \sum_{i=1}^{N} P_F^i(CPU) + (1 - \beta(CPU)) \cdot \sum_{i=1}^{N} P_I^i + P_{sw}} \tag{1.37}$$

Here again, if the power dissipation between the similar nodes in a same cluster is negligible, we have $P_F^i(X) = P_F^j(X) = P_F(X)$, $P_I^i = P_I^j = P_I$,

$\forall i, j \in \{1, ..., N\}$, and we obtain the simplified version:

$$\alpha > \frac{\beta(GPU) \cdot P_F(GPU) + (1 - \beta(GPU)) \cdot P_I + \frac{P_{sw}}{N}}{\beta(CPU) \cdot P_F(CPU) + (1 - \beta(CPU)) \cdot P_I + \frac{P_{sw}}{N}} \tag{1.38}$$

1.7 Synchronous and asynchronous distributed PDE solver

Our test application is an iterative PDE solver using the multisplitting-Newton algorithm together with an inner linear solver. We recall that iterative methods perform successive approximations toward the solution of a problem (notion of convergence) whereas direct methods give the exact solution within a fixed number of operations. Although iterative methods are generally slower than direct ones, they generally present the advantage of being less memory consuming. Moreover, they are often the only known way to solve some problems and they are also the only way to express asynchronous algorithms.

Our PDE solver is designed to solve a 3D transport model, which simulates the evolution of the concentrations of chemical species in shallow waters. A system of advection-diffusion-reaction (ADR) has the following form:

$$\frac{\partial c}{\partial t} + A(c, a) = D(c, d) + R(c, t) \tag{1.39}$$

where c is the unknown vector of the species concentrations, $A(c, a)$ is the vector related to the advection, and $D(c, d)$ is the vector of the diffusion. a is the field of local velocities in the liquid, and d is the matrix of the diffusion coefficients; those last two data are assumed to be known in advance (a may be for example computed by a hydrodynamic model). Finally, $R(c, t)$ represents the chemical reactions between the species.

In the following, we consider the problem in three spatial dimensions and with two chemical species. In that case, (1.39) can be written as a system of two PDEs:

$$\begin{pmatrix} \frac{\partial c_1}{\partial t} \\ \frac{\partial c_2}{\partial t} \end{pmatrix} + \begin{pmatrix} \nabla c_1 \times a \\ \nabla c_2 \times a \end{pmatrix} = \begin{pmatrix} \nabla \cdot ((\nabla c_1) \times d) \\ \nabla \cdot ((\nabla c_2) \times d) \end{pmatrix} + \begin{pmatrix} R_1(c, t) \\ R_2(c, t) \end{pmatrix} \tag{1.40}$$

The coupling between the two equations comes from the reaction term R.

1.7.1 Computational model

First of all, (1.40) is transformed into a discrete time (Euler method) and discrete space (second order finite differences) ODE system of the form:

$$\frac{\mathrm{d}x(t)}{\mathrm{d}t} = f\left(x(t), t\right) \tag{1.41}$$

where x is the mesh of points where the concentrations are computed and f is the non-linear function modeling the ADR.

By the use of an implicit temporal integration, this ODE system becomes:

$$\frac{x(t) - x(t-h)}{h} = f\left(x(t), t\right) \tag{1.42}$$

where h is a fixed time step. This equation can then be rewritten as:

$$F\left(x(t), x(t-h), t\right) = x(t-h) - x(t) + hf\left(x(t), t\right) \tag{1.43}$$

and the final problem is to solve $F\left(x(t), x(t-h), t\right) = 0$, which can be reformulated in a simpler way by $F\left(x(t), C(t-h)\right) = 0$, where $C(t-h)$ represents the constant terms of F at time t.

Using the Newton method, we obtain an iterative scheme to compute $x(t)$:

$$x^{k+1}(t) = x^k(t) - F'^{-1}\left(x^k(t)\right) F\left(x^k(t), C(t-h)\right) \tag{1.44}$$

where $x^i(t)$ is the i^{th} iterate of $x(t)$ and $F'\left(x^k(t)\right)$ is the Jacobian matrix of $F\left(x^k(t), C(t-h)\right)$. The equation can be reformulated as:

$$F'\left(x^k(t)\right)\left(x^{k+1}(t) - x^k(t)\right) = -F\left(x^k(t), C(t-h)\right) \tag{1.45}$$

Solving that equation requires to solve a linear system at each iteration. However, it can be noticed that in the ADR problem, the Jacobian matrix is sparse and its non-zero terms are scattered just on a few diagonals. The method used to get a parallel version of that algorithm is the *multisplitting-Newton* scheme.

1.7.2 Multisplitting-Newton algorithm

There are several methods to solve PDE problems, each of them including different degrees of synchronism/asynchronism. The method used in this test application is the multisplitting-Newton [7] which allows for a rather important level of asynchronism. Indeed, it is important to validate our model to get an application which can work either in synchronous or asynchronous mode.

In the computational model described above, the size of the simulation domain can be huge and the domain is then distributed among several nodes of a cluster. Each node solves a part of the resulting linear system and sends the relevant updated data to the nodes that need them. The parallel algorithmic scheme of the method is as follows:

- Initialization:
    - Formulation of the problem under the form of a fixed point problem: $x = T(x), x \in \mathbb{R}^n$ where $T(x) = x - F'(x)^{-1}F(x)$ and F' is the Jacobian

  - We get $F' \times \Delta x = -F$ with F' a sparse matrix
  - F' and F are distributed over the computing units

- Iterative process:

  - Each unit computes a different part of Δx using the Newton algorithm over its sub-domain as can be seen in Fig. 1.7
  - The local elements of x are directly updated with the local part of Δx
  - The non-local elements of x come from the other units by messages exchanges
  - F is updated using the entire vector x

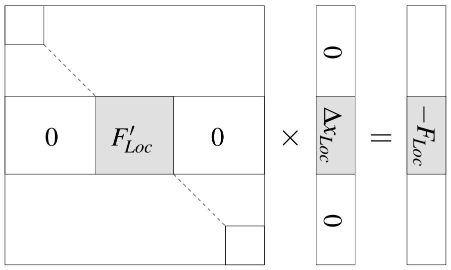

FIGURE 1.7: Local computations associated to the sub-domain of one unit

1.7.3 Inner linear solver

The method described above is a two-stage algorithm in which a linear solver is needed in the inner stage. In fact, most of the time of the algorithm is spent in that linear solver. This is why, in the context of our comparison between CPU and GPU nodes, it is that part of the computations that has been deported on the GPUs. Due to their regularity, those treatments are very well suited to the SIMD architecture of the GPU. Hence, on each computing unit, the linear computations required to solve the partial system are performed on the local GPU while all the algorithmic control, non-linear computations and data exchanges between the units are done on the CPU.

The linear solver has been implemented both on CPU and GPU, using the biconjugate gradient algorithm (see [19] for further details). This linear solver was chosen because it performs well on non-symmetric matrices (on both convergence time and numerical accuracy), it has a low memory footprint, and it is relatively easy to implement on a GPU.

GPU implementation

Several aspects are critical in a GPU: the regularity of the computations and the memory which is of limited amount and the way the data are accessed. In order to reduce the memory consumption of our sparse matrix, we have used a compact representation, depicted in Fig. 1.8, similar to the DIA (diagonal) format [18] in BLAS [13], but with several additional advantages. The first one is the regularity of the structure which allows us to do coalescent memory accesses most of the time. The second one is that it provides an efficient access to the matrix itself as well as to its transpose (using simple array index computations), which is a great advantage as the transpose is required in the biconjugate gradient method.

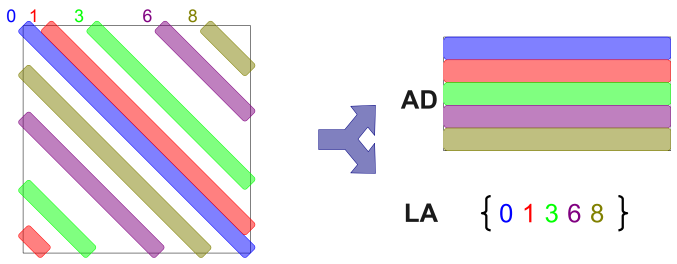

FIGURE 1.8: Compact and regular sparse matrix representation

In order to be as efficient as possible, the shared memory has been used as a cache memory whenever it was possible in order to avoid the slower accesses to the global memory of the GPU. The different kernels used in the solver are divided to make as much as possible data reuse at each call, minimizing by this way the transfers between the global memory and the registers. To get full details on those kernels, the reader should refer to [19].

1.7.4 Asynchronous aspects

It is quite obvious that over the last few years, the classical algorithmic schemes used to exploit parallel systems have shown their limit. As the most recent systems are more and more complex and often include multiple levels of parallelism with very heterogeneous communication links between those levels, the synchronous nature of the schemes presented previously has become a major drawback. Indeed, synchronizations may noticeably degrade performances in large or hierarchical systems, even for local systems.

For a few years now, asynchronous algorithmic schemes have emerged [10, 2, 9, 3, 15, 6], and although they cannot be used for all problems, they are efficiently usable for a large part of them. In scientific computing, asynchronism can only be expressed in iterative algorithms, as already mentioned. This condition has strongly motivated the nature of our PDE solver.

The asynchronous feature consists in suppressing any idle time induced by the waiting for the dependency data to be exchanged between the computing units of the parallel system. Hence, each unit performs the successive iterations on its local data with the dependency data versions it owns at the current time. The main advantage of this scheme is to allow for an efficient and implicit overlapping of communications by computations. On the other hand, the major drawbacks of asynchronous iterations are a more complex behavior, which requires a specific convergence study, and a larger number of iterations to reach the convergence. However, the convergence conditions in asynchronous iterations are verified for numerous problems and, in many computing contexts, the time overhead induced by the additional iterations is largely compensated by the gain in the communications [4, 8]. In fact, as soon as the frequency of communications relatively to computations is high enough and the communication costs are larger than local accesses, an asynchronous version of an application may provide better performances than its synchronous counterpart.

In the asynchronous version of our PDE solver, the exchanges of parts of the vector x are performed asynchronously. One synchronous global exchange is still required between each time step of the simulation, as illustrated in Fig. 1.9.

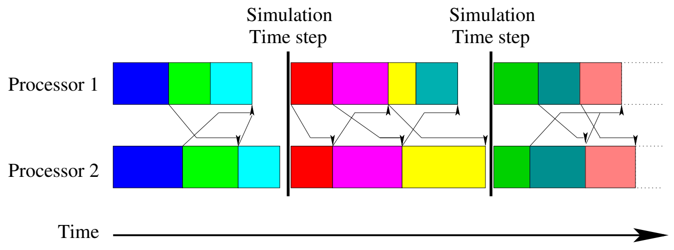

FIGURE 1.9: Asynchronous iterations inside each time step of the computation

At the practical level, the main differences with the synchronous version lie in the suppression of some barriers and in the way the communications between the units are managed. Concerning the first aspect, all the barriers between the inner iterations inside each time step of the simulation are suppressed. The only remaining synchronization is the one between each time step as pointed out above.

The communications management is a bit more complex than in the synchronous version as it must enable sending and receiving operations at any time during the algorithm. Although the use of non-blocking communications seems appropriate, it is not sufficient, especially concerning receptions. This is why a multi-threaded programming is required. The principle is to use separated threads to perform the communications, while the computations are continuously done in the main thread without any interruption, until convergence detection. In our version, we use non-blocking sends in the main thread and an additional thread to manage the receptions. It must be noted that in order to be as reactive as possible, some communications related to the control of the algorithm (the global convergence detection) may be initiated directly by the receiving thread (for example to send back the local state of the unit) without requiring any process or response from the main thread. Subsequently to the multi-threading, mutexes are necessary to avoid concurrent accesses to data and variables.

Another difficulty brought by the asynchronism comes from the convergence detection. Some specific mechanisms must replace the simple global reduction of local states of the units to ensure the validity of the detection [5]. The most general scheme may be too expensive in some simple contexts such as local clusters. So, when some characteristics of the system are guaranteed (such as bounded communication delay), it is often more pertinent to use a simplified mechanism whose efficiency is better and whose validity is still ensured in that context. Although both general and simplified schemes of convergence detection have been developed for this study, the performances presented in the following sections are those of the simplified scheme, which gave the best results.

1.8 Experimental validation

In this part, a series of experiments is presented which evaluates the pertinence and accuracy of our model at both node and cluster levels.

1.8.1 Testbed introduction and measurement methodology

All the experiments presented below have been performed on a homogeneous cluster of 16 machines with Intel Nehalem CPUs (4 cores + hyperthreading) running at 2.67GHz, 6GB RAM and one NVIDIA GeForce GTX 480 card with 768MB RAM. The OS is Linux Fedora with CUDA 3.0.

In the following experiments, the tested application is the three-dimensional PDE solver described in Section 1.7. Such an application is quite representative of the scientific applications run on a cluster. The results related to that application are an average of five consecutive executions. The small generic benchmarks used to extract the model parameters from the test platform are very simple floating point computations performed either on a specified number of CPU cores or on one CPU core and one GPU. For each benchmark, the conserved power is the maximal one obtained during a period of 30s. In Table 1.1 are provided the results obtained with those benchmarks for the test platform.

P_{sw}	34.00			
node id	$P_F(CPU1)$ (1 core)	$P_F(CPU2)$ (2 cores)	$P_F(GPU)$	P_I
1	167	167	228	146
2	159	159	228	128
3	159	167	218	133
4	167	174	228	139
5	167	167	228	139
6	159	167	218	133
7	174	182	238	152
8	167	174	238	146
9	152	167	218	133
10	174	182	228	146
11	152	159	228	133
12	167	174	238	139
13	167	167	218	139
14	159	167	228	139
15	174	182	238	159
16	182	190	249	159

TABLE 1.1: Powers (Watts) of the switch and the 16 nodes of the cluster

1.8.2 Node level

At the node level, we consider the nodes of the cluster separately and we apply our simplified model to compare the PDE solver executions with or without the GPU. In that context, the columns of interest in Table 1.1 are only $P_F(CPU1)$ and $P_F(GPU)$. Indeed, our test application uses only one CPU core when run on a single node and either zero or one GPU. In Table 1.2 are presented, for every node of the cluster, the respective estimations of the α limits deduced from (1.25), the observed ones, and the estimation/observation ratios.

Node id $\longrightarrow$	1	2	3	4	5	6	7	8
Estimated α	1.365	1.434	1.371	1.365	1.365	1.371	1.368	1.425
Ratio	1.090	1.136	1.109	1.166	1.105	1.108	1.106	1.128
Observed α	1.252	1.262	1.236	1.171	1.235	1.238	1.237	1.264
Node id $\longrightarrow$	9	10	11	12	13	14	15	16
Estimated α	1.434	1.310	1.500	1.425	1.305	1.434	1.368	1.368
Ratio	1.110	1.070	**1.194**	1.064	1.031	1.148	1.111	1.085
Observed α	1.292	1.225	1.256	1.339	1.266	1.249	1.231	1.261

TABLE 1.2: Model estimates compared to experimental observations for every node of the cluster.

It can be seen that the estimations are quite close to the observed frontiers. However, a global trend of overestimation can be observed in the whole set of estimates (all the ratios are greater than one). Although that global bias is quite reasonable (mean bias around 0.11), the most important ones are quite significant as they reach a little more than 19% for node 11 (in bold face). In order to give a better idea of the error made in that extreme case, the computing and energy ratios are depicted in function of the problem size for node 11 in Fig. 1.10. Such bias may lead to wrong choices near the frontier.

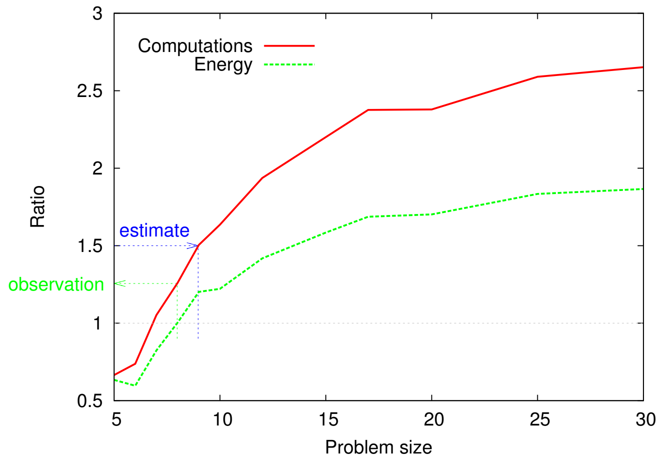

FIGURE 1.10: Computing and energy ratios of the CPU version over the GPU one for the ADR problem on node 11, in function of the problem size (3D cubic domain $x \times x \times x$).

Although a part of that bias may be explained by the fair accuracy of the energy measures, it is not sufficient to explain the whole bias. In fact, the main cause of error comes from the approximations made in the simplified model. Typically, in the case of our test application, the GPU is not used to perform all the computations but only to solve the inner linear systems. Thus, it is only used during a fraction of the total execution time, and considering $P_F(GPU)$ for the entire execution in (1.25) results in an evident overestimation of the required power, and thus of α.

So, when it is possible, evaluating at least the main ratios $\beta(u_i)$ for the target application can substantially enhance the final estimation. As an example, in our test application, we evaluate $\beta(GPU)$ and we obtain the following equation for the estimation of α:

$$\alpha \geq \frac{\beta(GPU)P_F(GPU) + (1 - \beta(GPU))P_F(CPU1)}{P_F(CPU1)} \tag{1.46}$$

To obtain $\beta(GPU)$, we have measured the percentage of GPU usage in the entire execution time. Unfortunately, that measure cannot be used directly for two reasons. The first one is that unlike the full CPU version, the percentage of GPU usage strongly varies with the problem size. In fact, for small problem sizes (between 5^3 and 30^3 here), the GPU is not fully loaded and its usage time does not evolve as fast as the CPU usage with the problem size. We have then to consider an average from a few problem sizes. The second reason is that the energy consumption does not follow discontinuous variations and once a high power has been reached, the return to the lowest power takes some time, as shown in Fig. 1.11 with a simple benchmark.

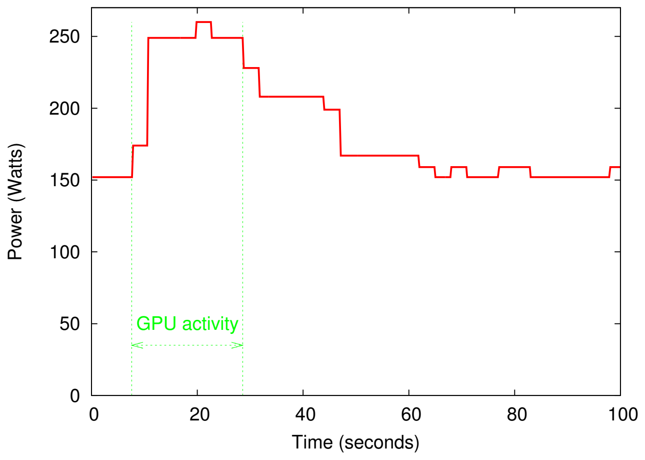

FIGURE 1.11: Power consumption during and after a GPU usage of 20s on a single node.

So, to compensate for the slow power decrease after high consumption periods, the measure has to be weighted according to the mean power level after the GPU usage. As the power decrease has a typical step at the middle value between full GPU usage and idle power, a good weighting is the middle value between the measured GPU usage percentage and the total time (i.e 1), that is:

$$\beta(GPU) = \frac{\beta_{measured}(GPU) + 1}{2}$$

Finally, from (1.46) and the deduced $\beta(GPU)$ for every single node, we obtain a mean ratio between the predicted α and the observed ones of 0.985 and minimal and maximal values respectively of 0.934 and 1.042. Obviously, this is much better than our first estimations as the standard deviation of the ratios is smaller and the ratios are globally centered around one.

In conclusion, the previous experiments performed on single nodes have shown that the simplified node level model has a slight bias. That bias can be discarded by performing a deeper study and/or benchmark of the target application. However, as discussed in Section 1.5.2, these corrections require application-dependent benchmarks which are not always possible nor desirable to do. Hence, given its relevance, its rather limited estimation bias and its simple practical process, our simplified model can be used in most cases as a good indicator for energy comparisons.

1.8.3 Cluster level: asynchronous mode

In the following experiments, we consider the entire cluster of 16 machines. In the asynchronous case, the columns $P_F(CPU2)$ and $P_F(GPU)$ in Table 1.1 are used. The 2-cores version corresponds to the number of threads required in the asynchronous version of the PDE solver.

Here, we had the possibility to get the powers of all the nodes in the cluster. However, for homogeneous clusters, if the power information is available only on one node, our model can be adapted by considering that all the $P_F^i()$ are identical. Nonetheless, the user must be aware that this is an approximation which may induce an additional bias in the final predictions of the model.

According to (1.29), we can deduce that the use of GPUs on that cluster will be interesting from the energy point of view as soon as we have:

$$\begin{aligned} \alpha &> \frac{\sum_{i=1}^{16} P_F^i(GPU) + P_{sw}}{\sum_{i=1}^{16} P_F^i(CPU2) + P_{sw}} \\ &> \frac{3669+34}{2745+34} \\ &> 1.332 \end{aligned} \tag{1.47}$$

So, the GPU version must be at least around 33.2% faster than the CPU one to provide an energy gain on this cluster.

This estimation is confirmed by the experiments. Indeed, it can be seen on Fig. 1.12, depicting the computing and energy ratios of the CPU version over the GPU one in function of the problem size for the entire cluster (16 nodes), that the estimated speedup α closely matches the frontier at which the GPU version becomes more interesting than the CPU from the energy point of view.

1.8.4 Cluster level: synchronous mode

In that last context, the columns $P_F(CPU1)$, $P_F(GPU)$ and P_I of Table 1.1 are relevant. The 1-core version corresponds to the synchronous version of the PDE solver, in which there is only one main thread.

As discussed before, in the synchronous case, generic benchmarks alone are not sufficient to be able to deduce the constraint over α. An evaluation of the percentage of time performed at full load during the execution (that is during computations) is required. The following measures of the $\beta()$ have been performed on the target cluster with the target application with a problem size of $30 \times 30 \times 30$.

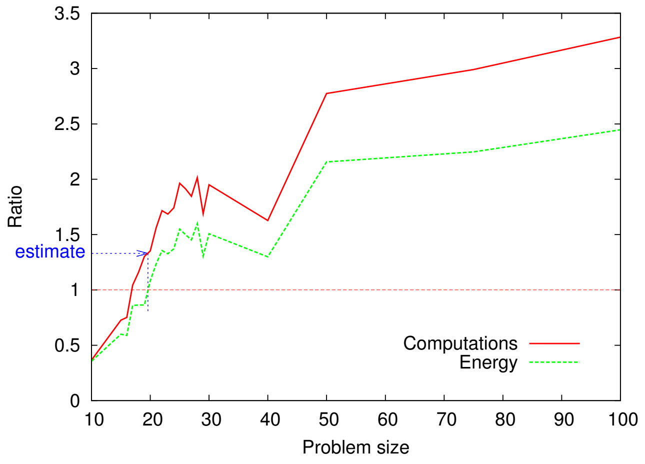

FIGURE 1.12: Computing and energy ratios of the CPU version over the GPU one for the ADR problem on the 16 nodes, in function of the problem size (3D cubic domain $x \times x \times x$).

$\beta(CPU)$	$\beta(GPU)$
0.749	0.602

TABLE 1.3: Computation ratios of the total execution time for the CPU and GPU versions of the ADR problem with a problem size of $30 \times 30 \times 30$.

In fact, complementary measures have shown that the values of $\beta(CPU)$ and $\beta(GPU)$ within a range of problem sizes from 10^3 to 100^3 do not vary very much, and their respective averages are quite close to the ones given in Table 1.3.

According to (1.37), we deduce that it is worth using the GPUs of the cluster as soon as we have:

$$\begin{aligned}\alpha &> \frac{\beta(GPU)\sum_{i=1}^{16} P_F^i(GPU) + (1-\beta(GPU))\sum_{i=1}^{16} P_I^i + P_{sw}}{\beta(CPU)\sum_{i=1}^{16} P_F^i(CPU) + (1-\beta(CPU))\sum_{i=1}^{16} P_I^i + P_{sw}} \\ &> \frac{0.602 \cdot 3669 + (1-0.602) \cdot 2263 + 34}{0.749 \cdot 2646 + (1-0.749) \cdot 2263 + 34} \\ &> 1.217\end{aligned}$$

That is, the GPU version should be less energy consuming as soon it is around 21.7% faster than the CPU one.

Similarly to the node level context, there is a slight bias in that estimation with respect to the experimental observation, as can be seen in Fig. 1.13. That bias is around 5.9% under the actual observed value of α (1.293) and may therefore lead to wrong choices of hardware near the energy efficiency frontier between CPU and GPU versions.

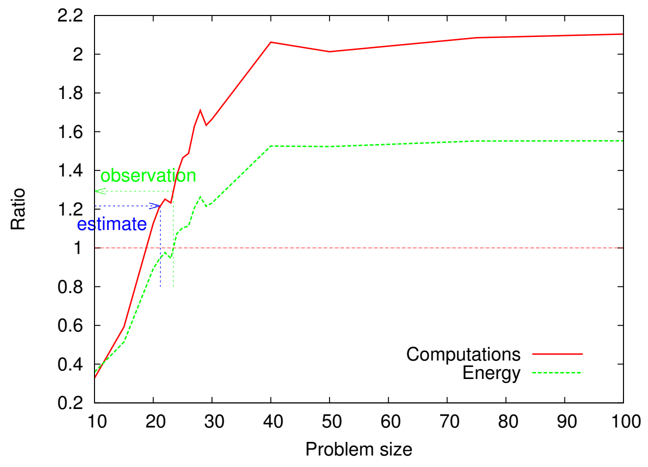

FIGURE 1.13: Computing and energy ratios of a synchronous CPU version over its GPU counterpart for the ADR problem in function of the problem size (3D cubic domain $x \times x \times x$).

Here also, that bias can be explained by the slow power decrease after high consumptions, which are not taken into account in that simplified version. This results in significant underestimations of $\beta(GPU)$ and $\beta(CPU)$, leading to a global underestimation of α. There are two ways for correcting the estimation, which lead to slightly different results but with almost the same final accuracy (final bias under 1%). The former is also the simplest one and the most convenient in practice as it consists in taking into account the slow power decrease directly at the level of $\beta(GPU)$ and $\beta(CPU)$. For the same reasons as in Section 1.8.2, we consider the following corrections:

$$\begin{aligned} \beta(CPU) &= \tfrac{\beta_{measured}(CPU)+1}{2} &= 0.875 \\ \beta(GPU) &= \tfrac{\beta_{measured}(GPU)+1}{2} &= 0.801 \end{aligned} \tag{1.48}$$

and we obtain:

$$\begin{aligned} \alpha &> \frac{0.801 \cdot 3669 + (1 - 0.801) \cdot 2263 + 34}{0.875 \cdot 2646 + (1 - 0.875) \cdot 2263 + 34} \\ &> 1.301 \end{aligned}$$

which is only 0.6% over the observation.

The second possible correction is a bit more complex and require more information about the application. The idea is to get closer to the complete model described in Section 1.5.1. In fact, in the synchronous version of the application, we can distinguish three kinds of activities during the execution: communication (considered similar to idle), CPU-only computations, and GPU computations. This leads to splitting the initial $\beta(GPU)$ into $\beta(GPU)$ on one side and $\beta(CPUo)$ (computations only on CPU) on the other side, leading to the following reformulation of the condition over α:

$$\begin{aligned}\alpha \quad > \frac{1}{\gamma} \quad & \Bigg(\beta(GPU)\sum_{i=1}^{16} P_F^i(GPU) + \beta(CPUo)\sum_{i=1}^{16} P_F^i(CPU1) \\ & + (1-\beta(GPU)-\beta(CPUo))\sum_{i=1}^{16} P_I^i + P_{sw}\Bigg)\end{aligned}$$

where γ is the total energy consumption of the CPU version of the application: $\gamma = \beta(CPU)\sum_{i=1}^{16} P_F^i(CPU) + (1-\beta(CPU))\sum_{i=1}^{16} P_I^i + P_{sw}$. The final estimation is obtained after having applied the same slow power decrease correction as above to the different $\beta()$. The result is:

$$\begin{aligned}\alpha \quad & > \quad \frac{0.565 \cdot 3669 + 0.737 \cdot 2646 + (1-0.565-0.737) \cdot 2263 + 34}{0.875 \cdot 2646 + (1-0.875) \cdot 2263 + 34} \\ & > \quad 1.281\end{aligned}$$

which is 0.9% under the observation.

Finally, as both corrections seem to give very close lower and upper estimations, it may be interesting to use both of them to get a small fuzzy area around the real frontier of energy efficiency between CPU and GPU versions. However, this point has yet to be confirmed with other scientific applications and should be fully investigated in future works.

1.9 Discussion on a model for hybrid and heterogeneous clusters

As described in Section 1.5.1 and Section 1.6, the proposed model directly includes the potential hardware heterogeneity of the used parallel system. We have seen in Section 1.8 that this model allows us to compare different systems or configurations of a single system from the energy point of view. However, the possibility to compare different operating modes of a same application has not been discussed. In particular, it is relevant to compare the efficiencies of the synchronous and asynchronous modes when both of them are available for a given application.

In fact, in previous works [11, 12], we performed a whole set of experiments with the ADR application on a heterogeneous cluster composed of two homogeneous clusters (respectively 14 and 17 machines). In those experiments, we compared the computing and energy performances of the synchronous and asynchronous versions of our test application.

As can be seen in Fig. 1.14, the frontier between the two versions is not linear and an accurate model is necessary to be able to determine which operating mode is preferable for a given context of use (number of used nodes in each cluster).

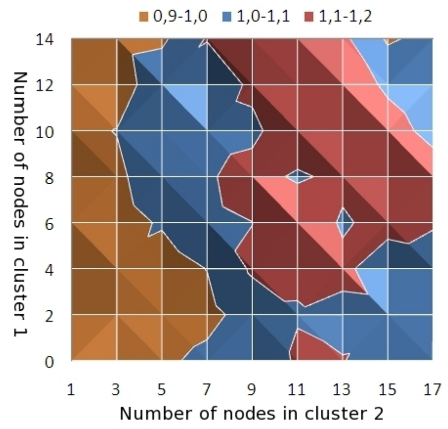

FIGURE 1.14: Speedup of asynchronous vs synchronous versions of the ADR application with a heterogeneous GPU cluster

The model proposed in this chapter can be quite easily extended to allow for the comparison of operating modes. Indeed, from (1.16) in Section 1.5.1, it can be seen that the energy can always be expressed as a product of the execution time and the average electrical power consumed by the application during this time. So, when comparing two contexts of use, whatever those contexts are, it is always possible to deduce a constraint over α that only depends on the respective powers corresponding to those contexts. Thus, it is possible to say that context 2 will be preferable to context 1 as soon as:

$$\alpha = \frac{T(\text{context 1})}{T(\text{context 2})} \quad > \quad \frac{\overline{P}(\text{context 2})}{\overline{P}(\text{context 1})} \tag{1.49}$$

And the comparison of operating modes in the same hardware context can be expressed by:

$$\alpha = \frac{T(\text{sync})}{T(\text{async})} \quad > \quad \frac{\overline{P}(\text{async})}{\overline{P}(\text{sync})} \tag{1.50}$$

Hence, with the formulations of the energy consumptions given in Sections 1.6.1 and 1.6.2, the generic benchmarks on all the nodes of the heterogeneous cluster, and a few application benchmarks, it is possible to provide estimations of α for each heterogeneous configuration. Finally, a comparative map as depicted in Fig. 1.14 could be entirely deduced from only a few executions of the application.

Current researches are developed on this topic and a complete experimental study should be proposed in a near future.

1.10 Perspectives: towards multi-kernel distributed algorithms and auto-adaptive executions

In this chapter, a complete feedback of our experience in practical and theoretical works over the energy aspects of parallel computing has been proposed as a starting point. Then, our experimental process has been described together with the main experimental issues that can be encountered when measuring energy consumptions. Finally, a complete model linking the computing and energy performances has been presented and experimentally validated with a representative scientific application. Also, possible extensions to various contexts have been discussed.

The simplest way to use the model proposed in this chapter is when a user has to choose between several execution contexts and modes. The user only has to get the required information by executing a small set of benchmarks and, according to the results yielded by the model, the user can choose the most suited environment with respect to his needs.

Nevertheless, in many situations, that choice protocol may not be adapted. In fact, the user may not be a specialist in computer science and thus not be able to perform the required benchmarks nor to use the program implementing the model computations. Moreover, it is also probable that the user would not want to do such tasks which are outside his/her main domain of work.

So, it is desirable to provide a system that would implement the entire protocol. However, it is not interesting to insert that choice protocol directly into the application. First of all, this is not possible when the source code of the application is not available. Moreover, even in the opposite case, it may be quite complex and time-consuming to perform such modifications into the code. And it withdraws the advantage of the application independence of the protocol.

Thus, the best solution seems to be an additional system controlling the application execution. The idea is that the user specifies the optimization criteria for the application execution (execution time and/or energy consumption) on that system. In addition, the execution control system (ECS) needs either a multi-kernel version of the application or a set of different versions, as well as a description of all the available nodes in the parallel system. Provided this, the ECS can automatically run the required generic benchmarks as well as the few application-dependent benchmarks needed to feed the model. It is then able to produce the estimations for the set of possible execution configurations and to choose the optimal one with respect to the user's specification. Finally, the adequate version of the application is executed in the corresponding system and configuration.

The complete design and implementation of the ECS is our priority goal in our future works in the domain.

Bibliography

[1] L. Abbas-Turki, S. Vialle, B. Lapeyre, and P. Mercier. High dimensional pricing of exotic european contracts on a GPU cluster, and comparison to a CPU cluster. In *Parallel and Distributed Computing for Finance (PDCoF09)*, Roma, Italy, May 29 2009.

[2] D. Amitai, A. Averbuch, M. Israeli, and S. Itzikowitz. Implicit-explicit parallel asynchronous solver for PDEs. *SIAM J. Sci. Comput.*, 19:1366–1404, 1998.

[3] J. Bahi. Asynchronous iterative algorithms for nonexpansive linear systems. *Journal of Parallel and Distributed Computing*, 60(1):92–112, January 2000.

[4] J. Bahi, S. Contassot-Vivier, and R. Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Computing*, 31(5):439–461, 2005.

[5] J. Bahi, S. Contassot-Vivier, and R. Couturier. An efficient and robust decentralized algorithm for detecting the global convergence in asynchronous iterative algorithms. In *8th International Meeting on High Performance Computing for Computational Science, VECPAR'08*, pages 251–264, Toulouse, June 2008.

[6] J. Bahi, R. Couturier, K. Mazouzi, and M. Salomon. Synchronous and asynchronous solution of a 3D transport model in a grid computing environment. *Applied Mathematical Modelling*, 30(7):616–628, 2006.

[7] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. *Parallel Iterative Algorithms: from sequential to grid computing.* Numerical Analysis & Scientific Computing Series. Chapman & Hall/CRC, 2007.

[8] Jacques M. Bahi, Sylvain Contassot-Vivier, and Raphaël Couturier. Asynchronism for iterative algorithms in a global computing environment. In *The 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'2002)*, pages 90–97, Moncton, Canada, June 2002.

[9] Z. Bai, V. Migallon, J. Penades, and D.B. Szyld. Block and asynchronous two-stage methods for midly nonlinear systems. *Numer. Math.*, 82:1–21, 1999.

[10] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice Hall, Englewood Cliffs, New Jersey, 1999.

[11] S. Contassot-Vivier, T. Jost, and S. Vialle. Impact of asynchronism on GPU accelerated parallel iterative computations. In *PARA 2010 conference: State of the Art in Scientific and Parallel Computing*, Reykjavík, Iceland, June 2010.

[12] S. Contassot-Vivier, S. Vialle, and T. Jost. Optimizing computing and energy performances on GPU clusters: experimentation on a PDE solver. In Jean-Marc Pierson and Helmut Hlavacs, editors, *COST Action IC0804 on Large Scale Distributed Systems,1st Year*. IRIT, 2010. ISBN: 978-2-917490-10-5.

[13] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[14] Agner Fog. Optimizing software in C++: An optimization guide for windows, linux and mac platforms. Technical report, Copenhagen University College of Engineering, sept 2009.

[15] A. Frommer and D. B. Szyld. On asynchronous iterations. *J. Comput. and Appl. Math.*, 123:201–216, 2000.

[16] R. Gonzalez and M. Horowitz. Energy dissipation in general pupose microprocessors. *IEEE Journal of solid-state circuits*, 31(9), September 1996.

[17] N. Govindaraju, S. Larsen, J. gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *ACM/IEEE Conference on Supercomputing (SC'06)*, Tampa, FL, USA, November 11-17, 2006.

[18] Michael A. Heroux. A proposal for a sparse blas toolkit. SPARKER working note #2, Cray research, Inc, 1992.

[19] T. Jost, S. Contassot-Vivier, and S. Vialle. An efficient multi-algorithm sparse linear solver for GPUs. In *Parallel Computing : From Multicores and GPU's to Petascale*, volume 19 of *Advances in Parallel Computing*, pages 546–553. IOS Press, 2010.

[20] X. Ma, M. Dong, L. Zhong, and Z. Deng. Statistical power consumption analysis and modeling for gpu-based computing. In *Workshop on Power Aware Computing and Systems (HotPower'09)*, Big Sky, MT, USA, October 10, 2009.

[21] M. Roufouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M. Sarrafzadeh. Energy-aware high performance computing with graphic processing units. In *Workshop on Power Aware Computing and Systems (HotPower'08)*, San Diego, CA, USA, December 7, 2008.

[22] H. Takizawa, K. Sato, and H. Kobayashi. SPRAT: Runtime processor selection for energy-aware computing. In *Third International Workshop on Automatic Performance Tuning (iWAPT'08), in 2008 IEEE International Conference on Cluster Computing (Cluster 2008)*, Tsukuba, Japan, October 1st, 2008.

[23] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.