Janus2: an FPGA-based Supercomputer for Spin Glass Simulations

The Janus Collaboration[*]

ABSTRACT

We describe the past and future of the Janus project. The collaboration started in 2006 and deployed in early 2008 the Janus supercomputer, a facility that allowed to speed-up Monte Carlo Simulations of a class of model glassy systems and provided unprecedented results for some paradigms in Statistical Mechanics. The Janus Supercomputer was based on state-of-the-art FPGA technology, and provided almost two order of magnitude improvement in terms of cost/performance and power/performance ratios. More than four years later, commercial facilities are closing-up in terms of performance, but FPGA technology has largely improved. A new generation supercomputer, Janus2, will be able to improve by more than one orders of magnitude with respect to the previous one, and will accordingly be again the best choice in Monte Carlo simulations of Spin Glasses for several years to come with respect to commercial solutions.

Categories and Subject Descriptors

J.2 [**Physical Sciences and Engineering**]: Physics

General Terms

Algorithms, Performance, Experimentation

Keywords

FPGA, Large Scale Simulations

1. INTRODUCTION

The Janus supercomputer [3] was designed and assembled five years ago as a big facility for running simulations of lattice systems with discrete variables. The application around which we took most of the architectural design decisions was the Monte Carlo simulations of glassy systems defined on regular lattices. Spin glasses [1] are a wide category of prototypical glassy systems. Spin variables, taking a small set of discrete values (e.g. two states, *up* and *down*) sit at the nodes of a regular D-dimensional lattice, and tend to take the same value of their neighbors if the coupling defined on the edge connecting the corresponding nodes are positive (and to misalign if the coupling is negative). Coupling on nodes are fixed during the simulation and taken to be positive or negative at random throughout the lattice (quenched disorder configuration). The deceivingly simple rules (we will briefly describe them later), governing the evolution of a single node, reflects in a very complex collective dynamics, especially when one deals with very large lattice (order 10^6 nodes), as required to compute results that can be compared with experiments on glassy materials.

One striking property of glassy materials – shared by the systems by means of which we model them – is their extremely slow relaxation dynamics. At low temperature (below the *critical temperature*) glasses remains out of equilibrium at all relevant experimental time scales. This translates into (relatively) long-lasting experiments, to which we must compare numerical results.

One usually relates the basic step in the simulated dynamics, the Monte Carlo step (the trial of one spin-reversal on all the nodes of the lattice), to the average spin-flip time in real samples, which is estimated to be order 1 ps. Some real experiments on materials span a time scale of order 1 second and more, corresponding to 10^{12} complete lattice updates in the simulated dynamics. If one has to get in touch with relevant length scales (besides the time scale mentioned), the lattice size cannot be too small (actually, a practical recipe to tailor lattice sizes as a function of the programmed simulation time, in order to correctly reproduce the out-of-equilibrium behavior, has been one of the main results of our investigation [2]). For a three-dimensional cubic lattice, a number of sites of 100^3 is sufficient to study the system up to experimentally relevant time scales. The simulation program then consists then in some 10^{18} spin-flip trials. In addition, simulation must be repeated on several copies of the system, to cope with finite size effects; in off-equilibrium computations, one usually deals with about 100 different independent samples to minimize the effect of one particular disorder configuration, amplifying the computational burden to 10^{20} spin-flip trials. However, this is not the case for *equilibrium* simulations, in which one deals with *thousands* of copies of smaller systems; however the computational effort is of the same order of magnitude in both cases.

At the time the project started, available commercial CPU

[*] See the Additional Authors section for a detailed collaboration member list

ICS2012 Workshop "Future HPC systems: the Challenges of Power-Constrained Performance" Venice, Italy, 2012

technology made it possible to develop simulation codes that partially exploited the (in principle very large) available parallelism; a large optimization effort involving simultaneous handling of different sites, wise memory organization (multi-spin coding), vectorization (SSE instructions, two- or quad-core parallel implementation) resulted in a spin-flip time of order 1 ns. A cluster of order 100 CPUs would then need some 10^9 seconds (that is, more than 30 years) to complete the simulation campaign described above.

The advent of Janus marked a substantial progress: Janus deploys 256 processors built with state-of-the-art FPGAs. The simple dynamical rules governing a single spin variable can be easily implemented as a small block of logical rules; a single FPGA hosts up to 1000 single-spin-flip engines. With a conservative clock frequency of 62.5 MHz, one Janus node has a 16 ps single spin-flip time. A simulation program as the one outlined above can be completed in just a few months, making it a viable option.

In more recent years, progress in computer architectures pushed towards increasing parallelism; on one side, we have many-core processors with intrinsics on sets of wider registers; also, GPU technology provides processors with thousands of streaming processors. Careful optimization of our applications for these architectures increases performance by one order of magnitude with respect to standard architectures, coming close to Janus performances, but still not making it possible for e.g. a graduate student to complete a large simulation campaign during its PhD studies.

On the other side, a performance increase that is compatible with Moore's law is inevitably going to finally close-up and compete with Janus on spin glass applications. Besides, FPGA technology improved too, and larger and faster programmable devices will be shortly available on the market; we foresee that a new generation FPGA-based supercomputer, named Janus2, will be able to outperform its ancestor by at least two order of magnitude, guaranteeing for itself a long lifetime in terms of absolute performance and cost/performance and power/performance ratios in the simulation of spin glass systems.

In what follow, we briefly describe a typical spin glass model and the structure of a typical Monte Carlo simulation and discuss why its implementation is more effective on FPGAs than on conventional CPUs (and GPUs).

We then describe the hardware and software architecture of Janus, and give performance an power consumption comparison figures with respect to other systems available at the time the project started and available today.

We finally describe the new Janus2 supercomputer, that is at an early architectural design stage, and discuss expected performance and expected its expected lifetime as a top speed Carlo engine for spin glasses.

2. A SAMPLE SPIN GLASS APPLICATION

We start by introducing a sample application and describe a typical computational kernel, discussing its features and implications for an FPGA and CPU based implementation.

The three-dimensional Edwards-Anderson model [4] is easily described in terms of the energy of the system:

$$E = -\sum_{\langle ij \rangle} \sigma_i J_{ij} \sigma_j; \tag{1}$$

where σ_i are L^3 spin variables (modeling magnetic moments at atomic lattice sites) taking values $+1$ (spin up) and -1 (spin down), sitting at the nodes of a three-dimensional cubic lattice of linear size L. J_{ij} are the strengths of the interaction (couplings) along the edges connecting nearest-neighbor nodes; a positive J_{ij} favors alignment of the spins of the two neighboring nodes; a negative value favors misalignment. The sum in (1) spans all pairs of nearest neighbors.

The values in the set of couplings J_{ij} is usually extracted from a distribution with zero mean and unit variance; we consider the case in which the couplings are $+1$ or -1 with probability 0.5 (binary model). A set of $3L^3$ J_{ij} is a *sample* of the system; the couplings in a sample remain fixed and do not participate in the dynamics;

The *local energy* of a single spin, say σ_k at site k, is determined by the interaction with its six neighbors only (x,y,z one lattice-site steps in the three directions of the lattice):

$$\begin{aligned} \epsilon(\sigma_k) &= -\sigma_k \phi_k; & (2)\\ \phi_k &= \sum_{j=k\pm x, k\pm y, k\pm z} J_{kj}\sigma_j \, , & (3) \end{aligned}$$

for a given configuration of its neighbors σ_j, it's a two valued function, taking values $\epsilon(+1) = -\phi_k$ and $\epsilon(-1) = \phi_k$. The *local field* ϕ_k is determined only by nearest neighbors of σ_k.

If we make the assumption that the spin σ_k is always at equilibrium with its nearest spins (the local field) at a given temperature T, the probabilities of the spin to be *up* or *down* is given by the Boltzmann-Gibbs distribution:

$$P(\sigma_k = \pm 1) = \frac{\exp\left[\pm\beta\phi_k\right]}{\exp\left[\beta\phi_k\right] + \exp\left[-\beta\phi_k\right]} \, , \tag{4}$$

with $\beta = 1/T$ the *inverse temperature* (we take the Boltzmann constant k_{B} to be one in our units). This defines the *Heat-Bath* algorithm (see ref. [5] for a review on dynamic Monte Carlo algorithms): at any given time, we may decide if the spin σ_k is up or down by comparing the probability to be up with a (pseudo-)random number ρ extracted uniformly in the interval $[0, 1)$.

The recipe for the simulation of the dynamics of a single sample of the model (1) is then as follows:

1. Extract the complete J_{ij} configuration (each J_{ij} can be $+1$ or -1 with equal probability.

2. Extract the complete σ_i configuration (each spin can be up or down with equal probability; note that by eq. (4) this implies preparing the system at very high temperatures).

3. Begin a trial spin-flip: pick a site k at random, each site equal probability.

4. compute the local field $\epsilon(\sigma_k)$, eq. (3) and the spin up probability $P(+1)$, eq. (4)

5. Pick a uniformly distributed pseudo-random number $0 \leq \rho < 1$ from your favorite generator.

6. If $\rho < P(+1)$, then put $\sigma_k = +1$, otherwise put $\sigma_k = -1$; end of the trial spin-flip.

7. Repeat from step 3 above as many times as needed.

A Monte Carlo steps in this scheme is a number of trial spin-flip equal to the number of sites in the lattice; it is the duration of the Monte Carlo steps to be compared to the average spin-flip times in experiments, as discussed in the introduction.

Each Monte Carlo step produces a brand new spin configuration on the lattice; at regime, one would expect that the sampled configurations follow the Boltzmann-Gibbs distribution at the given inverse temperature β for the total energy (1) (but this is not actually the case for off-equilibrium simulations of spin glasses, as real equilibrium is never reached, conforming to the behavior of samples of glassy materials; notwithstanding this, the chosen microscopic dynamics still tends to real equilibrium).

A nice property of this scheme is that when one performs a huge number of Monte Carlo steps, the statistical properties of the observables that one measures (averages over all sites and averages over Monte Carlo steps) do not depend anymore on the particular order in which the algorithm visits each lattice sites.[1] It turns out that this property is independent of any particular lexicographic order we could impose on the site-visiting scheme, so it is useful to choose an order, and impose that each site is visited once and only once in each Monte Carlo step, and always in the same order. This brings substantial simplification and makes room for a very efficient exploitation of the internal parallelism, as we will see shortly.

As pointed out earlier, when one completes the necessary N Monte Carlo steps, the simulation must be repeated for a different realization of the disorder configuration, repeating the scheme above from point 1, generating results for many (hundreds, thousands) disorder samples. Yet another degree of replication is needed, however: some properties of the system comes out when comparing two independent simulations of the same sample, starting from independent initial spin configurations. Usually each sample must be simulated twice at least; the two identical copies with independent histories are called replicas.

Let's discuss some of the usual techniques used in coding efficient routines for the problem depicted above. The principal optimization is usually obtained by *multi-spin* coding. In order to compute the local field acting on a spin (step 4), we need to perform a small number of products and sums of two-valued variables. We then turn to the representation:

$$\sigma_k \quad \rightarrow \quad S_k = (1+\sigma_k)/2\,, \tag{5}$$

$$J_{ij} \quad \rightarrow \quad \hat{J}_{ij} = (1+J_{ij})/2\,, \tag{6}$$

$$\phi_k \quad \rightarrow \quad F_k = \sum_j \hat{J}_{kj} \oplus S_j = (6-\phi_k)/2\,, \tag{7}$$

where the products are substituted by logical XORs. There is an obvious advantage: spins and couplings may be arranged in long integer words (say 64-bit), and products of many spin and coupling variables may be carried out by bit-parallel logical operations on machine (e.g. 64-bit) words (this technique also offers huge memory saving). Some complication arises when one has to extract the values of the local field; still, F_k takes only seven integer values between 0 and 6, and only three 64-bit words are needed to accumulate the sums of XOR products in a bitwise fashion. Note that few values for F_k means that $P(+1)$ values (whose computation might in principle be very lengthy) can be precomputed and stored in a look-up table, addressed by the values of F_k.

This efficient representation has two main variants depending on the internal or external level of parallelism involved when filling each multi-spin coded word.

In the *Asynchronous Multi-Spin Coding* (AMSC) approach, one fills bit positions in the long word with spins (or couplings) belonging to the same site of independent samples.

In the *Synchronous Multi-Spin Coding* approach, one fills the bits with spin variables (and couplings) taken from a single sample, choosing of course a set of spins that can be updated simultaneously. A possibility consists in classifying the lattice sites as *black* or *white* following a checkerboard scheme. We see that if we choose to update all the black sites first, and subsequently all the white ones, the particular order in which we visit them is completely irrelevant to the specific outcome, in terms of spin configuration, of the Monte Carlo step. We may then in principle update in parallel all black (white) sites, as they have only white (black) neighbors. For what we discussed earlier, any particular update order is good as long as, when one performs a measure, each spin has been visited an equal number of times on average.

Both approaches have advantages and weaknesses. In the AMSC scheme, being all spins in a multi-spin-coded word pertaining to completely independent samples, we might tolerate some amount of correlation between samples (sample-to-sample fluctuations will still be the main source of statistical uncertainty) and use a single random number for all the probability comparisons (step 6) in the spin-flip trials of each of the 64 spins of a multi-spin-coded word. This amounts to an important saving in terms of random number generation and increase in the overall throughput. This is not possible at all in the SMSC scheme, in which any small spurious correlation between spins in the same sample would produce unphysical artifacts. On the other side, the AMSC processes just one spin at a time for each single sample, so the wall-clock-time needed to complete a given simulation is not reduced by this technique. The SMSC instead accelerates the simulation time for single sample, shortening considerably the wall-clock time for completing the simulation of a single realization; however, if many samples are needed, SMSC is of no help. Sometimes a trade-off between the two schemes is needed to meet the features of the specific architecture on which the application is implemented. But when one has to deal with very large lattices, as in off-equilibrium dynamics simulations, having a large number of samples is less critical than in equilibrium simulations, and the SMSC is the approach of choice.

At the time the Janus project started, it was clear, analyzing the structure of the data sets and operation sequences involved in the algorithm presented here, that a reconfigurable device would have been an outstanding choice for an efficient implementation.

- First of all, most computations involve a small number of logical operations on a small set of discrete variables (just two values in the "simplest" case discussed above).
- There is a lot of internal parallelism, unveiled, for instance, in the checkerboard decomposition.
- The main computational kernel (steps 4, 5, and 6) when repeated over the whole lattice, gives rise to regular loops performing the same set of operation on

[1] Both random and deterministic sequences guarantee that the asymptotic distribution for the spin configurations on the whole lattice obey the Boltzmann-Gibbs distribution at the given temperature $1/\beta$ with energy given by Eq. 1, see ref. [5] for details.

a data base which has a regular structure, with data values stored in memory in regular patterns that do not depend on the computation itself; Simple state-machines might control the flow of data from and to a set of identical computational blocks.

- The nature of the main basic computation block may be cast to exclusively combinatorial logic.

- Pseudo-random number generation, which is an important time-consuming task in the algorithm, is completely independent on the rest of details of the algorithm; a set of specialized cores could compute pseudo-random numbers with the necessary sustained rate.

The ideal machine for spin glass application would then be a large set of identical cores, able to perform efficiently only the small set of logical manipulation and a little arithmetics; a single control structure can drive all the cores working concurrently and performing all the same thread at the same time. The cost in term of each core would be order 1000 gates, and thousands of them could be arranged in reconfigurable devices. Modern FPGA also come with sufficient internal RAM blocks to store dynamical variables and sustain the required bandwidth.

Traditional architectures, at the time the project started, were poor in extracting the internal parallelism. Even with the SMSC scheme, the generation of several high-quality random numbers (one per spin) was the principal bottle-neck in the computation. Traditional CPU are in fact tailored to manage complex basic computations (integer and floating-point arithmetics) on (relatively) large data sets (32- and 64-bit words).

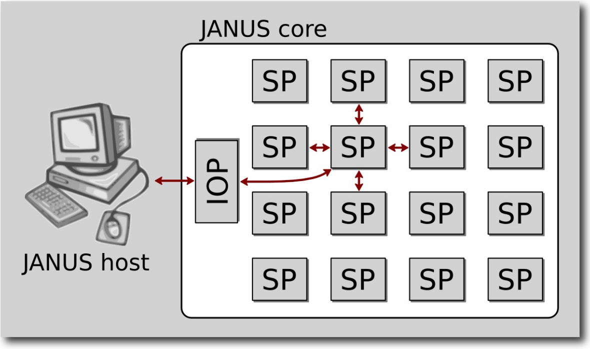
The ideal spin glass machine would be instead more similar to an *application specific* GPU, with data paths tailored to perform the specific sequence of logical operations, a control structure shared by a number of cores larger than in state-of-the-art GPUs, data storage on on-chip memory only and a memory controller optimized for typical access patterns as required by the chosen algorithm.

We resorted then to assemble a mesh of FPGA processors, that gave us freedom to meet all requirements of the prototypical application. Janus is essentially an array of hundreds of thousand of properly tailored small computational cores for the SMSC simulations of systems of interacting discrete variables on regular structures.

Resorting to reconfigurable devices as our enabling technology allows for a large degree of customization and reconfiguration of the applications. In this sense, Janus is not an application specific facility, performing well on a wide range of applications, and with outstanding performances on spin glasses.

3. THE JANUS ARCHITECTURE

The smallest subsystem of Janus is the Janus board (see fig. 1), that contains $16 + 1$ FPGA-based components: 16 SPs (*Simulating Processors*) and an IOP (the *Input-Output Processor*). A board needs an host PC to be operated (but a single PC may control several boards). The SP is the unit in which we exploit internal parallelism, efficiently implementing SMSC algorithms for the chosen applications. External parallelism is usually obtained by farming SPs (16 per board) and driving more boards. In its full configuration, Janus is a stack of 16 independent boards, totaling 256 SPs. There are 8 Linux hosts, each connected to two boards.



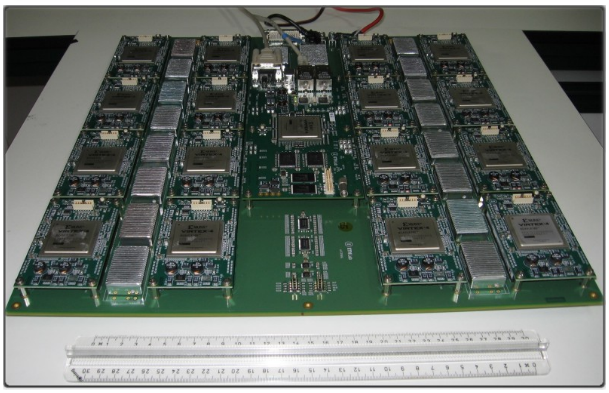


Figure 1: Top: a schematic view of the Janus board internal ed external connections. Bottom: a picture of a Janus board, outside is enclosure box.

On a single board, the 16 SPs are connected in a $2D$ (4×4) toroidal network with nearest neighbors physical links. More complex communications between nodes may be performed by the IOP processor, as more point-to-point connection links the IOP to each SP in the board.

For some applications, the internal parallelism is applied at board level (splitting the simulation of a single sample to 2 or more SPs in the board) and replicating simulations on several boards.

All the devices inside the Janus Board can be managed through the IOP, which is connected to the host PCs via a gigabit Ethernet interface. The host PC runs a standard Linux operating systems, and a set of specific C libraries designed on top of raw socket API allow a user to get through the link to and from the IOP.

We choose the Xilinx Virtex-4 LX200 FPGA as the reconfigurable device for IOPs and SPs. The main clock in the board is 62.5 MHz and it is distributed to all IOP and SP devices. Such a conservative choice has been useful to guarantee mapping of our application to very dense firmware codes (we reached near 95% resource occupation for our heaviest applications).

The 16 SPs and the IOP are housed on daughter cards, plugged onto the Janus motherboard. The choice of daughter cards came from ease of maintenance and possibility of hardware upgrade.

3.1 IOP

The architectural choices for the IOP comes directly from the features of typical spin glass : a Janus board acts as a coprocessor to a standard PC running the simulations, connected on a standard network interface, and with limited interactions between host and board with respect to computational tasks. The FPGA on the IOP is a Xilinx Virtes-4 LX200; the IOP module hosts 8 MB of static memory, a PROM programming device (to load the IOP's FPGA firmware on power-up) and some I/O interfaces: two Gigabit Ethernet and a serial link (useful for debugging purposes).

The IOP's FPGA firmware is not designed to be reconfigured during uptime: it's main task is to manage data streams between the host PC and the SPs. The IOP firmware is logically divided in two main blocks:

- The *IOlink* block is responsible for managing the low level details of communications between IOP and host PC, in particular, manages the Gigabit Ethernet protocol and performs data integrity checks.
- The *MultiDev* block comprises several sub-blocks (devices), each responsible od one specific task; there is a device for each hardware subsystem that might be reached for control or data transfer; among them:
  - A *Program Interface* (ProgInt) connected to the programming interfaces of the FPGAs loads firmware modules onto the SPs
  - Memory Interfaces (MemInt) to read/write internal FPGA memory and the external 8 MB static memory.
  - A *SP Interface* (SPint) is responsible for routing data to SPs and collecting data from them.
- The *Stream Router* routes the data streams from the IOlink to each specific device in the MultiDev, depending on control data present in the data stream itself.

The logic occupancy of the IOP firmware is small compared to typical simulation firmwares in SPs; the IOP occupancy is 4% of logic and 18% of total internal memory, leaving room for implementation of more complex tasks: to perform some global algorithmic operation, for example, or to include an instance of a MicroBlaze$^{\text{TM}}$ microprocessor in order to further loose the coupling to the host PC.

3.2 Running applications on Janus

The SPs are the computational nodes of Janus. Each SP may operate independently or together with other SPs, depending on the firmware loaded by the user program running in the host PCs. The user application consists basically of two layers: a firmware layer, loaded on the SPs, implements the many cores performing the computational tasks as described in section 2. The main flow of the simulation, data initialization, firmware selection and load to SPs, data loading to SPs, SPs operation start, status check and stop, and data retrieve are managed by the user application. From the user point of view, the Janus machine appears as a grid of 32 identical nodes (each host PC manages two boards, but could in principle manage many more). A software layer, the *Janus Operating System* (JOS), comprises a program running as a background process in the host PC (`josd`); it performs the abstraction and acts as a job queue manager, reserving resources to users' jobs depending on their requests on number and topology of SPs (one might require sixteen SPs anywhere in the machine or require them on a single board, actually reserving it as a whole). `josd` sits on the low level communication libraries that write and read data to and from the Gigabit Ethernet devices; `josd` is also aware of internal IOP functions, and is able to translate high-level user requests to data streams that IOP can understand and route to devices in the MultiDev. For instance, if an user wants to send a buffer as a new configuration for spins to a specific SP, `josd` builds the data burst to be sent through Ethernet to the IOlink; the data burst contains the appropriate headers that will be stripped by the Stream Router and the data stream will reach the appropriate device (the SPint); the latter will route the data stream to the chosen SP node. At this point, it is up to the user to ensure that data reaching the SPs be correctly interpreted by the SP. Development of a Janus application consists in facts of two tasks: programming the firmware (in a hardware description language, VHDL is our choice, for instance) for the SPs, complying with the I/O requirements of the SPint in the IOP, and programming the mid-level library functions that, based on the `josd` interface to user application (the `jlib` library), implement the chosen protocol for communication with the SPs. In other words, the user application and the SP firmware can speak their own language, that will be incapsulated in the data flow between `josd` and the SPint in the IOP. Other functionalities common to any possible application (firmware loading onto SPs, data read/write to memory interfaces of the MultiDev, read/write of communication interfaces status registers in the IOlink) can be accessed by other JOS user interfaces (`JOSlib`), or through an interactive shell built on top of them.

3.3 A Sample Spin Glass Application Implementation

We briefly describe our FPGA implementation of the algorithm described in Section 2; for more details we refer the reader to Refs. [3].

We try to exploit all FPGA resources (mainly configurable logic and RAM blocks) to achieve best performance. The Virtex-4 LX200 FPGA by Xilinx comes with many small RAM blocks that we can logically combine and stack to reproduce the $3D$ array of spins on a lattice of size L (L $2D$ memories with width L and depth L). In addition, we *have* to simulate two replicas at least as mentioned in section 2. Then, we consider two replicas of a single disorder sample and divide them in black and white sites in a checkerboard scheme; then, we arrange all black spins of one replica together with all white spins of the other replica in the same $3D$ memory structure. We end up with two mixed $3D$ memory structures (we call them structure P and structure Q); each spin in one structure has neighbors only in the other structure: no spins in the same structure are neighbors of each other in the physical lattices. Now, we can read or write an L-wide word from each memory of the $3D$ structure P per clock cycle; it turns out that having L memories, we can read an entire plane of the $3D$ memory P, update it and write back to memory at the next clock cycle; we only need three planes of *neighbors* from the $3D$ memory structure Q; if we update planes from, say, bottom to top; of the three planes of neighbors in Q needed to update a plane in P, two of them will be neighbors also for the subsequent plane of the structure P. At regime, spanning the whole $3D$ mem-

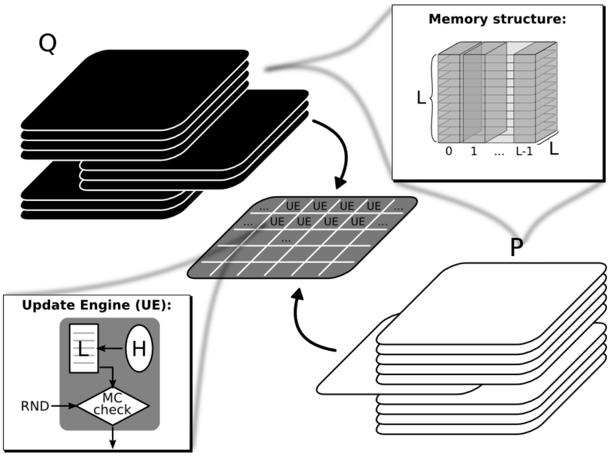


Figure 2: Diagram of the SP firmware operation for the sample application. In the UE, the logic block *H* computes the local field, and use it as an address to the probability Look-Up Table *L*; the *MC check* block compares the probability value with an incoming pseudo-random number *RND* and returns the outcome as the new spin value.

ory structure, only one read and at most one write is needed from each RAM block in order to update an entire plane of the P structure. We can then update an entire plane of L^2 spins per clock cycle. Besides, we instantiate identical $3D$ memory structures to store all the necessary coupling constants, to be fetched with the same rate as the spins in the P structure. We then arrange a set of L^2 identical *update engines* (UE); each UE receives in input the six neighbors of a single spin and combinatorially computes the local field; the pre-computed and constant-in-time spin-up probabilities, normalized to 32-bit unsigned integers, are stored in a Look-Up Table; each UE contains its own table and accesses it independently; the UE also receives a 32-bit word of random bits; a comparison between the addressed 32-bit probability and the 32-bit random number is performed and the new value of the updated spin is returned; Look-Up Tables un UEs are small 32-bit wide memories, for which we exploit *distributed* RAM (instantiated from configurable logic and not consuming RAM blocks).

Once the whole P structure is updated, the controlling state machine interchanges the roles of P and Q, and the update of the Q structure starts with the same procedure described above. When the Q structure is updated, a Monte Carlo step is complete. The whole machinery is depicted in fig. 2

A key advantage of the FPGA implementation is that we can generate pseudo-random numbers concurrently with the rest of the computation and feed them to UEs at a proper rate. We usually resort to the 32-bit Parisi-Rapuano random number generator [6].

A single Parisi-Rapuano 32-bit shift register is a sequence $I(k)$ of 32-bit integers; if we initialize externally the first 62 values, the set of rules

$$\begin{aligned} I(k) &= I(k-24) + I(k-55) \\ R(k) &= I(k) \otimes I(k-61) \ , \end{aligned} \tag{8}$$

generates $I(k)$ as the new element of the sequence and $R(k)$ is the generated pseudo-random value. This simple generation rule is easily arranged on configurable logic only, so that exploiting cascaded structures each wheel is capable of hundreds of iterations per clock cycle. (a close inspection of the rules, for example, reveals that 24 consecutive values can always be generated independently; requiring more than 24 numbers at once, forces keeping intermediate values on registers and cascading the necessary combinatorial logic) This saves us a lot of resources, as a single shift register requires to keep 62 32-bit integers in registers or distributed RAM, and we need hundreds of generated random numbers per clock cycle.

Increasing the number of random generation by a single wheel increase its combinatorial complexity but saves resources; having more generators is resource greedy but allow for easier routing of paths at compilation time.

The trade off between the number of generators and how many numbers any of them could produce per clock cycle impacts on the total number of UEs we may insert in the implementation. Another constraint on the number of UEs is the size of the system: it is advisable, to reduce the complexity of the control logic, that the number of UEs be an integer divider of the number L^2 of sites in a plane;

It is usually possible to place more than 512 UEs and up to 1024 UEs in a single FPGA, thus sustaining an update time of 16 ps per spin at 62.5 MHz clock speed.

3.4 Other applications

There are straightforward generalizations of the model presented in Section 2. If we consider q-valued spin variables, we obtain the *q-states disordered Potts model* (the Edwards-Anderson model is the 2-states Potts glass). We can also easily add dilution variables (a binary variable on each site representing the presence or absence of a spin on that site) and forcing fields (another binary variable representing a preferred orientation for a spin, adding to the local field).

When dealing with *equilibrium* simulations, one must ensure that equilibrium has been reached for the simulated dynamics. Due to the sluggish dynamics that characterize these systems, it is very hard to approach equilibrium by means of a local simulated dynamics like the Heat-Bath algorithm described in Section 2. In these situations, one resorts to the Parallel Tempering scheme [8], in which the local dynamics is interleaved with global moves that try to change the temperature of the systems. Actually, one simulates in parallel a number N of replicas of a samples, each at a different temperature, from very high (the system freely moves away from its initial configuration) to very low (the system configuration is almost frozen, correlations are hard to decay with time). Periodically, the interchange of temperature is proposed between the replicas, so that frozen ones get the chance to evolve at higher temperature and decorrelate faster; even if the computational burden appears to increase by a factor N, the decrease in terms of time needed to thermalize a single sample at constant temperature is dramatic. When the algorithm is at regime, each replica is always in equilibrium at the respective temperature. The (Monte Carlo) time needed to reach the equilibrium condition strongly depend on the disorder of each sample: for such reason, even if we need thousands of samples to extract statistical measures, it is convenient to pursue synchronous parallelism.

The computation for the trial temperature exchange is as follows: one has to compute the difference in total energy ΔE between the two replicas, an multiply it by the difference $\Delta\beta$ in the inverse temperature of the two replicas, and then generate a uniformly distributed random number in $[0, 1)$. Then the following condition

$$\log \rho < \Delta\beta\Delta E \qquad (9)$$

triggers the acceptance of the temperature exchange (if $\Delta\beta\Delta E$ is positive then the test is always successful[2]); the trials are performed in a specific order, usually among replicas with adjacent β values and starting from the hottest replicas down to the coldest.

To implement parallel tempering, we simulate the Heat-Bath dynamics of the N replicas inside one FPGA, arranging them in P, Q pairs as described above (now we must differentiate the Look-Up Tables as the two replicas in the P, Q structures have different temperatures; there will be in total N sets of 7 probability values; we store all of them on RAM blocks, and the sets are indexed by a *beta index*; each replica refers to a single probability set using the beta index; when the parallel simulation of a pair of replicas starts, we load the corresponding sets of of probabilities into the UEs Look-Up Tables consistently with the checkerboard scheme); every now and then, say each 10 complete lattice sweeps of all replicas, we perform the Parallel Tempering update; the ΔE computation is straightforward (it amounts, random number generation apart, as a full lattice sweep, with inhibited P or Q memory writes; the local energies on the sites as computed in the UEs can be summed up in one clock-cycle by a pipelined binary tree adder) but the logarithm computation takes several cycles; being it completely independent on the 10 Monte Carlo steps outcome, it can be put in parallel with the Heat-Bath dynamics to avoid slowing down the simulation.

Due to limited resource in the SPs' FPGAs, our implementation of Parallel Tempering is limited to $N = 128$ replicas of a single samples of linear size up to $L = 32$;

In order to speed up the simulations of particularly hard samples, a variant implementation provides the Parallel Tempering scheme at board level, distributing replicas among the 16 SPs and performing the temperature exchange protocol in the IOP.

We successfully programmed and used the Janus computer to simulate various glassy systems, studying in great detail their properties at equilibrium and out of equilibrium. The interested reader can find in Refs. [7] an exhaustive survey on grounbreaking results.

4. JANUS PERFORMANCES

We refer mainly to the sample algorithm outlined in section 2. We consider an application running at 62.5 MHz clock cycle and performing the conservative number of 800 updates per clock cycle (this is actually the firmware version used for production in the work presented in [2]) in a single SP.

In what follows we consider the single spin-update-time (SUT, the time needed to perform steps 3, 4, 5, and 6 of the procedure described in section 2 on a single spin variable) as the unit of performance. On an SP running 1024 updates per clock cycle, the SUT is 16 ps. In 2007, one year after the project started, our best Ising spin glass application on a Core 2 Duo CPU ran at 400 ps SUT with a fully AMSC implementation on 256 independent samples; our best SMSC implementation ran at 1 ns SUT.

Since then, commodity CPUs have largely improved in terms of explicit parallelism; the performance gap with Janus is closing-up, being the latter still 10 times more efficient than any commercially available solution. We performed several performance measures on a wide range of many-core systems (Cell Broadband Engine, 4-core Nehalem Intel CPU Xeon 5560, Tesla C1060 GP-GPU) [9], and some partial tests on a 8-core Intel Sandy Bridge processor (Xeon E5-2680). Results are summarized in Table 1.

We described the SP of Janus as a module for strict SMSC, and that we trivially obtain repetition by farming over the whole Janus machine. A comparison to commercial processor is still unfair in this sense, since the latter greatly benefit, in terms of performance, by the use of AMSC techniques. Usually, a mixed degree of parallelization has been introduced by either mixing the AMSC data-word filling scheme with internal (synchronous) parallelization or the SMSC scheme with external (asynchronous) parallelization, using all available cores on the CPU or vector registers, in order to exploit peculiar features and to maximize performances. In what follows, we compare Janus performances to mixed AMSC-SMSC implementations, reporting on two kind of SUT measures: the time needed for completing the simulation divided by the total amount of trial spin-flips needed to update a single sample (single-system SUT): this has to be compared with the full SMSC implementation in a single Janus SP; the time needed for completing the simulation divided by the total amount of trial spin-flips needed to update all the simulated sample (global SUT): we compare this to the performance of a Janus board, in which one may consider both AMSC and SMSC levels of parallelism.

Another important point in making comparisons is that many-core CPUs performances become better as the lattice size increases, exploiting more efficiently the possibility to split the system on several cores. In addition, a comparison for very large sizes is not possible, as the Janus SPs is strongly limited by the on-chip RAM, which is about 6 Mbit for the devices we chose (we need roughly 5 bits of memory per lattice size; at largest sizes, routing difficulties between logical devices and RAM blocks arise, as the use of FPGA resources approaches 100%). In this respect, standard CPUs do not suffer for very large lattice sizes as long as there is enough cache memory, and even for the reasonably largest lattice size, multi-spin-coded implementation have very low memory occupancies. On the other side, performance of Janus depends on lattice sizes only by our particular choices in the firmware implementation: having a number of UEs equal to an integer multiple of a power of 2 greatly simplifies our coding; it comes from requiring the number of UEs to be equal to or an integer submultiple of the number of sites in a lattice plane; it happens then that our code for $L = 16$ runs at 256 updates per clock-cycle, the $L = 32$ and $L = 64$ codes run at 1024, the $L = 80$ code runs at 800, the $L = 48$ code runs at 768 and the $L = 96$ code can be com-

[2] We are trying to transition from a state with replica i at temperature β_i and replica j at temperature β_j to a state with β values interchanged. Equation 9 comes from requiring that either the probability of the transition or its inverse be the maximum possible (unity): this is essentially then a Metropolis algorithm [5] for moving around replicas in the space of β values.

single-system SUT (ns/spin)						
L	Janus SP	I-NH (8 Cores)	CBE (8-SPE)	CBE (16-SPE)	Tesla C1060	I-SB (16 cores)
16	0.063	0.98	0.83	1.17	–	–
32	0.016	0.26	0.40	0.26	1.24	0.37
48	0.021	0.34	0.48	0.25	1.10	0.23
64	0.016	0.20	0.29	0.15	0.72	0.12
80	0.020	0.34	0.82	1.03	0.88	0.17
96	0.027	0.20	0.42	0.41	0.86	0.09
128	–	0.20	0.24	0.12	0.64	0.09

global SUT (ns/spin)						
L	Janus	I-NH (8 Cores)	CBE (8-SPE)	CBE (16-SPE)	Tesla C1060	I-SB (16 cores)
16	0.004 (16)	0.031 (32)	0.052 (16)	0.073 (16)	–	–
32	0.001 (16)	0.032 (8)	0.050 (8)	0.032 (8)	0.31 (4)	0.048 (8)
48	0.0013 (16)	0.021 (16)	0.030 (8)	0.016 (16)	0.27 (4)	0.015(16)
64	0.001 (16)	0.025 (8)	0.072 (4)	0.037 (4)	0.18 (4)	0.015 (8)
80	0.0013 (16)	0.021 (16)	0.051 (16)	0.064 (16)	0.22 (4)	0.011 (16)
96	0.0017 (16)	0.025 (8)	0.052 (8)	0.051 (8)	0.21 (4)	0.012 (8)
128	–	0.025 (8)	0.120 (2)	0.060 (2)	0.16 (4)	0.011 (8)

Table 1: Comparison of Janus performance on the sample application to some commercial processor: Intel Nehalem (dual socket board with two 4-core Xeon 5560), Tesla C1060 and Intel Sandy-Bridge (dual socket board with two 8-core Xeon E5-2680). In the upper part, the single-system spin update time, to be compared with the SMSC implementation of a single SP of Janus. In the lower part, the global spin-update time, to be compared to the global spin-update time of a whole Janus board. The number of independent systems simulated in parallel in the AMSC scheme is shown in parentheses

piled meeting timing requirement at 576 updates per cycle "only". There is an intrinsic limit in the number of update cells which is around 10^3 with our implementation, due to resource limits in the FPGA; our code for the $L = 64$ lattice running at 1024 updates per clock cycle occupies 93% of the available logic cells and 57% of memory (RAM blocks) resources.

The upper part of Table 1 show single-system SUTs for Janus compared to that of some commercial architectures, for several lattice sizes. In the lower part, the comparison between global SUTs is made with a Janus board (16 SPs); number in parentheses represent the degree of external parallelism adopted by each solution (i.e. the number of independent systems updated in parallel).

The figures we show for the GPU processor deserve a further comment. Floating-point application usually get large advantages by the use of GPUs, but this is not the case for our sample application; in facts, the Tesla C1060 does not really outperform other standard architectures, and keeps one order of magnitude less performing than Janus hardware. GP-GPU are more effective on applications with reduced memory accesses; the computational kernel of our application requires few operations: the local field of a single spin, for instance, can be computed with 6 XORs and 5 sums on a set of 12 variables (6 neighbor spins and 6 couplings). The GP-GPU performances are then limited by memory-bandwidth (and thread synchronization overheads) despite peak performances are one order of magnitude larger than competing multi-core architectures.

Let's now try an estimate and comparison of power consumption, computing approximatively the energy needed for a typical simulation campaign, as, for instance the one from which we obtained the results presented in [2]. We performed 10^{11} Monte Carlo steps in the Heat-Bath dynamics for a three dimensional Edwards-Anderson model, of linear size $L = 80$, with 256 samples, each replicated twice, and farming out the samples to the 256 SPs of the whole machine. Each SP simulated a single couple of replicas, so it performed $80^3 \times 10^{11} \times 2$ spin flips, at a net single-system SUT of 20 ps, which amount to approximately 2000000 seconds ($\simeq 24$ days). For this application, 16 boards run uninterruptedly at a constant power consumption of around 560 W each (35 W per SP, neglecting IOP's contribution, and considering all power conversion between the power supply and the SPs), for a total energy cost of $16 \times 600 \times 2000000 \simeq$ 18 GJ. Let's take the best figure for the global SUT (allowing the best total energy performance) for the $L = 80$ case from table 1, the 8-cores Intel Sandy-Bridge: 0.011 ns global SUT corresponding to 0.17 ns single-system SUT running 16 samples in parallel. This count for completing the above program with a 16 processors farm (256 samples) running uninterruptedly for 17000000 seconds (201 days; note that five years ago on dual-core CPUs, the wall-clock time would have been 10 times longer), and consuming in total 26 GJ, counting a contribution of 95 W per CPU.

Six years ago, our best code on a single dual-core CPU performed at 0.385 ns global SUT with no external parallelism implemented (full AMSC); the two cores together would have simulated all the 256 samples in parallel, with a single-system SUT of ~ 100 ns, (taking 5000 times longer than Janus), and with a power consumption of about 65 W would have needed 660 GJ. A SMSC implementation running at 3 ns single-system SUT would have shortened considerably the wall-clock time (roughly a factor 30), at the cost of a worse global SUT (256 CPUs involved), and then approximately 8 time more power consumption (about 5 TJ).

We see then that recent standard architecture did indeed fill the gap with Janus in terms of energy consumption. Still, the Janus computer, based on six-year-old technology, remains architecture of choice for spin glass simulations. The figures we presented show that Janus is capable of performing a given simulation in 10 times shorter wall-clock-time. Still, FPGA technology has followed its course and we expect that today state-of-the-art FPGA based processors would outperform the Janus SPs.

Finally, we remark that straightforward generalizations of the Edwards-Anderson model, e.g. the q-states disordered Potts model, come with small degradation of performances in terms of spin update time (and almost no degradation

device	logic cells	distributed RAM	block RAM
XC4VLX200	200448	1392 Kb	6048 Kb
XC7VX485T	485760	8175 Kb	37080 Kb

Table 2: Resource comparison between FPGAs in Janus (Xilinx XC4VLX200) and Janus2 (Xilink XC7VX485T) (figures reported from Xilinx datasheets).

at all in terms of SUT/$\log_2 q$). In facts, we only need to make room for $log_2 q$ bits to store for a single Potts variable, paying a cost in simulable system sizes and number of UEs (that will be slightly more complex than described in the 2-states case) but we can always reconfigure the FPGA to perfectly balance the data bandwidth needed to feed the grid of local updating engines. On standard CPUs, this small increase in complexity reflects invariably in heavier computational kernels and increased bandwidth demand with the same data-paths.

In addition, we also expect, with the new supercomputer, to expand both time and length scales of our simulations of an order of magnitude at least, with a limited increase in power requirement. A similar improvement in simulations with standard processors would imply roughly 100 times more computation, and then 100 more power consumption (supposing an optimistic linear increase in wall-clock time).

5. JANUS2

The next generation of the Janus supercomputer will respect the general architecture of its predecessor, with many improvements made possible by technology advances in the last years.

We plan the following main improvements:

- Latest-generation FPGA devices.
- A tighter coupling between the IOP and the host PC.
- A faster and more flexible communication between IOP and SPs inside one board and across boards.

Janus2 will be a again cluster of processing boards. Each board is a set of 16 SPs and an IOP as in Janus, built as piggy-back modules on a processing board (PB). The PB provides all electrical links and connectors to arrange the SP grid into a $4 \times 4 \times 1$ $3D$ toroidal network: the added third dimension with respect to the Janus board allow for communication between SPs on adjacent boards; a Janus2 machine with a number B of boards will be a $4 \times 4 \times B$ $3D$ toroidal network. The PB also provides the 125 MHz master clock to all devices.

The IOP features the largest increase in complexity with respect to other Janus components. It will integrate all of the previous IOP functionality and will integrate the host PC. A Computer-On-Module (COM) express board plugs onto the IOP piggy-back module. The COM express system will have an Intel Nehalem or Sandy-Bridge class CPU and at least 4 GB DDR3 memory. A Solid State Disk will be connected to the COM express via SATA links. The core of the IOP will of course be an FPGA performing all data-routing and featuring all control features described for the Janus IOP (see Sec 3.1). We select the Xilinx Virtex-7 XC7VX485T. The FPGA processor will export a 8x PCI Express Gen 2 link to the COM module and provide enough high-speed serial links (GTX) to all the SPs on the board and to IOPs in other boards. The IOP interfaces to the world via an Infiniband adapter; there will be service connections also: two Gigabit Ethernet channel and two serial interfaces (one of each type for both the COM and the FPGA). The IOP of course provides also the programming interface for on-the-fly configuration of the SPs' FPGAs; each SP will be configurable independently one on each other.

The SP also features one Xilinx Virtex-7 XC7VX485T. At variance with the previous Janus SP, it will also host some DDR3 DRAM. It will have direct connections to neighboring SPs in the $3D$ logical toroidal network (1 per space directions, 6 total), physically implemented with high-speed serial transceivers in the FPGA (5 GB/s transfer rate minimum in each direction).

The fast interconnection network and the new FPGA features provide large margins for performance improvements with respect to the previous Janus computer. Janus 2 features FPGAs with more than twice the logic, 6 times more distributed memory and 6 times more block RAM (see table 2) than the largest Virtex-4 devices.

The available logic permits a factor 2 increase in terms of number of updates per clock-cycle. Another factor 2 comes from the faster clock of the system (in Janus it was 62.5 MHz). This amounts for a global SUT increase of a factor 4. Please note that we consider running even at faster speed inside the SPs, and that we are conservatively supposing that our application will run at the rate given by the master clock. Another factor 2 improvement could come from extreme tailoring our application to the new available fast FPGA) The main difference with Janus will be the high-speed direct connections between the SPs in the $3D$ mesh: each link is about five times faster than each link in the $2D$ SP mesh of a Janus board.

Taking in consideration the sample application described in section 2, available memory and connection speed allow for simulating two replicas of a single sample of a $3D$ spin glass on a lattice of linear size $L = 1200$ dividing it into sublattices of $300 \times 300 \times 150$ among the 128 SPs of 8 adjacent boards. With a sustained update rate of 2000 spins per clock on 300×300 planes cycle inside each SPs, the most demanding bandwidth would be the border spins transfer through the links connecting SPs in different boards, requiring 2000 bits transfered in 150 cycles (before they become necessary to start the next update sequence in the neighbor SP) corresponding to 3.3 Gb/s.

Such implementation would then be capable of an update rate of 256000 spins per clock-cycle, corresponding to a single-system SUT of 32×10^{-15} (and the global SUT is just half that amount in a compete configuration with 16 Janus2 boards): in Janus2 many SPs in many boards will be able to concur in exploiting all the internal parallelism. With such very large simulated lattice sizes, the fluctuations of measures between different disorder samples become less important, and we could be contented with tens of samples instead of hundreds. A simulation program as the one described at the end of section 4 (two replicas, 10^{11} Monte Carlo steps) with only 16 samples, would obtain relevant results in about one year (if one decides to sacrify a large fraction of the machine durability to a single ambitious task).

In a single SP the single-system SUT would be around 4 times better than in Janus. With only two SPs, we could allocate two replicas of a $L = 150$ system and simulate them

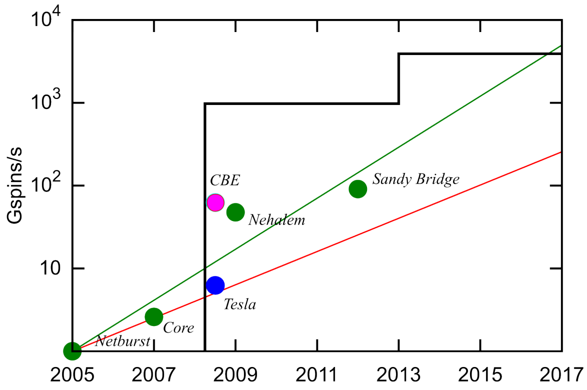


Figure 3: Sketch of performance growth in time of our simulation codes following the technology improvement; each point correspond to the time (abscissae) we actually get a code running on a specific architecture; the obtained performances, as reported in table 1 are in ordinates, in *billions of updated spins per second* units. Green points are the Intel series (one CPU); the red point corresponds to the CBE and the blue point is the Tesla GPU. The black line are performances of one board of the Janus machines (Janus2 is expected starting scientific production in mid 2013). The red line represents the performance growth predicted by a Moore's Law with doubling performance each 18 **months (taking our first implementations in 2005). The green line is the performance growth trend extracted by considering only our data points for the Intel series, giving a doubling in performance each year, approximately.**

with a 4500 updates per spin rate and complete the task in a month. On a farm of 16 8-cores Sandy-Bridge processors it would take almost two years (assuming the measured performance for a $L = 128$ system from table 1 applies). It is very difficult to foresee the equivalent energy consumption, as we can only forecast the SPs power consumption by the software development tools, without direct measures on the running hardware. We foresee an optimistic figure of 50 W per SP, obtaining a 30 GJ total needed to complete the simulation program for a $L = 150$ system (it would be 5 GJ for the $L = 80$ lattice). The farm of 16 commercial CPUs (Sandy-Bridge) would consume 92 GJ.

The Janus2 computer then establish again near two orders of magnitude in the gap with commercial processors. Power consumption improvements are sizeable but no so impressive as in that case of Janus with respect to its competitors six years ago; in this respect, standard CPUs' have largely improved in terms of energy-per-performance in the last years.

6. CONCLUSIONS

We have described two Janus generations of supercomputer for Spin Glass application. In the second generation we expect a significant improvement with respect to the previous one, due to progresses in FPGA technology and some major architectural design modifications (mainly the interconnection network topology) that will significantly decrease wall-clock time of simulations of Spin Glasses.

We expect Janus2 to be a durable facility; we can compare the performances of one board of the Janus machine and the expected performances of a Janus2 board to the performance growth trend of commercial CPUs (see fig. 3). In both cases, we consider a technology available when we have optimized code ready for testing; of course, in the case of janus machines, this includes the necessary long development time; on the other side, carefully optimize the simulation code on commercial CPUs can be a hard and lengthy task.

In figure 3 we compare best performance values taken from table 1. The black line represents performances in terms of billions of spins updated per second for a single board of a Janus-type computer (we expect Janus2 to start operation in summer 2013).

We also report on the figure (the green line) the performance growth trend we experienced with the Intel series for our applications (a doubling in performance every year approximately) and the red line is the expected growth since 2005 by a Moore's law (doubling each 18 months).

We have only two "measurements" of the performances of FPGAs in Spin Glass applications, and one of them is the conservative estimate we provide for Janus2. Performance increase with reconfigurable devices appear slower than the average performance growth rate of other technologies. It should be noted that, in the case of Intel processors, scaling of performances in our application either fell behind Moore's law or surpassed it between two consecutive microarchitecture releases (as in the case of Nehalem-Sandy Bridge transition). Newer FPGAs get larger and faster, but the technology used to improve them could not be ideal to our applications (this is the case, for example, of the largest devices in the Virtex-7 family, whose structure made of stacked smaller devices may introduce a bottleneck in the data-paths due to the physical links between layers, that are slower than the typical signal path in any smaller device). On the other side, any increase of a factor X in the target clock frequency is transferred directly as a factor X in performances, and is usually easier to meet time requirements when compiling a firmware for a larger devices.

If the trend in Intel performance growth should continue, and if the estimate we provide for the Janus2 computer holds, the latter has an advantage of almost four years, which is enough to ensure a significant output of scientific production. If the trend continues by the usual Moore's law starting form the Sandy-Bridge performances, Janus2 will remain unrivaled for two years more than in the previous case.

7. ADDITIONAL AUTHORS

M. Baity-Jesi, L.A. Fernandez, V. Martin-Mayor and B. Seoane (Departamento de Física Teórica I, Universidad Complutense, 28040 Madrid, and Instituto de Biocomputación y Física de Sistemas Complejos (BIFI), 50018 Zaragoza, Spain);
R.A. Baños, A. Cruz, J. Monforte-Garcia and A. Tarancon (Departamento de Física Teórica, Universidad de Zaragoza, 50009 Zaragoza, and BIFI);
J.M. Gil-Narvion, M. Guidetti and S. Perez-Gaviro (BIFI);
A. Gordillo-Guerrero (Departamento de Ingeniería Eléctrica, Electrónica y Automática, Universidad de Extremadura, 10071 Cáceres, and BIFI);
D. Iñiguez (Fundacion ARAID, Diputación General de

Aragón, Zaragoza, and BIFI);
A. Maiorano and D. Yllanes (Dipartimento di Fisica, Sapienza Università di Roma, 00185 Rome, Italy and BIFI);
F. Mantovani (Dipartimento di Fisica, Università di Ferrara, 44100 Ferrara, Italy);
E. Marinari, G. Parisi and F. Ricci-Tersenghi (Dipartimento di Fisica, IPCF-CNR, UOS Roma Kerberos and INFN, Sapienza Università di Roma, 00185 Rome, Italy);
A. Muñoz-Sudupe (Departamento de Física Teórica I, Universidad Complutense, 28040 Madrid, Spain);
D. Navarro (Departamento de Ingeniería, Electrónica y Comunicaciones and Instituto deInvestigación en Ingeniería) de Aragón (I3A), Universidad de Zaragoza, 50018 Zaragoza, Spain);
M. Pivanti (Dipartimento di Fisica, Sapienza Università di Roma, 00185 Rome, Italy);
J.J. Ruiz-Lorenzo (Departamento de Física, Universidad de Extremadura, 06071 Badajoz, Spain and BIFI);
S.F. Schifano (Dipartimento di Matematica e Informatica, Università di Ferrara and INFN - Sezione di Ferrara, 44100 Ferrara, Italy);
P. Tellez (Departamento de Física Teórica, Universidad de Zaragoza, 50009 Zaragoza, Spain);
R. Tripiccione (Dipartimento di Fisica, Università di Ferrara and INFN -Sezione di Ferrara, 44100 Ferrara, Italy)

8. REFERENCES

[1] M. Mézard, G. Parisi and M.A. Virasoro, *Spin Glass Theory and Beyond* (World Scientific, Singapore, 1987); A. P. Young (editor), *Spin Glasses and Random Fields*, (World Scientific, Singapore, 1998).

[2] The Janus Collaboration, Phys. Rev. Lett. **101**, (2008) 157201;

[3] The Janus Collaboration, Comp. Phys. Comm. **178**, (2008) 208-216; *IANUS: Scientific Computing on an FPGA-based Architecture*, in *Proceedings of ParCo2007, Parallel Computing: Architectures, Algorithms and Applications* (NIC Series Vol. 38, 2007) 553-560; Computing in Science & Engineering **8**, (2006) 41-49; Computing in Science & Engineering **11**, (2009) 48-58.

[4] S. F. Edwards and P. W. Anderson, J. Phys. F: Metal Phys. **5**, (1975) 965-974; *ibid.* **6**, (1976) 1927-1937.

[5] A.D. Sokal, *Functional Integration: Basics and Applications (1996 Cargèse School)* ed. C. DeWitt-Morette, P. Cartier and A. Folacci (1997 New York: Plenum)

[6] G. Parisi and F. Rapuano, Phys. Lett. B **157**, (1985) 301-302.

[7] The Janus Collaboration: J. Stat. Phys. **135**, 1121-1158 (2009); Phys. Rev. B **79**, 184408 (2009); J. Stat. Mech. (2010) P05002; J. Stat. Mech. (2010) P06026; Phys. Rev. Lett. **105**, 177202 (2010); Phys. Rev. B **84**, 174209 (2011); Proc. Natl. Acad. Sci. USA (2012) **109**, 6452-6456

[8] Hukushima, K., Nemoto, K. *Exchange Monte Carlo Method and Application to Spin Glass Simulations* J Physical Soc Japan 65,1604âĂŞ1608 (1995), `arXiv:cond-mat/9512035`, E. Marinari, *Optimized Monte Carlo Methods* in Advances in Computer Simulation. Lecture Notes in Physics, 1998, Volume 501/1998, 50-81 (1996), `arXiv:cond-mat/9612010`.

[9] M. Guidetti et al., *Spin Glass Monte Carlo Simulations on the Cell Broadband Engine* in *Proc. of PPAM09*, (Lecture Notes on Computer Science (LNCS) 6067, Springer 2010) 467-476. M. Guidetti et al., *Monte Carlo Simulations of Spin Systems on Multi-core Processors* (Lecture Notes on Computer Science (LNCS) 7133 K. Jonasson (ed.), Springer, Heidelberg 2010) 220-230.