The Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Time-energy Analysis of Multi-level Parallelism in Heterogeneous Clusters: The case of EEG Classification in BCI Tasks

Juan José Escobar · Julio Ortega · Antonio F. Díaz · Jesús González · Miguel Damas

Received: date / Accepted: date

Abstract Present heterogeneous architectures interconnect nodes including multiple multi-core microprocessors and accelerators that allow different strategies to accelerate the applications and optimize their energy consumption according to the specific power-performance trade-offs. In this paper, a multi-level parallel procedure is proposed to take advantage of all nodes of a heterogeneous CPU-GPU cluster. Two more alternatives have been implemented, and experimentally compared and analyzed from both running time and energy consumption. Although the paper considers an evolutionary master-worker algorithm for feature selection in EEG classification, the conclusions from the experimental analysis here provided can be frequently applied, as many other useful bioinformatics and data mining applications show the same master-worker profile than the classification problem here considered. Our parallel approach allows to reduce the time by a factor of up to 83, with only about a 4.9% of energy consumed by the sequential procedure, in a cluster with 36 CPU cores and 43 GPU compute units.

Keywords EEG classification · Hybrid programming · Time-energy analysis · Heterogeneous CPU-GPU parallel architectures · Master-worker algorithms · Multi-level parallelism

This research has been funded by the Spanish "Ministerio de Ciencia, Innovación y Universidades" through the grant PGC2018-098813-B-C31 and the ERDF funds

This is a post-peer-review, pre-copyedit version of an article published in The Journal of Supercomputing. The final authenticated version is available online at: http://dx.doi.org/10.1007/s11227-019-02908-4

J.J. Escobar · J. Ortega · A.F. Díaz · J. González · M. Damas
Dept. of Computer Architecture and Technology, CITIC, University of Granada (Spain)
Tel.: +34-958248994
Fax: +34-958248993
E-mail: jjescobar, jortega, afdiaz, jesusgonzalez, mdamas{@ugr.es}

1 Introduction

Machine learning tasks for classification, clustering, feature selection, and optimization problems are present in many useful applications such as bioinformatics and bioengineering, which usually require high-performance platforms whose cost in both economic and environmental terms should be carefully taken into account. Indeed, the minimization of energy consumption has emerged as one of the main issues of research in computer systems, and efficiency today not only means good speedups but also optimal energy consumption. Moreover, the tasks present in machine learning also show different processing characteristics and thus diverse profiles in energy consumption with respect to their parallel behavior.

The use of heterogeneous computing platforms, including multi-core CPUs (Central Processing Units) and other accelerators with different architectures, such as GPUs (Graphic Processing Units) and FPGAs (Field Programmable Gate Arrays), would make possible faster parallel codes that take advantage from different kinds of parallelism in the platform. Nevertheless, although the different speed-to-power ratios of present processors give more possibilities to distribute the work in order to optimize energy consumption, workload balancing is much more complex to solve [1], and approaches that take into account speedup and energy consumption criteria constitute an important research issue [2–5]. An alternative to take into account energy consumption issues in the code execution is to use the power management policies implemented by present microprocessors. However, as these policies usually depend on the microprocessor and are not visible to the user, black-box approaches [6] to determine the main characteristics of the applied power management policy could be devised. Also, techniques such as Dynamic Voltage and Frequency Scaling (DVFS) could make possible for the user to control the frequency and voltage of the processors in the platform [3, 4].

In this paper, we illustrate this situation through a multi-objective approach for Electroencephalogram (EEG) classification in BCI tasks. This application deals with feature selection in classification problems involving patterns with a high number of features and the curse of dimensionality problem. The multi-objective evolutionary algorithm implements a *wrapper* procedure for feature selection together with a unsupervised procedure whose performance is used to evaluate the fitness of each individual in the subpopulation [7]. The paper analyzes the energy-time behavior on a heterogeneous CPU-GPU cluster of different multi-level parallel alternatives. For a number of years, the rate of increase in computing power has started to fall due to the Moore's law, with R&D more focused on energy efficiency due to environmental issues, and less on increasing CPU and GPU brute power. Even the introduction of multi-core processors is not enough to reverse this trend since most applications are not optimized to use more than one core, and the efficient use of multiple cores is still a topic of active research in computer science. In addition, there is a problem with the exponential growth of data generated by applications, which is impossible to cope without the development of new processing techniques.

Thus, currently, the use of computing clusters to solve some scientific problems considering a close CPU-GPU cooperation [8], like the workload distribution here considered, constitutes the mainstream approach to take advantage of technology improvements and overcome the barrier of hardware limitation and energy efficiency [1,9]. Indeed, the development of efficient codes for heterogeneous CPU-GPU systems needs to address hardware and software issues related with the cooperation among computing nodes. Among those, the size of CPU and GPU memories and their bandwidth limitations, the workload balancing among the CPU and GPU devices, the overlapping of data during the computation, and the parallel profile of the application considered.

The main contribution of this paper is to provide two new algorithms derived from the one presented in [10], and a complementary energy-time model that not only contributes to understand the experimental power-performance behavior of the new codes but also to get an insight in the identification of the workload distribution issues to be considered when coding the algorithm. One of the proposed algorithms is postulated as an improved, optimized, and faster version than the algorithm proposed in [10], while in the other the changes are related to the library used for the parallelization of the code and it is presented as an alternative that it could be better or worse than that of [10] under different conditions. The three algorithms are compared from the point of view of running time and energy consumption to compare their performance.

After this introduction, Section 2 summarizes the work in the literature related with parallel implementations of evolutionary algorithms in CPU-GPU platforms, and their corresponding energy-aware performance evaluation and scheduling. Section 3 introduces the evolutionary multi-objective approach here considered for feature selection in EEG classification. In addition, this section illustrates the characteristics of the parallel applications that can benefit from the parallel procedures similar to those considered in the paper. Section 4 describes our multi-level parallel approach along with its proposed versions, while Section 5 provides the experimental work and the energy-time approximation models to get an insight into the observed behavior. Finally, Section 6 gives the conclusions of the paper.

2 Related work

Many contributions on parallel implementations of evolutionary algorithms in CPU-GPU platforms have been proposed in the literature, although most of them do not completely exploit the CPU-GPU computing capacity as only use one CPU thread to control the GPU activity [11]. Among the approaches for GPU-based implementations of evolutionary algorithms analyzed in [12], some of them propose to implement all or the most of the steps of the evolutionary algorithm in GPU to decrease the cost of transferring information between CPU and GPU. In general, for an evolutionary algorithm, as the fitness evaluation can be independently done for each individual in the population, this step is usually implemented in parallel. Nevertheless, other steps, such as the

application of evolutionary operators, require interaction among individuals, thus involving some kind of synchronization among the computing elements. This way, two main researching lines can be distinguished among the proposals for GPU implementations: a parallel implementation that shows the same behavior than the sequential one, and the implementation of an evolutionary parallel algorithm tuned to the features of the GPU architecture. However, its characteristics could be different from those of the corresponding sequential algorithm. In this last alternative, an analysis of the suitability of the attained solutions should be done.

Paper [13] describes a CUDA implementation of a parallel genetic algorithm based on an island model, and it is an example of the approaches that, as the one presented in this paper, modify the evolutionary algorithm to reach a more suitable version for the available architecture. An alternative GPU implementation of the non-dominated rank used in NSGA-II, the Archived-based Stochastic Ranking Evolutionary Algorithm (ASREA), is provided in [14]. Paper [15] provides a parallel GPU implementation of a multi-objective evolutionary algorithm for a data mining application on marketing. This approach executes all steps of an NSGA-II algorithm in GPU except the non-dominated sorting and selection, for which a fast procedure is proposed.

There are not many approaches using the CPU and GPU as resources that can be equally considered to distribute the workload of the optimization procedure like in this paper. Paper [11] proposes a methodology to solve optimization problems in heterogeneous CPU-GPU architectures that benefits from both CPU and GPU devices, and points out the usefulness of further researching on this approach. Our approach includes an evolutionary multi-objective optimization and a clustering algorithm applied to a set of high-dimensional patterns. Although the use of heterogeneous architectures has been proposed in previous papers, the parallelization on a heterogeneous platform of a whole data mining application with the characteristics of our target application and the analysis of its energy consumption besides the time performance is less frequent. Paper [16] analyzes the effect of factors such as the communication patterns and the data partition on the performance of data mining applications, and in [17] a parallel multi-objective evolutionary procedure using MPI on only one platform is described. Our approach takes advantage of heterogeneous clusters including multiple CPU-GPU nodes and distributes the workload among both CPU and GPU devices of the nodes to accelerate the application, and thus also allowing energy-saving.

Several approaches have been reported on scheduling procedures that take into account not only running time but also the energy consumption of the program [18–20]. Most of them are based on Dynamic Voltage Scaling (DVS), a technique similar to DVFS but only allows dynamic scaling of the CPU voltage to reduce the energy consumption. Although by reducing the processor supply voltage the computing times of the tasks allocated in the CPU would rise, it could be even possible not to increase the runtime of the parallel program if these increments in the computing times fill the slack times, and thus reducing the time lighter tasks have to spent waiting for heavy tasks. Paper [21]

provides a survey that distinguishes between the approaches based on the use of low-power components, and the techniques that use software approaches and power-scalable components. These last techniques, in turn, can be also classified into two alternatives: (i) those based on the dynamical adjustment of power consumption by taking advantage of power-scalable components and (ii) the energy-aware load balancing techniques.

Nevertheless, it is not always possible to handle the runtime power management, and some rules or principles should be taken into account for developing efficient procedures for a given platform. To do that, models of time and energy consumption are required to estimate the time and energy consumed by a program, according to its profile (inherent parallelism and required synchronization, memory and I/O requirements, etc.). Paper [1] provides a complete survey of power and energy models with the corresponding set of references. These models are classified according to their parameters, level of abstraction, whether they model the instantaneous or average power and the energy, and whether they provide a decomposition of power and energy among elements in the platform. The paper also gives information about the power-energy prediction accuracy, portability to different architectures, and complexity of the model. In this line, [6] proposed a black-box scheduling approach based on an offline power model and an online workload modeling, and papers [22, 23] deal with the determination of power and energy consumption models, either by running micro-benchmarks [22] or through the evaluation of energy consumption of the platform components [23]. Energy consumption characterization by applying multiple linear regression models is proposed in [24].

With respect to the energy consumption efficiency of hybrid CPU-GPU platforms, some results on this topic have been provided by papers [25–28]. For example, [25] provides analytical models to get an insight into performance gains and energy consumption in different CPU-GPU platforms and concludes that a greater parallelism allows opportunities for energy-saving and encourages the development of energy-saving parallel applications. In this paper, we use approximate models based on those given in [28] and in the use of averages of the instantaneous power experimentally sampled in the nodes, to explain the energy-time behavior in a CPU-GPU cluster of the parallel procedures described in Section 4.

3 Master-worker evolutionary algorithms and EEG classification

EEG classification, like other high-dimensional applications in bioinformatics, requires the processing of a huge amount of data, especially whenever the patterns that make up the databases are defined by a large number of features, and therefore the execution times are very high or prohibitive. This way, parallel processing and feature selection are commonly applied to decrease the execution time and/or to improve the quality of the solutions they provide. An approach to the aforementioned applications based on evolutionary algorithms usually implies the evaluation of the utility (or fitness) of a population

Table 1 Runtime distribution among the most relevant steps of an evolutionary multi-objective feature selection procedure for different population sizes, N

N	Evaluation		Non-DS		Crossover		Others		Total
	Time (s)	%	Time (s)	%	Time (s)	%	Time (s)	%	Time (s)
120	119.19	99.93	0.01	0.01	0.03	0.03	0.04	0.03	119.27
240	236.38	99.92	0.07	0.03	0.09	0.04	0.03	0.01	236.57
480	477.00	99.90	0.14	0.03	0.18	0.04	0.14	0.03	477.46
960	954.85	99.87	0.70	0.07	0.31	0.03	0.29	0.03	956.15
2500	2492.26	99.72	5.21	0.21	0.87	0.03	1.00	0.04	2499.34
5000	4973.70	99.48	21.74	0.43	2.03	0.04	2.66	0.05	5000.13
10000	9984.37	99.05	85.22	0.85	3.75	0.04	6.36	0.06	10079.70
15000	14946.12	98.60	196.61	1.30	5.53	0.04	9.87	0.06	15158.13

of solutions on the problem at hand. Moreover, the diverse profiles of parallelism present in the corresponding codes make possible to develop efficient parallel codes for the present heterogeneous CPU-GPU platforms.

For example, in [7], an evolutionary multi-objective optimization method is applied to solve a feature selection problem in a BCI application. The individuals of the population correspond to different sets of features that define the components of the patterns to be classified. These sets of features have to be evaluated by the accuracy of the classifier once it has been adjusted by using the training patterns, which are characterized by the selected features. The iterations required to train the classifier usually require a high amount of computing time as it is shown in Table 1. This table provides the runtime for the main steps of a multi-objective feature selection procedure, analyzed with the tool *gprof* [29]. The fitness evaluation needs between 99.93% of runtime with 120 individuals in the population, and 98.60% with 15,000 individuals. Thus, as the fitness evaluation is completely independent for each individual of the population and is the most time-demanding step, the efforts to parallelize the algorithm should be focused on this task.

This paper analyzes both runtime and energy consumption, in a heterogeneous CPU-GPU cluster, of three multi-level parallel approaches for Multi-Objective Feature Selection (MOFS) applied to EEG classification. They are based on the NSGA-II algorithm, and are responsible for evolving one or multiple subpopulations. In MOFS problems, the patterns usually are characterized by a high number of features, being necessary to select the most relevant features in order to get good classification performance besides decreasing the computational cost of the procedure, among others. Due to the NP-hard complexity of this kind of applications [30,31], it is necessary to use parallel meta-heuristics that are capable of reducing both running time and energy consumed by the algorithms, and heterogeneous parallel architectures to implement them. Our master-worker procedures use the K-means algorithm as unsupervised method to evaluate the solution provided by each individual, whose multi-objective fitness is composed of two cost functions, f_1 and f_2, corresponding to the minimization and maximization of the intra-cluster and inter-cluster distances, respectively, as Equations (1) and (2) show:

$$f_1 = 1 - \left(\sum_{j=1}^{W} \frac{1}{|C^t(j)|} \cdot \left(\sum_{P_i \in C^t(j)} \|P_i - K^t(j)\| \right) \right) \quad (1)$$

$$f_2 = \sum_{j=1}^{W-1} \cdot \left(\sum_{i>j} \|K^t(i) - K^t(j)\| \right) \quad (2)$$

being $|C^t(j)|$ the number of patterns in the cluster $C^t(j)(j = 1, .., W)$ whose centroid is $K^t(j)$, and $\|P_i - K^t(j)\|$ is the Euclidean distance between the pattern P_i and the centroid $K^t(j)$. As each individual performs one or several executions of the K-means algorithm, and this is based on a highly parallelizable operation such as the Euclidean distance, we think that a master-worker procedure that distributes individuals or entire subpopulations on the computing nodes of the cluster could be a good approach to extract the maximum parallelism. In what follows, we describe the different parallel versions to implement the MOFS procedure in CPU-GPU clusters.

4 Multi-level heterogeneous parallel approach to MOFS

The proposed parallel master-worker procedure for multi-objective feature selection takes advantage of up to four parallelism levels depending on the devices used to evaluate the fitness of the individuals. In this paper, we compare three different versions for the procedure that differ on the workload distribution and also in the use of either OpenCL [32] or OpenMP [33] in the corresponding parallel CPU implementations. In what follows, these versions will be called *v1*, *v2*, and *v3*. The pseudocodes and an extensive analysis for version *v1* are provided in [10,34], while present paper describes and provides a comparative analysis of the new version *v3*, whose master and worker pseudocodes are shown in Algorithms 1 and 2, respectively.

Version *v1* uses OpenCL for both CPU and GPU devices to implement the K-means algorithm, which performs the fitness evaluation of the individuals for each subpopulation. In the case of version *v2*, it differs from *v1* in the CPU implementation, since the K-means is coded by using OpenMP pragmas instead of OpenCL. Concerning to the version *v3* here proposed, some improvements in the workload distribution have been added with respect to versions *v1* and *v2* in order to reduce not only the running time but also the energy consumed by the procedure.

The following subsections detail the levels of parallelism implemented for the three versions, *v1*, *v2*, and *v3*, and their differences. Figure 1 summarizes all procedure steps, from the creation and distribution of subpopulations done by the master process, to the fitness evaluation of the individuals in the worker processes. All steps are repeated several times depending on the total number of subpopulations to be evolved and the global migration previously set.

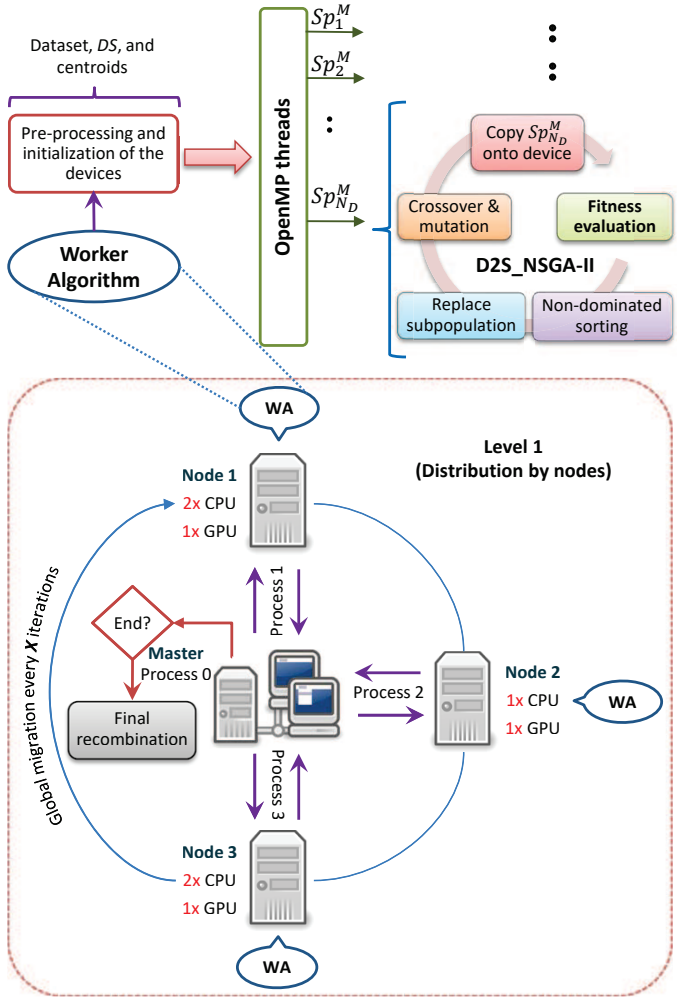


Fig. 1 MPI-OpenMP scheme of the evolutionary procedure to distribute the subpopulations, which constitutes the first and second parallelism levels

4.1 Distribution of subpopulations among nodes

The first level of parallelism corresponds to a dynamic distribution of the subpopulations over the different workers used in the cluster. It has been implemented by using OpenMPI [35] as a Message Passing Interface (MPI) library. We employ this approach instead of a static distribution to avoid imbalanced workload. Firstly, observing Algorithm 1, the master (MPI process with rank number 0) initializes the individuals of the subpopulations (line 2). In line 3, by using an `if-else` statement, the program checks the total number of MPI processes that the application is running. If the condition becomes true, the master will be responsible for doing the entire job. That is, evolve

Algorithm 1: Pseudocode of the master algorithm for *v3*. The master distributes subpopulations among all worker nodes. If only one MPI process is detected (master), it will evaluate all subpopulations

```
1  Function Master(Sp, N_Spop, M, DS, N_D, N_W)
       Input  : Initial subpopulations, Sp_i; ∀i = 1, ..., N_Spop
       Input  : Number of subpopulations, N_Spop, to evolve
       Input  : Number of individuals in the subpopulation, M
       Input  : Dataset with the EEG patterns, DS
       Input  : Number of devices, N_D
       Input  : Number of workers, N_W
       Output: The hypervolume metric
2      Sp ← initSubpopulations(S_p)
3      if Only_master is detected then
4          K ← getRandomCentroids(DS)
5          D ← initDevices(DS, K, N_D)
6          repeat
7              Sp ← evolve(Sp, M, DS, K, D, N_D)
8              Sp ← globalMigration(Sp, N_Spop, M)
9              Sp ← nonDominationSorting(Sp, N_Spop · M)
10         until all N_Gm global migrations are completed;
       // Start the dynamic distribution of subpopulations
11     else
12         repeat
13             RemainingWork ← N_Spop
14             repeat
15                 S_R ← With Recv the request of worker W_j is attended
16                 sent ← min(RemainingWork, S_R)
17                 S_p → Send sent subpopulations to W_j with Send
18                 RemainingWork ← RemainingWork − sent
19             until all N_W workers have their first chunk OR RemainingWork is 0;
20             repeat
21                 S_p ← Receive 1 subpopulation from the worker W_j^t with Recv
22                 sent ← min(RemainingWork, 1)
23                 S_p → Send sent subpopulation to the worker W_j^t with Send
24                 RemainingWork ← RemainingWork − sent
25             until RemainingWork is 0;
26             Sp ← globalMigration(Sp, N_Spop, M)
27             Sp ← nonDominationSorting(Sp, N_Spop · M)
28         until all N_Gm global migrations are completed;
           // Finish the MPI communications
29         END_SIGNAL → broadcastSignal(N_W) with Bcast
30     end
31     S ← subpopulationMerging(Sp, N_Spop · M)
32     return (zitzlerHypervolume(S))
33 End
```

all subpopulations using the available devices of the node to which it belongs, in conjunction with the necessary global migrations between subpopulations and the non-dominated sorting (lines 6-10). Since in this case the master also includes the worker role, it must previously obtain the centroids from dataset DS, which are necessary to perform the K-means algorithm, and initialize the devices in lines 4 and 5, respectively.

In versions *v1* and *v2* the alternative in which the master can have the worker role does not exist (`if-else` in line 3), and then a minimum of two MPI processes are necessary. This would not be relevant if more than one computing node is available, since one of them would allocate the master process and the other one the worker process. However, when only one computing node is available, an MPI process that acts as master and another one that acts as a worker is still mandatory. Logically, this presents a problem with the saturation of CPU and memory resources, since each MPI process is mapped to a logical core of the same node and thus less computing resources are available to evaluate the fitness of the individuals. In addition, other overheads such as the message passing between MPI processes or their synchronization requirements have to be taken into account as it affects to the execution time and the energy consumed. All this issues have been considered and fixed in version *v3* giving the master the ability to do all the job.

If the `if-else` statement becomes false, it means that the subpopulations are not computed by the master. At this point, master and workers are synchronized and ready to start communicating. The distribution of subpopulations to be evolved and the global migration are repeated as many times as the number of global migrations, N_{Gm}, has been set (lines 12-28). Firstly, the master sends a first chunk of subpopulations to all nodes less than or equal to the number of subpopulations requested by the worker (lines 14-19). The idea is to make all nodes busy as soon as possible and thus trying to reduce the imbalanced work. Then, the master asynchronously continues to attend the requests of each worker, and dynamically distributes the rest of subpopulations until there is no more work to do (lines 20-25). Then, the master proceeds to perform a global migration (line 26) between all N_{Spop} subpopulations to improve their diversity, thus trying to avoid the local optima in order to increase the quality of the solutions. A global migration implies to build a new set of subpopulations. To define a new subpopulation, the given set of solutions in the subpopulation receives solutions from the rest of subpopulations. More specifically, each subpopulation contributes with half of its solutions in its present Pareto's front at most.

Once all N_{Spop} subpopulations have been evolved, and all N_{Gm} global migrations have been completed, the master sends the signal (END_SIGNAL) to the workers to notify that there are no more subpopulations to be evolved and the MPI communications have ended (line 29). Moreover, the solutions obtained by the different subpopulations are merged in line 31 to perform the final subpopulation, which includes the best individuals of each subpopulation belonging to the Pareto's front. Finally, in line 32 the solution obtained is returned to the main function.

Algorithm 2: Pseudocode of the worker algorithm for *v3*. The received subpopulations from the master are distributed among the devices. If only one subpopulation is received in the first chunk, its individuals are dynamically assigned to all N_D devices to perform the fitness evaluation

```
1  Function Worker(M, DS, N_D, j)
       Input: Number of individuals in the subpopulation, M
       Input: Dataset with the EEG patterns, DS
       Input: Number of devices, N_D
       Input: Worker ID, j
2      K ← getRandomCentroids(DS)
3      D ← initDevices(DS, K, N_D)
       // Start the dynamic request of subpopulations
4      N_D → Request the master with Isend as many subpopulations as N_D devices
5      Sp_l ← Receive rcv subpopulations from the master with Recv
6      repeat
           /* Start OpenMP section with rcv CPU threads,
              W_j^t; ∀t = 1, ..., rcv ≤ N_D (D2S_NSGA-II algorithm)        */
7          #pragma omp parallel num_threads(rcv)
8          {
9          repeat
10             Sp_{l_i} ← evolve(Sp_{l_i}, M, DS, K, D, N_D)
11             Sp_{l_i} → Return Sp_{l_i} to the master with Isend and ask for another
12             Sp_{l_i} ← W_j^t receives new subpopulation from the master with Recv
13         until no new subpopulation is received in Sp_l;
14         }
           /* End OpenMP section. Waiting for the master to perform the
              global migration or send the END_SIGNAL                     */
15         N_D → Request again the master with Isend N_D subpopulations at most
16         Sp_l ← Receive rcv subpopulations from the master with Recv
17     until the END_SIGNAL is received;
18 End
```

4.2 Distribution of individuals or subpopulations by devices

If more than one MPI process is running, while the master process is scheduling the distribution of subpopulations, the workers (Algorithm 2) perform all steps of the evolutionary procedure for each subpopulation. To do it, at the beginning, and after a global migration, each worker W_j requests to the master as many subpopulations as devices, N_D, are present in the node (lines 4 and 15). However, the number of subpopulations received, Sp_l, could be lower than N_D if there are not enough subpopulations available to be evolved, as it is shown in lines 5 and 16. Previously, before starting the MPI communications with the master, the centroids, K, must be obtained and the devices, D, initialized. In GPU, the initialization also includes copying some data from the host, such as the centroids and the DS dataset (lines 2-3).

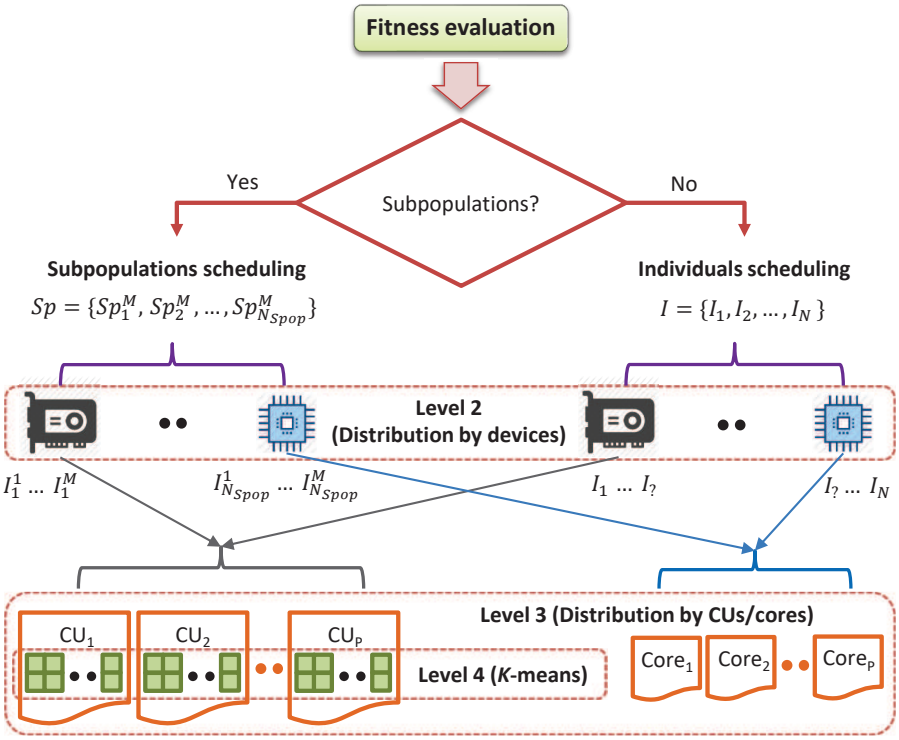


Fig. 2 Fitness evaluation in the devices, providing the third an fourth parallelism levels

In the OpenMP section (line 7), as many CPU threads as received subpopulations, $W_j^t; \forall t = 1, ..., rcv \leq N_D$, are created through the corresponding OpenMP pragma to parallelize the evolutionary steps contained within the function `evolve` (line 10). This way, each subpopulation, Sp_{l_i}, is assigned to one of these CPU threads, which manage and execute one call to the `evolve` function to perform the evolutionary operators for its corresponding subpopulation, such as crossover, mutation, and non-dominated sorting, which are repeated according to the required number of subpopulation generations. However, the evaluation of individuals, also inside the `evolve` function, may be executed either on the free CPU cores or on other accelerators, depending on whether the CPU thread associated with Sp_{l_i} manages the CPU or not.

To summarize, this constitutes the second level of parallelism because the subpopulations received by the workers are assigned to the available devices, either the CPU itself or other devices. If the worker receives only one subpopulation, $rcv = 1$, and one OpenMP thread is created, the second parallelism level is preserved because the `evolve` function detects this situation and dynamically distributes the individuals among all devices. Thus, two dynamic scheduling alternatives for evaluation of individuals are present. Figure 2 illustrates more clearly this situation. In the versions *v1* and *v2*, the worker W_j will not ask for more work to the master until their Sp_l subpopulations

have been computed. However, this strategy caused load imbalance since the devices are not homogeneous, being this imbalance even greater when their computing capabilities differ significantly. Normally some devices will finish their work before others, causing idle time for the most powerful devices, and thus, not only an increase in energy consumption but also a reduction in the acceleration of the algorithm. Thus, in the *v3* version, each thread, W_j^t, now has the capacity to return its subpopulation, Sp_{l_i}, directly to the master, and ask for a new subpopulation (lines 11-12). The cost of introducing this improvement is the elimination of the so-called local migrations implemented in the *v1* and *v2* versions to migrate individuals between the subpopulations of each device. However, we have verified that its elimination does not affect the quality of the solutions, as it is shown in Section 5. Probably, other strategies to cope with load imbalance could be studied to analyze the computing capacity of each device in order to assign more work to those that are more powerful, or modify the clock frequency of the cores to match the running time of the devices, allowing better performance and energy-saving by eliminating the idle time. Finally, once the whole process is finished, the OpenMP section ends (line 14) and the worker W_j waits for the assignment of more work (lines 15-16), or the END_SIGNAL signal (line 17), implying that all N_{Gm} global migrations have been carried out by the master, and the workers can return.

4.3 Distribution of individuals by CUs or CPU cores. K-means algorithm

The third and fourth levels of parallelism occur within the `evolve` function (lines 7 and 10 of Algorithms 1 and 2, respectively). Regardless of whether one or multiple subpopulations are evaluated, the third level works with individuals. In version *v1*, the fitness evaluation is performed by launching OpenCL kernels, as was proposed in [36], and optimized/analyzed in [37,38]. Each device distributes individuals among their processing elements, i.e. CPU cores or Compute Units (CUs), in case of GPUs. Just before calling the kernel, the individuals to be evaluated must be transferred from the host to the devices.

On the other hand, the implementation of K-means for versions *v2* and *v3* is not coded with OpenCL, but in OpenMP. The objective of studying an OpenMP version is to analyze the impact of both implementations in the execution time and energy consumed. Its use is also motivated by the ease of implementation, since a `#pragma omp parallel for` pragma in the loop that iterates over the array of individuals is enough to distribute them through the CPU threads (see Algorithm 3). Although it could be thought that the result will be the same, it must be taken into account that the OpenCL kernels for K-means are compiled and executed at runtime (online) by the OpenCL driver of the CPU, while in the OpenMP versions, *v2* and *v3*, K-means is compiled by the corresponding *C++* compiler. This circumstance, along with the compiler used and the possible optimizations that each one is able to achieve, can determine the difference between choosing one option or the other.

Algorithm 3: Pseudocode of the OpenMP CPU K-means for *v2* and *v3* that evaluates a chunk of individuals

```
1  Function K-means(I, N_I, DS, K, N_T)
       Input  : The individuals to be evaluated, I
       Input  : Number of individuals to be evaluated, N_I
       Input  : Dataset with the EEG patterns, DS
       Input  : Set K of W centroids randomly chosen from DS
       Input  : Number of CPU threads to execute the algorithm, N_T
       Output: f_1(I): intra-cluster distances of I according to Equation (1)
       Output: f_2(I): inter-cluster distances of I according to Equation (2)
2      #pragma omp parallel for num_threads(N_T)
3      for i ← 1 to N_I individuals do
4          K_C ← Create a copy of the centroids
5          Initialization of the mapping table, M_T ← 0
6          repeat
7              M_T ← Patterns in DS are assigned to the cluster
8              D ← Store nearest distance for each pattern
9              Check if any pattern has changed its assignment
10             K_C ← Update the centroids using the dataset DS
11         until stop criterion is not reached;
12         f_1(I_i) ← intraCluster(K_C, DS, D)
13         f_2(I_i) ← interCluster(K_C, DS)
14     end
15     return (f_1(I), f_2(I))
16 End
```

As commented in Section 3, the fitness evaluation is carried out by applying a K-means algorithm over each individual. In CPU, since each individual is assigned to one core, K-means is sequentially executed within that core. In GPU, as each CU is composed by multiple work-items, the data parallelism available in K-means allows the parallelization of both centroids assignment and centroids update, which constitutes the fourth (and last) parallelism level (see Algorithm 4). We are aware that data parallelism is also possible in the CPU cores taking into account the usually available vectorization instructions. It would allow an even more efficient use of the architecture resources although, for simplicity, this will be considered in our future works.

5 Experimental work

In this section, we analyze the performance of our *C++* codes, running on a cluster that executes CentOS (v7.4.1708) and contains four NUMA nodes connected by Gigabit Ethernet. The source codes have been compiled with the GNU compiler (GCC v4.8.5) and the optimization level `-O2` for all executions except of those in Fig. 3, where the Intel compiler (ICC v19.0.0.117) has also been used. The OpenMPI library used (v1.10.7) has support for the MPI API (v3.0.0). When multiple nodes are used, the front-end node is dedicated to the

Algorithm 4: Pseudocode of the OpenCL GPU K-means for *v1*, *v2*, and *v3* that evaluates a chunk of individuals. The expression inside the double angles << *expr* >> defines the distribution of compute units and work-items used in the different steps of the algorithm

```
1  Kernel function K-means(I, N_I, DS, K, DS^t)
       Input  : The individuals to be evaluated, I
       Input  : Number of individuals to be evaluated, N_I
       Input  : Dataset with the EEG patterns, DS
       Input  : Set K of W centroids randomly chosen from DS
       Input  : Dataset DS^t is DS in column-major order
       Output: f_1(I): intra-cluster distances of I according to Equation (1)
       Output: f_2(I): inter-cluster distances of I according to Equation (2)
2      << All compute units, All work-items >>
3      for i ← 1 to N_I individuals do
4          << Compute unitID, All work-items >>
5          K_L ← Copy the centroids from global memory to local memory
6          I_L ← Copy individual I_i from global memory to local memory
7          Initialization of the mapping table, M_T ← 0
8          repeat
9              << Compute unitID, Work-itemID >>
10             M_T ← Patterns in DS^t are assigned to the cluster
11             D ← Store nearest distance for each pattern
12             Check if any pattern has changed its assignment
13             << Compute unitID, All work-items >>
14             K_L ← Update the centroids using the dataset DS
15         until stop criterion is not reached;
16         << Compute unitID, Work-item number 0 >>
17         f_1(I_L) ← intraCluster(K_L, DS, D)
18         f_2(I_L) ← interCluster(K_L, DS)
19     end
20     return (f_1(I), f_2(I))
21 End
```

master process and the others are the workers. The CPU-GPU characteristics of each platform are shown in Tables 2 and 3. To perform the data parallelism of the K-means algorithm in GPU, each CU schedules 1,024 work-items, and thus the total amount of threads, T_t, in each GPU to perform the algorithm is $T_t = CUs \cdot 1,024$.

In our experiments, we have used three datasets from the BCI Laboratory at the University of Essex, described in [39]. They correspond to subjects coded as 104, 107, and 110, and each include 178 EEG patterns with 3,600 features per pattern. However, we only show the results for dataset 110 due to their similar results. The maximum value of the cost functions is $f_1 = f_2 = 1.0$, and the hypervolume metric is calculated according to the Zitzler algorithm [40,41], which uses (0,0) as the reference point, and thus, the maximum value is $hv = 1.0$. We have evaluated 3,840 individuals distributed into 1 to 32 subpopulations, along 150 generations, including global migrations between different nodes (only available in the cluster) and local migrations between

Table 2 CPU characteristics of the platforms used in the experiments

Platform	Model	Cores/Threads	Core (MHz)	RAM
Front-end	2x Intel Xeon E5-2620 v2	12/24	2,100	32 GB
Node 1				
Node 2	1x Intel Xeon E5-2620 v4	8/16		
Node 3	2x Intel Xeon E5-2620 v4	16/32		

Table 3 GPU characteristics of the platforms used in the experiments

Platform	Model	CUs/Cores	Core/Memory (MHz)	RAM
Node 1	1x Tesla K20c	13/2,496	706/5,200	5 GB
Node 2	1x Tesla K40m	15/2,880	745/6,008	12 GB
Node 3				

Table 4 Parameters of the implemented NSGA-II algorithm

Subpopulation	Total individuals (N)	3,840
	Number (N_{Spop})	1 to 32
	Size (M)	N/N_{Spop}
Evolution	Generations (g)	150
	Selection	Binary tournament
Migration	Global (N_{Gm})	5
	Local (N_{Lm})	15
Crossover	Type	Uniform
	Probability	0.75
Mutation	Type	Bit inversion
	Probability	0.025

subpopulations of the same node (only available for versions *v2* and *v3*). Table 4 summarizes the values of the parameters used in the implemented NSGA-II algorithm. Due to the total number of generations is 150, each global migration is performed every 30 generations and one local migration every 10 generations in the cases where they are enabled. All experiments have been repeated 20 times to obtain more reliable measurements on the behavior of the procedure.

The instantaneous power and the energy consumption of the four nodes of our cluster have been measured by a watt-meter we have developed based on Arduino Mega. It provides, in real-time, four measures per second for each node corresponding to the instantaneous power (in Watts) and the cumulated consumed energy (in $W \cdot h$) of the whole node. Moreover, in the measures, the instantaneous power and the energy consumed by the switch, necessary for the communications, are included too (below 5 W). However, as it will be shown in Subsection 5.1, the consumption of the switch is much lower compared to the consumption of the devices.

5.1 Experimental results

Figure 3 provides the measures for running time and energy consumption when using the GCC and Intel compilers, and the optimization levels `-O2` and `-O3`.

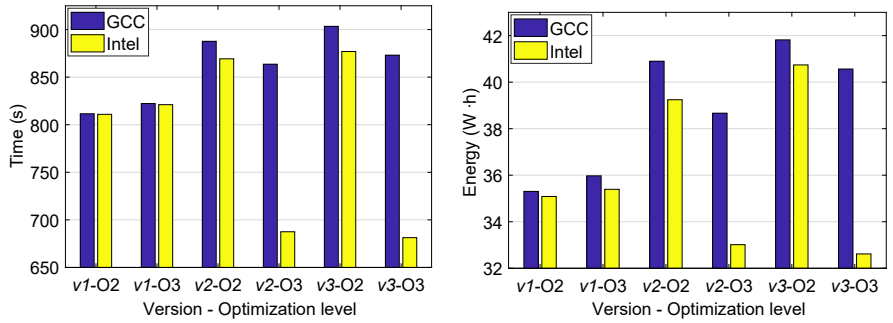
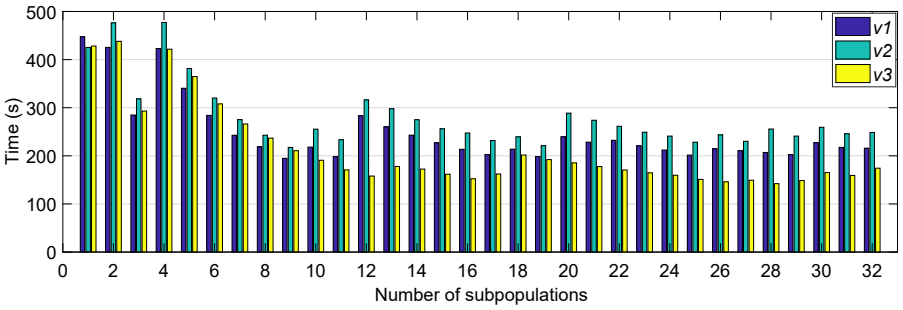


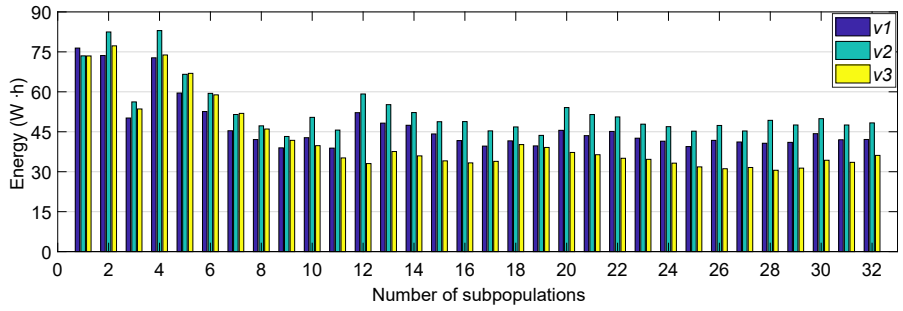
Fig. 3 Comparison of (a) running time and (b) energy consumed obtained in node $N2$ for versions *v1*, *v2*, and *v3* with 1 subpopulation and CPU-only configuration. Code compiled with GCC and Intel compilers, and optimization levels `-O2` and `-O3`

They have been obtained after executing the versions *v1*, *v2*, and *v3* in node $N2$ in an CPU-only configuration. Focusing on the GCC compiler, it can be observed that *v2* and *v3*, which are the same program as *v1* but with OpenMP implementation in CPU, are worse because the Intel OpenCL driver seems to apply better optimizations on the OpenCL code than those provided by GCC on the OpenMP code. Even the use of the optimization level `-O3` is not enough to achieve the performance of *v1*. An inexperienced programmer might think that the compilation with `-O3` cannot be valid in heterogeneous systems since there would be a risk of incompatibility of the executable with the different processors of each node. However, the `mpirun` command offers the possibility to specify a binary file for each MPI process that runs the program. Using this properly can be more or less easy and will depend on the application and how it is executed, as for example the order of the MPI processes or the number of them changes each time the application is executed. On the other hand, when analyzing the behavior of the bars for the Intel compiler, what is observed is that *v1* gets similar execution times since the Intel compiler cannot act on the OpenCL code, and in general all results obtained by the GCC compiler have been improved. This is logical considering that both CPU and compiler have been designed by Intel. The difference is even greater when the optimization level `-O3` has been used, since it seems that the Intel compiler has applied more aggressive optimizations, allowing to versions *v2* and *v3* decrease by approximately 19% the execution time and energy consumed by *v1*.

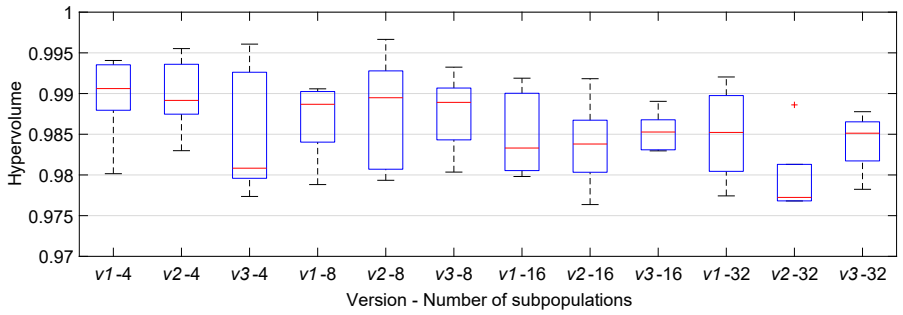
Figure 4 provides the time, energy consumption and solution quality (hypervolume of the Pareto's front obtained by the corresponding version) after executing the versions *v1*, *v2*, and *v3* in all nodes (Tables 2 and 3 show the CPU-GPU characteristics of each node). Figures 4.a and 4.b compare the time and energy for different versions from 1 to 32 subpopulations. From these figures, it is clear that version *v3* provides best values for both running time and energy consumption than *v1* and *v2* from 10 to 32 subpopulations. In the case of from 1 to 9 subpopulations, *v3* is better (or similar) in time and energy con-



(a)



(b)



(c)

Fig. 4 Behavior of versions *v1*, *v2*, and *v3* using all nodes in CPU-GPU configuration. Subpopulations from 1 to 32 for (a) running time and (b) energy consumed; (c) hypervolume (4, 8, 16, 32 subpopulations). Code compiled with GCC and optimization level `-O2`

sumption than *v2*. Moreover, it can be observed that the lowest running time has been obtained when using 28 subpopulations. Figure 4.c only provides the hypervolume boxplots of the Pareto's front for 4, 8, 16, and 32 subpopulations. Nevertheless, we have checked that the obtained hypervolumes from 1 to 32 subpopulations are higher than 0.975, thus corresponding to solutions with adequate quality levels (the maximum hypervolume value is $hv = 1.0$).

As the lowest running time corresponds to the use of 28 subpopulations in version *v3*, we have calculated the equivalent runtime and energy measure

for the sequential version by using the CPU of node $N3$, obtaining $T_{seq} = 11,840.26$ seconds and $E_{seq} = 626.2$ W·h. Thus, the parallel version provides a peak reduction in time by a factor of up to 83, and only requires about a 4.9% of the energy consumed by the sequential procedure.

5.2 An approach to explain the observed behavior for execution time and energy consumption

In what follows, we provide an insight of the reasons that explain the energy consumption and runtime of the implemented workload distribution alternatives, which are experimentally observed and shown in Figure 4. To do that, we use simple time and energy models whose purpose is not to achieve an accurate quantitative prediction of the experimentally measured data but to understand the observed behavior in order to extract conclusions about the best alternative to select according to the power-performance goals. The models here described are based on those proposed in [28] for a population distributed among the available CPU and GPU processing elements of a node to compute the fitness of each individual. The model for runtime starts from the following equation:

$$t = g \cdot \left(N \cdot t_{master} + max \left(\left\lceil \frac{x \cdot N}{P_{GPU}} \right\rceil \cdot t_{GPU}, \left\lceil \frac{(1-x) \cdot N}{P_{CPU}} \right\rceil \cdot t_{CPU} \right) \right) \quad (3)$$

where g is the number of generations, N is the number of individuals in the population, and P_{CPU} and P_{GPU} are, respectively, the CPU cores and GPU CUs in the platform. It is considered that the $x \cdot N$ individuals allocated to the GPU are equally distributed among its CUs, and the $(1-x) \cdot N$ individuals are equally distributed among the CPU cores. The parameter t_{master} corresponds to the time required by one of the cores to process the master task comprising the distribution of individuals among the CPU and GPU processing elements, and the computation of evolutionary operators once the fitness of the individuals have been processed for a given generation. The parameters t_{CPU} and t_{GPU} are, respectively, the time required by the CPU cores and the GPU CUs to evaluate one individual. The parameters t_{master}, t_{CPU} and t_{GPU} can be also expressed as a function of the corresponding workload and the frequency of the corresponding processor as $t_{master} = \frac{W_{master}}{F_{CPU}}$, $t_{CPU} = \frac{W_{CPU}}{F_{CPU}}$, and $t_{GPU} = \frac{W_{GPU}}{F_{GPU}}$, where F_{CPU} and F_{GPU} are the frequencies of, respectively, the CPU and GPU processing elements, and W_{master}, W_{CPU}, and W_{GPU} are respectively, estimations of the cycles of the workloads for the master task, the evaluation of an individual in the CPU, and the evaluation of an individual in the GPU. This way:

$$t = g \cdot \left(N \cdot \frac{W_{master}}{F_{CPU}} + max \left(\left\lceil \frac{x \cdot N}{P_{GPU}} \right\rceil \cdot \frac{W_{GPU}}{F_{GPU}}, \left\lceil \frac{(1-x) \cdot N}{P_{CPU}} \right\rceil \cdot \frac{W_{CPU}}{F_{CPU}} \right) \right) \quad (4)$$

where it is supposed that the times t_{CPU} and t_{GPU} required to evaluate one individual are the same for all individuals evaluated by the corresponding device. The EEG feature selection for BCI here considered can be suitably modelled this way as the evaluation of each individual has been done by a fixed number of iterations of the K-means algorithm. Thus, it can be supposed that the parallelism provided by the GPU gives similar acceleration to the K-means algorithm for all individuals in the population [38]. By using the estimations of W_{master}, W_{CPU}, and W_{GPU} done as in [28], for each node of the cluster and the characteristics shown in Tables 2-4, it is possible to derive an approximate runtime model of the parallel codes for the heterogeneous cluster as follows.

In all *v1*, *v2*, and *v3* versions, the master tasks are executed by a thread running in one of the nodes of the cluster and distributes the subpopulations among the rest of nodes in the cluster, which will execute the generations of the subpopulations allocated to each one along with the local migrations among subpopulations in the same node (only for *v1* and *v2*). Once a given node completes the processing of generations between local migrations corresponding to the subpopulations previously allocated, it asks for more subpopulations to the master. When the subpopulation generations and the evolutionary operators among global migrations have been processed by the corresponding node, the subpopulations are sent to the master node. Then, once the information from all nodes have been received, a new set of subpopulations is built and distributed again among the nodes of the cluster. By the definition of C_{GPU} and C_{CPU} in Equations (5) and (6) as the workload assigned to the GPU and CPU, respectively, the time required by the i-th node to compute the generations of a subpopulation between each global migration can be estimated according to Equation (7):

$$C_{GPU} = x \cdot \left(\frac{N}{N_{Spop}} \right) \cdot \Delta N_{Spop} \tag{5}$$

$$C_{CPU} = (1 - x) \cdot \left(\frac{N}{N_{Spop}} \right) \cdot \Delta N_{Spop} \tag{6}$$

$$T_i = N_{Lm} \cdot \left(g \cdot max \left(\left\lceil \frac{C_{GPU}}{P_{GPU}} \right\rceil \cdot \frac{W_{GPU}}{F_{GPU}}, \left\lceil \frac{C_{CPU}}{P_{CPU}} \right\rceil \cdot \frac{W_{CPU}}{F_{CPU}} \right) + T_{Lm} \right) \tag{7}$$

where N_{Lm} is the number of local migrations among subpopulations in the same node and T_{Lm} the time required to process a local migration. In the previous equation, ΔN_{Spop} the number of subpopulations the master sends to a given node once it asks for more work. The versions *v1* and *v2* use $\Delta N_{Spop} = 2$, while $\Delta N_{Spop} = 1$ in *v3*. This way, the granularity of the interchanges of information between the master node and the nodes that evaluated the subpopulations is coarser for *v1* and *v2* than for *v3*, and in the equation:

$$T_{cluster} = N_{Gm} \cdot (T_{master} + max\,(T_i; \forall i = 1, ..., N_{nodes}) + T_{com}) \tag{8}$$

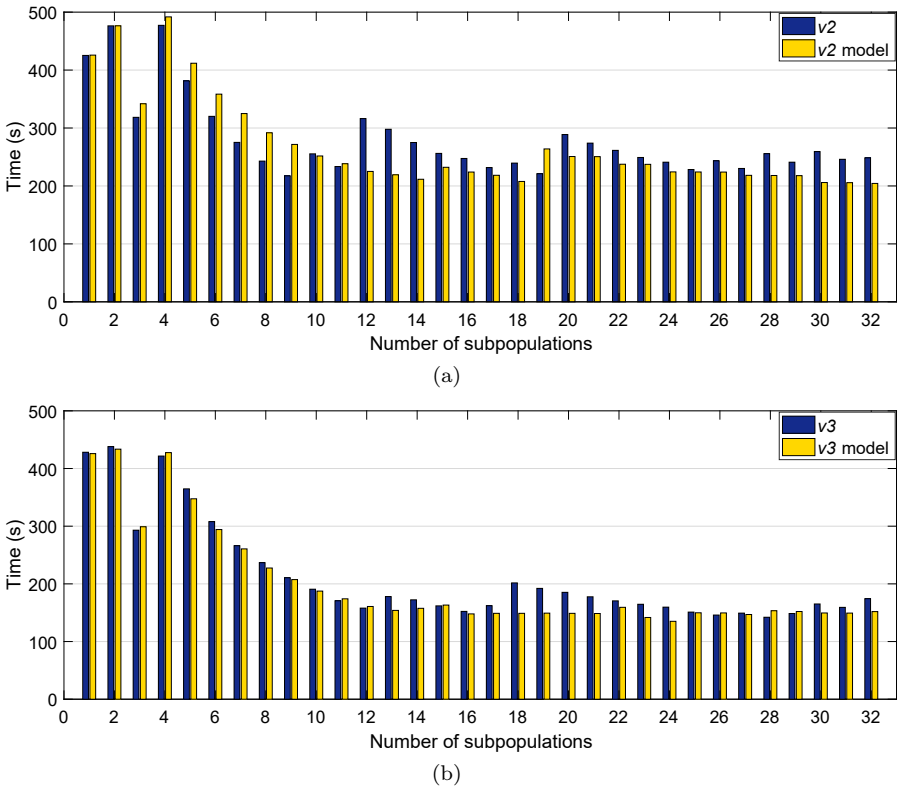


Fig. 5 Comparison of runtime between the experimental data shown in Fig. 4.a and the corresponding model, using all nodes in CPU-GPU configuration. Code compiled with GCC and optimization level `-O2`. Subpopulations from 1 to 32: (a) *v2*; (b) *v3*

that estimates the runtime of the application, $T_{cluster}$, when all N_{Gm} global migrations are completed. The magnitude of the communication time between nodes, T_{com}, is higher for *v1* and *v2* than for *v3*. Figures 5.a and 5.b compare the experimental and modeled running time for the versions *v2* and *v3*, respectively. Although there are significant deviations in the predictions for some cases, the model provides a shape that correctly shows the changes in case of one to six subpopulations, and also shows the presence of sawtooth in the evolution of the running time as the number of subpopulations grows. To quantitatively evaluate the quality of the fit of our model, we have used the Normalized Root-Mean-Squared Error (NRMSE), a standard criteria for regression whose values are between $(-\infty, 1]$ depending on if there is a bad fit between the model and the experimental data or a perfect fit, respectively. Values of $NRMSE = 0.439$ for *v2* and $NRMSE = 0.787$ for *v3* have been obtained (in [42] a value of $NRMSE > 0.2$ is considered as a good result).

With respect to the estimation of the energy consumption, a model based on the product of running times and the average instantaneous power con-

sumed by each node and the switch along different steps of those running times has been used. Thus, the energy consumed by a node of the cluster can be approximated by using the average of their instantaneous power consumed, Pow_i, when it is executing the corresponding workload allocated to it, and the average of the instantaneous power when the node is idle, Pow_i^{idle}. Thus, the energy consumed by the i-th element of the cluster, E_i, could be estimated by:

$$E_i = Pow_i \cdot T_i + Pow_i^{idle} \cdot (T_{cluster} - T_i) \tag{9}$$

This way, the amount of energy consumed by the cluster is obtained by the sum of the energy consumed by all nodes plus the energy consumed by the network. The energy consumed by the network could be estimated from the product of T_{com} and the average of the instantaneous power consumed by the switch when there is communication among nodes, Pow_{SW}, and the product of $(T_{cluster} - T_{com})$ and the average instantaneous power consumed by the network when there is not any communication, Pow_{SW}^{idle}. The quality of this approach to estimate the energy consumption depends on the behavior of the instantaneous power as we are assuming that the instantaneous power of each element of the cluster (nodes and switch in our case) only shows two different average values in its evolution: one when it is doing useful work and another one when it is idle. Of course, this situation does not exactly happen, but if the experimental values are quite close to two different average values, the approach here considered could be useful to estimate the energy consumption.

Figure 6 shows the evolution of the instantaneous power of the nodes and switch of the cluster for versions *v2* and *v3*, using 6 and 32 subpopulations. Some conclusions can be drawn from these figures, that reveal the way the runtime system takes advantage of the resources offered by the processors to achieve a more efficient energy consumption when running this program. The standard Advanced Configuration and Power Interface (ACPI) [43], includes mechanisms to manage and save energy and provides information about the configuration and control of the processor states in terms of energy consumption and performance. In the same way, the Linux kernel implements the infrastructure `Cpufreq` [44] that allows the operating system (either automatically through the events generated by the ACPI or through user program calls) to change the operating frequency of the processor for energy-saving. Although the so-called governors are included in `Cpufreq` to implement specific policies to control the processor clock, in this paper we are more interested on the observation whether to change the workload distribution strategy affects in the energy-saving strategy.

From previous figures, it is clear that the most part of the instantaneous power corresponds to nodes N_1 to N_3 where the main part of the work is being processed. The instantaneous power consumed by the network is quite low in comparison with the energy consumption of the devices. Nodes N_1 to N_3, which present the highest instantaneous power, clearly show two situations corresponding to their working and idle states. The observed instantaneous power exhibits changes in these two situations but these changes are smaller

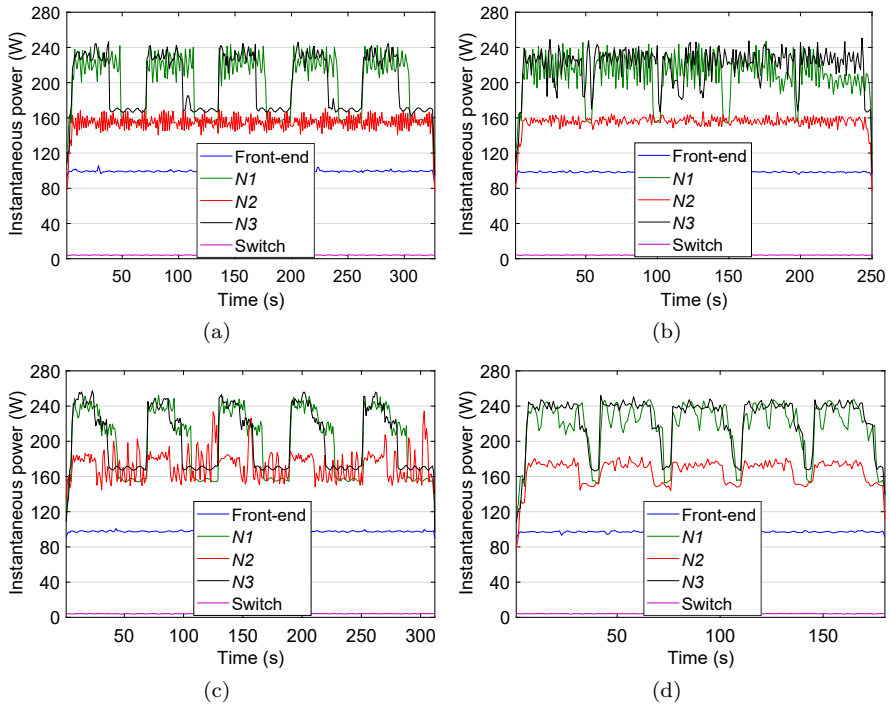


Fig. 6 Instantaneous power using all nodes for versions *v2* and *v3* in CPU-GPU configuration. Code compiled with GCC and optimization level `-O2`: (a) *v2*, 6 subpopulations; (b) *v2*, 32 subpopulations; (c) *v3*, 6 subpopulations; (d) *v3*, 32 subpopulations

than the differences between the mean instantaneous power for these two alternatives. With respect to N_2, the instantaneous power does not clearly presents the behavior shown by N_1 and N_3. In this case, it seems that the instantaneous power oscillates around a value that is lower than those values corresponding to the highest values shown for N_1 and N_2, but similar to the lowest values for N_1 and N_2 (corresponding to their consumption when they are idle).

Figure 7 shows the histograms of the instantaneous power values experimentally measured for all nodes, and versions *v2* and *v3* along with the corresponding density function obtained by a smoothing function fit of the histograms [45]. The number of bins in the histograms is equal to the square root of the number of samples experimentally obtained for each case. As it can be seen, the density functions present two maxima. The two instantaneous power values for these two maxima can be used as estimators of the parameters Pow_i and Pow_i^{idle} used in the energy model for the node N_i: the highest instantaneous power estimates Pow_i, and the lowest estimates Pow_i^{idle}.

Table 5 provides the estimated values of Pow_i and Pow_i^{idle} for versions *v2* and *v3* and all nodes. They have been obtained as the average of the maxima in the density functions corresponding to histograms for 6, 8, 16, 24 and 32 subpopulations. The values show that the estimated values for Pow_i are higher

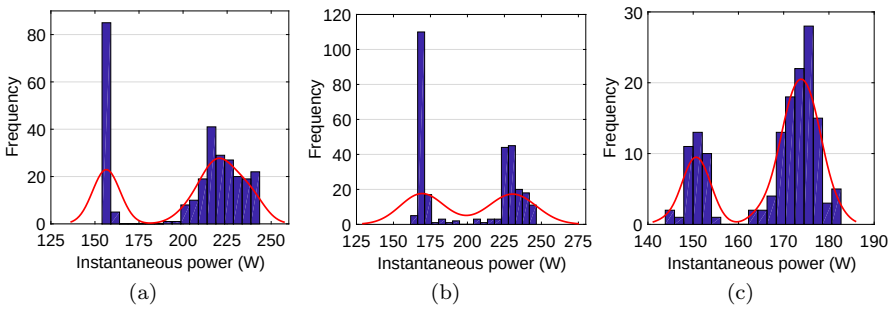


Fig. 7 Histograms of the instantaneous power in CPU-GPU configuration. Code compiled with GCC and optimization level `-O2`. Version *v2* and 6 subpopulations for nodes (a) $N1$ and (b) $N3$. Version *v3* and 32 subpopulations for node (c) $N2$

Table 5 Standard deviation of the estimated Pow_i and Pow_i^{idle} for all nodes in CPU-GPU configuration, and versions *v2* and *v3*. Code compiled with GCC and optimization level `-O2`

Node	Version	Pow_i^{idle} (W)	Pow_i (W)
N_1	*v2*	156.30 ± 0.82	224.72 ± 4.61
	v3	156.58 ± 1.47	234.72 ± 2.65
N_2	*v2*	146.08 ± 8.51	160.01 ± 4.69
	v3	154.38 ± 3.41	177.36 ± 3.21
N_3	*v2*	172.76 ± 6.78	226.38 ± 4.81
	v3	170.14 ± 2.16	244.58 ± 2.93

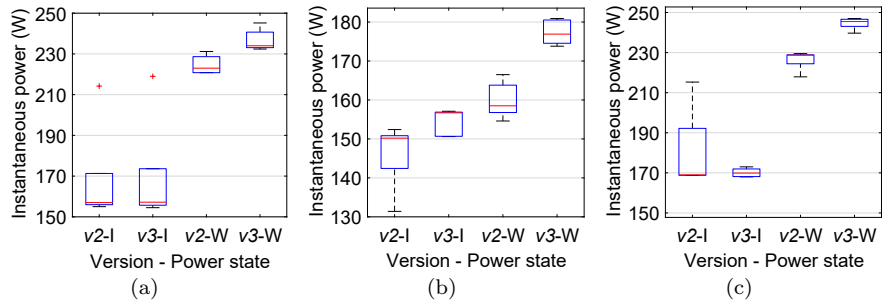
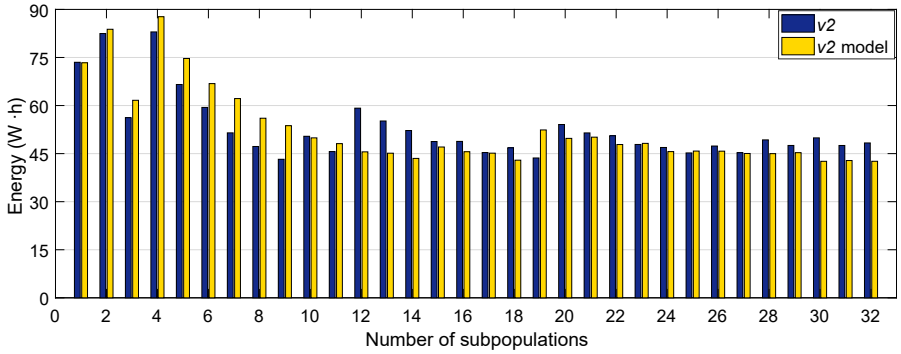


Fig. 8 Graphical comparison of the values shown in Table 5 for the power states (I) Idle and (W) Workload in CPU-GPU configuration: (a) Node $N1$; (b) Node $N2$; (c) Node $N3$. Code compiled with GCC and optimization level `-O2`

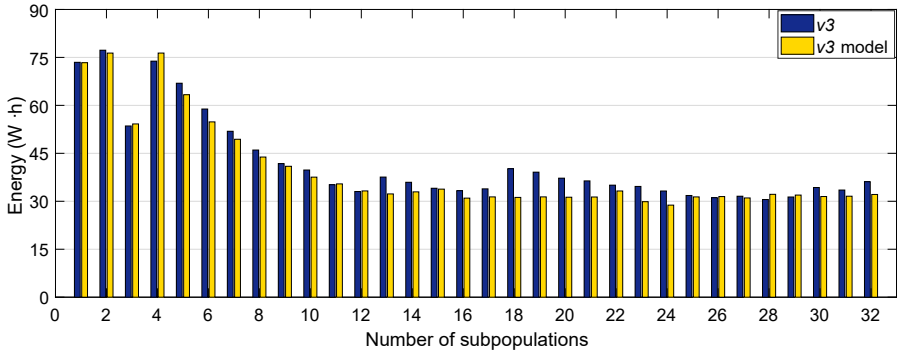
for version *v3* than for *v2*. This trend is similar for Pow_i^{idle} except in one case. Moreover, the differences among the estimated Pow_i^{idle} for *v2* and *v3* are lower than those estimated for Pow_i. All these conclusions have been tested for statistical significance by applying the Kolmogorov-Smirnov and the Kruskal-Wallis tests. Table 6 provides the obtained p-values for the previous boxplots and shows that except for two cases (bold cells), the differences between the other alternatives are significant (p-values below 0.05). Thus, the parameters estimated for Pow_i and Pow_i^{idle} can be used in the approximated models to provide an insight about the experimental results obtained.

Table 6 p-values for boxplots of Fig. 8 (p-values below 0.05 mean significant differences)

Node	Instantaneous power comparison	p-value
N_1	$Pow^{idle}_{N_1}(v2)$ vs $Pow_{N_1}(v2)$	0.008
	$Pow^{idle}_{N_1}(v3)$ vs $Pow_{N_1}(v3)$	0.009
	$Pow^{idle}_{N_1}(v2)$ vs $Pow^{idle}_{N_1}(v3)$	**0.753**
	$Pow_{N_1}(v2)$ vs $Pow_{N_1}(v3)$	0.008
N_2	$Pow^{idle}_{N_2}(v2)$ vs $Pow_{N_2}(v2)$	0.009
	$Pow^{idle}_{N_2}(v3)$ vs $Pow_{N_2}(v3)$	0.009
	$Pow^{idle}_{N_2}(v2)$ vs $Pow^{idle}_{N_2}(v3)$	0.028
	$Pow_{N_2}(v2)$ vs $Pow_{N_2}(v3)$	0.009
N_3	$Pow^{idle}_{N_3}(v2)$ vs $Pow_{N_3}(v2)$	0.008
	$Pow^{idle}_{N_3}(v3)$ vs $Pow_{N_3}(v3)$	0.008
	$Pow^{idle}_{N_3}(v2)$ vs $Pow^{idle}_{N_3}(v3)$	**0.465**
	$Pow_{N_3}(v2)$ vs $Pow_{N_3}(v3)$	0.008



(a)



(b)

Fig. 9 Comparison of energy consumption between the experimental data shown in Fig. 4.b and the corresponding model, using all nodes in CPU-GPU configuration. Code compiled with GCC and optimization level `-O2`. Subpopulations from 1 to 32: (a) *v2*; (b) *v3*

Figures 9.a and 9.b compare the experimental values for the energy consumption with those estimated by the approximate model we have used, respectively for the versions *v2* and *v3*. The values of NRMSE for these two alternatives are 0.403 for *v2*, and 0.744 for *v3*, thus corresponding to acceptable model approximations. As the previously described model for runtime, the model for energy shows similar behavior as the experimental results from one to six subpopulations, and also exhibits the sawtooth behavior when the number of subpopulations changes.

6 Conclusions

Different versions of a parallel procedure for multi-objective feature selection in EEG classification for BCI tasks have been implemented to take advantage of heterogeneous clusters whose nodes include CPU and GPU devices. The corresponding codes use MPI, OpenMP and/or OpenCL libraries to implement message-passing and shared-memory communication and benefit from thread-level and data-level parallelism across a heterogeneous CPU-GPU cluster. Versions *v1*, *v2*, and *v3*, compared in this paper, implement a master-worker evolutionary algorithm based on subpopulations but they differ in their workload distribution strategy and in the use of either OpenCL or OpenMP to evaluate the individuals of each subpopulation when using CPU. Thus, in *v1* the fitness evaluation of the individuals is coded with OpenCL in both CPU and GPU, while in *v2* and *v3* the evaluation in CPU is coded with OpenMP. When these alternative versions are executed by using only CPU, *v1* obtains slightly better results in both time and energy consumption than versions *v2* and *v3* whenever the GCC compiler has been used, because the OpenCL driver used for *v1* applies all optimizations by default, while *v2* and *v3* only take advantage of the relatively less effective GCC optimizations (regardless of the optimization level applied). Nevertheless, when the Intel compiler and the `-O3` optimization level are applied to generate the executable, *v2* and *v3* show relevant decrements in runtime and energy consumed when they are compared to *v1*.

The performances of *v1*, *v2*, and *v3* when they are executed in the cluster show differences in both time and energy consumption as *v1* and *v2* use a different strategy from *v3* to distribute the workload. Thus, although in all versions the subpopulations are dynamically distributed among the nodes, in versions *v1* and *v2* the worker nodes do not ask the master for more subpopulations until the subpopulations assigned to CPU and GPU are computed after completing the local migrations. In *v3*, once a subpopulation is computed, the worker node can ask for a new subpopulation and there is only synchronization among global migrations as the local migrations has been removed (we have experimentally seen this does not affect to the quality of the solutions). This way, the granularity of the information interchanged between master and worker nodes is finer for *v3* than for *v1* and *v2*. These circumstances could explain the improvement in time and energy consumption observed for *v3* with respect to *v1* and *v2*. Approximated models for time and energy consumption

have been fitted by taking into account the characteristics of the nodes in the platform, and the experimental evolution of their instantaneous power, to explain the experimental results. These models provide acceptable fits to the experimental data as they show similar shapes in the sawtooth behavior for *v2* and *v3*, which is caused by workload imbalances. Due to the granularity in *v3* is lower than in *v2*, the sawtooth decrease and the energy-time performance improves for *v3* when increasing the number of subpopulations.

The results show that the proposed parallel approaches are able not only to accelerate the runtime but also to reduce the energy consumption with respect to a sequential implementation. Time has been reduced by a factor higher than 83 with the implemented parallel versions, requiring only about a 4.9% of the energy consumed by the corresponding sequential code when using all nodes of our CPU-GPU cluster. However, new studies could be useful to complete the experimental analysis for new alternatives and experimental situations. For example, a detailed analysis of energy consumption devoted to communications or a better study about about the effect of the different steps of the parallel evolutionary algorithm in the instantaneous power would be also interesting to devise new strategies for energy-efficient programming. In addition, the algorithm must be improved when there are few subpopulations to avoid idle devices and nodes. We are working on a new version in which the granularity in the workload distribution is finer considering three alternatives: (i) discard the scheduling of subpopulations in favor of a more efficient scheduling of individuals, although it may involve more MPI communications; (ii) keep both strategies but prioritize the distribution by nodes instead of by devices. When each node detects that only one subpopulation has been received, the scheduling of individuals is applied and all devices of that node cooperate to evaluate the individuals; (iii) keep both strategies and detect the size of the problem when the program starts, to subsequently apply either the scheduling of subpopulation, or the scheduling of individuals during all execution, although a suitable study should be accomplished to find the threshold that determines which is the optimal approach.

Acknowledgements We would like to thank the BCI laboratory of the University of Essex, especially prof. John Q. Gan, for allowing us to use their databases

References

1. O'brien, K., Pietri, I., Reddy, R., Lastovetsky, A., Sakellariou, R.: A survey of power and energy predictive models in hpc systems and applications. ACM Computing Surveys 50(3), 37:1–37:38 (2017)
2. Zhang, Y., Hu, X., Chen, D.: Task scheduling and voltage selection for energy minimization. In: Proceedings of the 39th Annual Design Automation Conference. pp. 183–188. DAC'2002, ACM, New Orleans, Louisiana, USA (June 2002)
3. Baskiyar, S., Abdel-Kader, R.: Energy aware dag scheduling on heterogeneous systems. Cluster Computing 13(4), 373–383 (2010)
4. Lee, Y., Zomaya, A.: Energy conscious scheduling for distributed computing systems under different operating conditions. IEEE Transactions on Parallel and Distributed Systems 22(8), 1374–1381 (2011)

5. Dorronsoro, B., Nesmachnow, S., Taheri, J., Zomaya, A., Talbi, E.G., Bouvry, P.: A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems. Sustainable Computing: Informatics and Systems 4(4), 252–261 (2014)
6. Barik, R., Farooqui, N., Lewis, B., Hu, C., Shpeisman, T.: A black-box approach to energy-aware scheduling on integrated cpu-gpu systems. In: Proceedings of the 2016 International Symposium on Code Generation and Optimization. pp. 70–81. CGO'2016, ACM, Barcelona, Spain (March 2016)
7. Ortega, J., Asensio-Cubero, J., Gan, J., Ortiz, A.: Classification of motor imagery tasks for BCI with multiresolution analysis and multiobjective feature selection. BioMedical Engineering OnLine 15(1), 149–164 (2016)
8. Raju, K., Niranjan, N.: A survey on techniques for cooperative cpu-gpu computing. Sustainable Computing: Informatics and Systems 19, 72–85 (2018)
9. Mittal, S., Vetter, J.: A survey of methods for analyzing and improving gpu energy efficiency. ACM Computing Surveys 47(2), 19:1–19:23 (2014)
10. Escobar, J., Ortega, J., Díaz, A., González, J., Damas, M.: Speedup and energy analysis of eeg classification for bci tasks on cpu-gpu clusters. In: Proceedings of the 6th International Workshop on Parallelism in Bioinformatics. pp. 33–43. PBIO'2018, ACM, Barcelona, Spain (September 2018)
11. Vidal, P., Alba, E., Luna, F.: Solving optimization problems using a hybrid systolic search on gpu plus cpu. Soft Computing 21(12), 3227–3245 (2017)
12. Luong, T., Melab, N., Talbi, E.G.: Gpu-based island model for evolutionary algorithms. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation. pp. 1089–1096. GECCO'2010, ACM, Portland, OR, USA (July 2010)
13. Pospichal, P., Jaros, J., Schwarz, J.: Parallel genetic algorithm on the cuda architecture. In: Proceedings of the 13th European Conference on the Applications of Evolutionary Computation. pp. 442–451. EvoApplications'2010, Springer, Istambul, Turkey (April 2010)
14. Sharma, D., Collet, P.: Implementation techniques for massively parallel multi-objective optimization. In: Tsutsui, S., Collet, P. (eds.) Massively Parallel Evolutionary Computation on GPGPUs, pp. 267–286. Natural Computing Series, Springer (2013)
15. Wong, M., Cui, G.: Data mining using parallel multi-objective evolutionary algorithms on graphics processing units. In: Tsutsui, S., Collet, P. (eds.) Massively Parallel Evolutionary Computation on GPGPUs, pp. 287–307. Natural Computing Series, Springer (2013)
16. Gainaru, A., Slusanschi, E., Trausan-Matu, S.: Mapping data mining algorithms on a gpu architecture: A study. In: Proceedings of the 19th International Symposium. Foundations of Intelligent Systems. pp. 102–112. ISMIS'2011, Springer, Warsaw, Poland (June 2011)
17. Coello Coello, C., Sierra, M.: A study of the parallelization of a coevolutionary multiobjective evolutionary algorithm. In: Proceedings of the 3rd Mexican International Conference on Artificial Intelligence. pp. 688–697. MICAI'2004, Springer, Mexico City, Mexico (April 2004)
18. Pruhs, K., Stee, R., Uthaisombut, P.: Speed scaling of tasks with precedence constraints. Theory of Computing Systems 43(1), 67–80 (2008)
19. Rotem, E., Weiser, U., Mendelson, A., Ginosar, R., Weissmann, E., Aizik, Y.: H-earth: Heterogeneous multicore platform energy management. IEEE Computer Magazine 49(10), 47–55 (2016)
20. Nesmachnow, S., Dorronsoro, B., Pecero, J., Bouvry, P.: Energy-aware scheduling on multicore heterogeneous grid computing systems. Journal of Grid Computing 11(4), 653–680 (2013)
21. Valentini, G., Lassonde, W., Khan, S., Min-Allah, N., Madani, S., Li, J., Zhang, L., Wang, L., Ghani, N., Kolodziej, J., Li, H., Zomaya, A., Xu, C.Z., Balaji, P., Vishnu, A., Pinel, F., Pecero, J., Kliazovich, D., Bouvry, P.: An overview of energy efficiency techniques in cluster computing systems. Cluster Computing 16(1), 3–15 (2013)
22. Hong, S., Kim, H.: An integrated gpu power and performance model. SIGARCH Computer Architecture News 38(3), 280–289 (2010)

23. Ge, R., Feng, X., Burtscher, M., Zong, Z.: Peach: A model for performance and energy aware cooperative hybrid computing. In: Proceedings of the 11th ACM Conference on Computing Frontiers. pp. 24:1–24:2. CF'2014, ACM, Cagliari, Italy (May 2014)
24. De Sensi, D.: Predicting performance and power consumption of parallel applications. In: Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. pp. 200–207. PDP'2016, IEEE, Heraklion Crete, Greece (February 2016)
25. Marowka, A.: Energy consumption modeling for hybrid computing. In: Proceedings of the 18th International Conference on Parallel Processing, Euro-Par 2012. pp. 54–64. Euro-Par'2012, Springer, Rhodes Island, Greece (August 2012)
26. Ma, K., Li, X., Chen, W., Zhang, C., Wang, X.: Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. In: Proceedings of the 41st International Conference on Parallel Processing. pp. 48–57. ICPP'2012, IEEE, Pittsburgh, PA, USA (September 2012)
27. Allen, T., Ge, R.: Characterizing power and performance of gpu memory access. In: Proceedings of the 4th International Workshop on Energy Efficient Supercomputing. pp. 46–53. E2SC'2016, IEEE Press, Salt Lake City, Utah, USA (November 2016)
28. Escobar, J., Ortega, J., Díaz, A., González, J., Damas, M.: Energy-aware load balancing of parallel evolutionary algorithms with heavy fitness functions in heterogeneous cpu-gpu architectures. Concurrency and Computation: Practice and Experience p. e4688 (2018)
29. Free Software Foundation: Gnu gprof documentation. `https://ftp.gnu.org/pub/old-gnu/Manuals/gprof-2.9.1/html_node/gprof_toc.html` (Accessed: 2017-02-10)
30. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. The Journal of Machine Learning Research 3, 1157–1182 (March 2003)
31. Charikar, M., Guruswami, V., Kumar, R., Rajagopalan, S., Sahai, A.: Combinatorial feature selection problems. In: Proceedings of the 41st Annual Symposium on Foundations of Computer Science. pp. 631–640. FOCS'2000, IEEE, Redondo Beach, CA, USA (November 2000)
32. Khronos Group: Khronos opencl registry. `https://www.khronos.org/registry/cl/` (2015), accessed: 2015-11-30
33. OpenMP Community: Openmp specifications. `http://www.openmp.org/specifications/` (Accessed: 2016-11-21)
34. Escobar, J., Ortega, J., Díaz, A., González, J., Damas, M.: Multi-objective feature selection for eeg classification with multi-level parallelism on heterogeneous cpu-gpu clusters. In: Proceedings of the Annual Conference on Genetic and Evolutionary Computation. pp. 1862–1869. GECCO'2018, ACM, Kyoto, Japan (July 2018)
35. The Open MPI Project: Openmpi documentation. `https://www.open-mpi.org/doc/` (Accessed: 2018-11-19)
36. Escobar, J., Ortega, J., González, J., Damas, M.: Assessing parallel heterogeneous computer architectures for multiobjective feature selection on eeg classification. In: Proceedings of the 4th International Conference on Bioinformatics and Biomedical Engineering. pp. 277–289. IWBBIO'2016, Springer, Granada, Spain (April 2016)
37. Escobar, J., Ortega, J., González, J., Damas, M.: Improving memory accesses for heterogeneous parallel multi-objective feature selection on eeg classification. In: Proceedings of the 4th International Workshop on Parallelism in Bioinformatics. pp. 372–383. PBIO'2016, Springer, Grenoble, France (August 2016)
38. Escobar, J., Ortega, J., González, J., Damas, M., Díaz, A.: Parallel high-dimensional multi-objective feature selection for eeg classification with dynamic workload balancing on cpu-gpu. Cluster Computing 20(3), 1881–1897 (2017)
39. Asensio-Cubero, J., Gan, J., Palaniappan, R.: Multiresolution analysis over simple graphs for brain computer interfaces. Journal of Neural Engineering 10(4), 21–26 (2013)
40. Zitzler, E., Thiele, L.: Multiobjective optimization using evolutionary algorithms - a comparative case study. In: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature. pp. 292–301. PPSN V, Springer, Amsterdam, The Netherlands (September 1998)
41. Zitzler, E.: Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications. Shaker Verlag Germany (1999)

42. Sîrbu, A., Babaoglu, O.: Power consumption modeling and prediction in a hybrid cpu-gpu-mic supercomputer. In: Proceedings of the 22nd International Conference on Parallel Processing, Euro-Par 2016. pp. 117–130. Euro-Par'2016, Springer, Grenoble, France (August 2016)
43. Advanced Configuration and Power Interface (ACPI): Acpi specification. http://www.acpi.info/spec.htm (Accessed: 2018-11-30)
44. CPUFreq Governors: Information for users and developers. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt (Accessed: 2018-11-30)
45. Mathworks: Matlab histfit function. https://mathworks.com/help/stats/histfit.html (Accessed: 2018-12-02)

Juan José Escobar received the M.Sc. degree in computer science from the University of Granada, Spain, in 2014. Currently he is a Ph.D. student at the Department of Computer Architecture and Technology of the University of Granada. His main research interests include code optimization, energy-efficient parallel and distributed computing, and workload balancing strategies, specially in issues related to the development of heterogeneous parallel algorithms for evolutionary multi-objective feature selection problems

Julio Ortega received his B.Sc. degree in Electronic Physics in1985, M.Sc. degree in Electronics in 1986, and Ph.D. degree in 1990 from the University of Granada, Spain. His Ph.D. dissertation has received the Award of Ph.D. dissertations of the University of Granada. He was at the Open University, U.K., Department of Electronics (University of Dortmund, Germany), and Department of Computer Science and Electrical Engineering (University of Essex, UK), as invited researcher. Currently he is a Full Professor at the Department of Computer Architecture and Technology of University of Granada and Senior Member of the IEEE Computer Society. He has published more than 200 technical papers and contributions to international conferences. His research interests include the processing of parallel computer architectures, multi-objective optimization, neural networks, and evolutionary computation

Antonio F. Díaz received the M.S. degree in electronic physics in 1992 and the Ph.D. degree in 2001, both from the University of Granada, Spain. In 1993, he was at the Institute National Polythechnique of Grenoble, France as visitor researcher. Currently, he is an Associate Professor in the Department of Computer Architecture and Technology, the University of Granada. His research interests are in the fields of: cluster computing, GPUs, high performance mass storage and parallel I/O

Jesús González received the M.A.Sc. degree in Computer Engineering in 1997 and the Ph.D. degree in 2001, both with honors, from the University of Granada, Spain. He is currently an Associate Professor within the Department of Computer Architecture and Computer Technology in the University of Granada. His current areas of interest are related to the fields of embedded systems, neural networks, and evolutionary computation

Miguel Damas received the M.Sc. degree (1991) and Ph.D. degree (2000) in computer engineering, both with honours from the University of Granada (Spain). Associate professor in the Department of Computer Architecture and Computer Technology of the University of Granada from 2001. He currently teaches in the electrical engineering degree, in the computer engineering degree, and in the Master of Data Science and Computer Engineering. From 1990, he belongs to the research group CASIP (Circuits and Systems for Information Processing) and has collaborated with several companies in research and consultory activities. He has authored more than 80 technical papers in areas of interest related to industrial control and communications, human activity recognition systems, machine learning, and parallel programming for optimization problems