Algoritmo Evolutivo Multiobjetivo con Paralelismo Multinivel para Clasificación de EEGs: Análisis Energía-tiempo en Clústeres Heterogéneos

Juan José Escobar[1], Antonio Francisco Díaz[1], Miguel Damas[1], Beatriz Prieto-Campos[1], Marco Antonio Formoso[2], Raúl Baños[3] y Jesús González[1]

Resumen— **Las arquitecturas heterogéneas actuales interconectan nodos con múltiples microprocesadores y aceleradores multinúcleo que permiten diferentes estrategias para acelerar las aplicaciones y optimizar su consumo de energía. En este trabajo se propone un procedimiento paralelo multinivel que aprovecha todos los nodos de un clúster CPU-GPU heterogéneo. Se han implementado tres versiones diferentes, que han sido analizadas en términos de tiempo de ejecución y consumo energético. Aunque el trabajo considera un algoritmo maestro-trabajador evolutivo para selección de características y clasificación de EEGs, las conclusiones del análisis experimental se pueden extrapolar a otras aplicaciones en bioinformática y minería de datos con el mismo perfil de cómputo que el problema considerado aquí. El enfoque paralelo propuesto permite reducir el tiempo de ejecución en un factor de hasta 83 con sólo un 4,9 % de la energía consumida por el procedimiento secuencial.**

Palabras clave— **Algoritmo maestro-trabajador, Paralelismo multinivel, Análisis energía-tiempo, Clasificación de EEGs, Programación híbrida, Arquitecturas CPU-GPU heterogéneas.**

I. INTRODUCCIÓN

LAS tareas de aprendizaje automático para problemas de clasificación, clustering, selección de características y optimización están presentes en muchas aplicaciones útiles de la bioingeniería. Estas aplicaciones generalmente requieren plataformas de alto rendimiento cuyo coste, en términos económicos y ambientales, debe tenerse muy en cuenta. De hecho, minimizar el consumo de energía se ha convertido en uno de los principales temas de investigación en las Ciencias de la Computación, pues eficiencia hoy en día no sólo significa una buena ganancia en velocidad sino también un consumo de energía óptimo.

El uso de plataformas de cómputo heterogéneas, incluidas las CPUs multinúcleo y otros aceleradores con diferentes arquitecturas como las GPUs (Unidades de Procesamiento Gráfico), posibilitaría códigos paralelos más rápidos que aprovecharían diferentes tipos de paralelismo. Sin embargo, aunque los diferentes ratios de velocidad y potencia de los procesadores actuales brindan más posibilidades para distribuir la carga de trabajo y optimizar el consumo energético, su equilibrado es mucho más complejo de resolver [1]. Por esta razón, los enfoques que tienen en cuenta criterios de aceleración y consumo de energía constituyen un importante tema de investigación [2,3]. Además, existe un problema con el crecimiento exponencial de los datos generados por las aplicaciones, que es imposible de afrontar sin el desarrollo de nuevas técnicas de procesamiento. Por lo tanto, el uso de clústeres de cómputo considerando una estrecha cooperación CPU-GPU [4], como la distribución de la carga de trabajo aquí considerada, constituye actualmente el enfoque principal para aprovechar las mejoras tecnológicas y así superar la barrera impuesta por la limitación del hardware [5].

En este trabajo se ilustra esta situación a través de tres versiones de un enfoque multiobjetivo para clasificación de Electroencefalogramas (EEGs) en tareas BCI. El algoritmo evolutivo multiobjetivo implementa un procedimiento de tipo envoltura (*wrapper*) para la selección de características, y un procedimiento no supervisado que evalúa la aptitud (*fitness*) de los individuos de cada subpoblación [6]. Una de las tres versiones de dicho procedimiento ya fue presentada en [7]. Las otras dos son una evolución de ésta y constituyen la principal contribución de este trabajo. Uno de los algoritmos propuestos es una versión mejorada, optimizada y más rápida que el algoritmo propuesto en [7], mientras que en el otro los cambios están relacionados con la librería utilizada para paralelizar el código y se presenta como una alternativa que es mejor o peor según el compilador utilizado.

Tras esta introducción, la Sección II resume los trabajos relacionados con implementaciones paralelas de algoritmos evolutivos en plataformas CPU-GPU y su correspondiente evaluación energética. La sección III presenta el enfoque multiobjetivo evolutivo para selección de características en la clasificación de EEGs. La sección IV describe nuestro enfoque paralelo multinivel junto con sus versiones propuestas, mientras que la sección V proporciona el trabajo experimental llevado a cabo. Finalmente, en la Sección VI se presentan las conclusiones del trabajo.

[1]Grupo EFFICOMP. Dpto. de Arquitectura y Tecnología de Computadores, CITIC, Universidad de Granada (España). https://efficomp.ugr.es/. E-mails: {jjescobar, afdiaz, jesusgonzalez, beap, mdamas}@ugr.es

[2]Dpto. de Ingeniería de Comunicaciones, Universidad de Málaga (España). E-mail: marco.a.formoso@ic.uma.es

[3]Dpto. de Ingeniería, Universidad de Almería (España). E-mail: rbanos@ual.es

II. ESTADO DEL ARTE

Aunque en la literatura se han propuesto muchas contribuciones sobre implementaciones paralelas de algoritmos evolutivos en plataformas CPU-GPU, la mayoría de ellas no explotan completamente las capacidades de cómputo de ambos dispositivos ya que sólo usan una hebra de la CPU para controlar la actividad de la GPU [8]. Un ejemplo de ello puede verse en [9], donde se describe una implementación CUDA de un algoritmo genético paralelo basado en un modelo de islas. Dicho estudio es un ejemplo de enfoques que, como el presentado en este trabajo, modifican el algoritmo evolutivo para obtener la versión que mejor se adapte a la arquitectura disponible.

No hay muchos enfoques que utilicen, como en este trabajo, la CPU y la GPU como recursos que puedan considerarse por igual a la hora de distribuir la carga de trabajo del procedimiento de optimización. En [8] se propone una metodología para resolver problemas de optimización en arquitecturas CPU-GPU heterogéneas que se beneficia de ambos dispositivos y señala la utilidad de seguir investigando sobre ello. Sin embargo, nuestro enfoque incluye una optimización multiobjetivo evolutiva y un algoritmo de clustering aplicado a un conjunto de EEGs de alta dimensionalidad. Aunque el uso de arquitecturas heterogéneas se ha propuesto en trabajos anteriores, la paralelización sobre una plataforma heterogénea de toda una aplicación de minería de datos, con las características de nuestra aplicación y el análisis de rendimiento energía-tiempo, es menos frecuente. En [10] se analiza el efecto de factores como los patrones de comunicación y la partición de datos en el rendimiento de las aplicaciones de minería de datos, y en [11] se describe un procedimiento evolutivo multiobjetivo paralelo que utiliza MPI en una sola plataforma. Nuestro enfoque aprovecha los clústeres heterogéneos, distribuyendo la carga de trabajo entre los dispositivos CPU y GPU de cada nodo para acelerar la aplicación a la vez que se ahorra energía.

Se han publicado varios trabajos sobre procedimientos de planificación que tienen en cuenta tanto el tiempo de ejecución del programa como el consumo de energía [12–14]. La mayoría de ellos se basan en el Dynamic Voltage Scaling (DVS), una técnica similar a DVFS pero que sólo permite escalar dinámicamente el voltaje de la CPU para reducir el consumo de energía. Aunque una reducción en el voltaje del procesador podría conllevar un aumento en el tiempo de ejecución de las tareas asignadas, es posible evitarlo si estos incrementos se producen sólo en las hebras que ejecutan las tareas más ligeras, para así igualarse con las más pesadas. En este sentido, en [15] se proporciona información sobre técnicas de equilibrado de carga conscientes de la energía.

Con respecto a la eficiencia del consumo de energía de las plataformas híbridas CPU-GPU, en [16–18] se proporcionan algunos resultados sobre este tema. Por ejemplo, en [16] se describen modelos analíticos para obtener una idea de la ganancia en rendimiento y el consumo de energía en diferentes plataformas CPU-GPU, concluyendo que un mayor paralelismo brinda oportunidades para ahorrar energía y fomenta el desarrollo de aplicaciones paralelas para dicho objetivo.

III. ALGORITMOS EVOLUTIVOS MAESTRO-TRABAJADOR PARA CLASIFICACIÓN DE EEGs

La clasificación de EEGs, al igual que otras aplicaciones en bioinformática de alta dimensionalidad, requiere procesar una gran cantidad de datos cuando las muestras que componen los conjuntos de datos están definidas por un gran número de características, provocando que los tiempos de ejecución sean muy altos o prohibitivos. Por ello, es común aplicar procesamiento paralelo y selección de características para disminuir el tiempo de ejecución y/o mejorar la calidad de las soluciones. En estas aplicaciones, un enfoque de tipo *wrapper* basado en algoritmos evolutivos, como el desarrollado en este trabajo, suele implicar la evaluación del fitness de una población de soluciones sobre el problema a tratar. Generalmente, en un algoritmo evolutivo casi todo el tiempo de ejecución corresponde a la función que calcula la aptitud de los individuos. En el procedimiento propuesto en este trabajo, se ha comprobado con la herramienta `gprof` [19] que dicha función consume un 99,93 % del tiempo al evaluar 120 individuos y un 98,60 % con 15.000 individuos. Por lo tanto, como la evaluación del fitness es completamente independiente para cada individuo de la población y es el paso que más tiempo requiere, los esfuerzos para paralelizar el algoritmo deben centrarse en esta tarea.

Este trabajo analiza el tiempo de ejecución y consumo energético de tres enfoques paralelos multinivel para selección de características multiobjetivo (MOFS) y clasificación de EEGs. En los problemas MOFS los datos suelen estar definidos por un elevado número de características, por lo que es necesario utilizar metaheurísticas paralelas que reduzcan tanto el tiempo de ejecución como la energía consumida por los algoritmos. El enfoque maestro-trabajador propuesto está basado en el algoritmo NSGA-II y es responsable de evolucionar una o varias subpoblaciones de individuos. Además, utiliza el algoritmo K-medias como método no supervisado para evaluar la solución aportada por cada individuo. El fitness multiobjetivo de los individuos está compuesto por dos funciones de coste, f_1 y f_2, que pretenden minimizar y maximizar las distancias intraclúster (WCSS) e interclúster (BCSS), respectivamente:

$$f_1 = \sum_{j=1}^{W} \sum_{P_i \in C^t(j)} \left\| P_i - K^t(j) \right\|^2 \tag{1}$$

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

donde W es el número de clústeres y $\|P_i - K^t(j)\|^2$ es la distancia euclídea entre el punto P_i y el centroide $K^t(j)$. Como cada individuo realiza una o va-

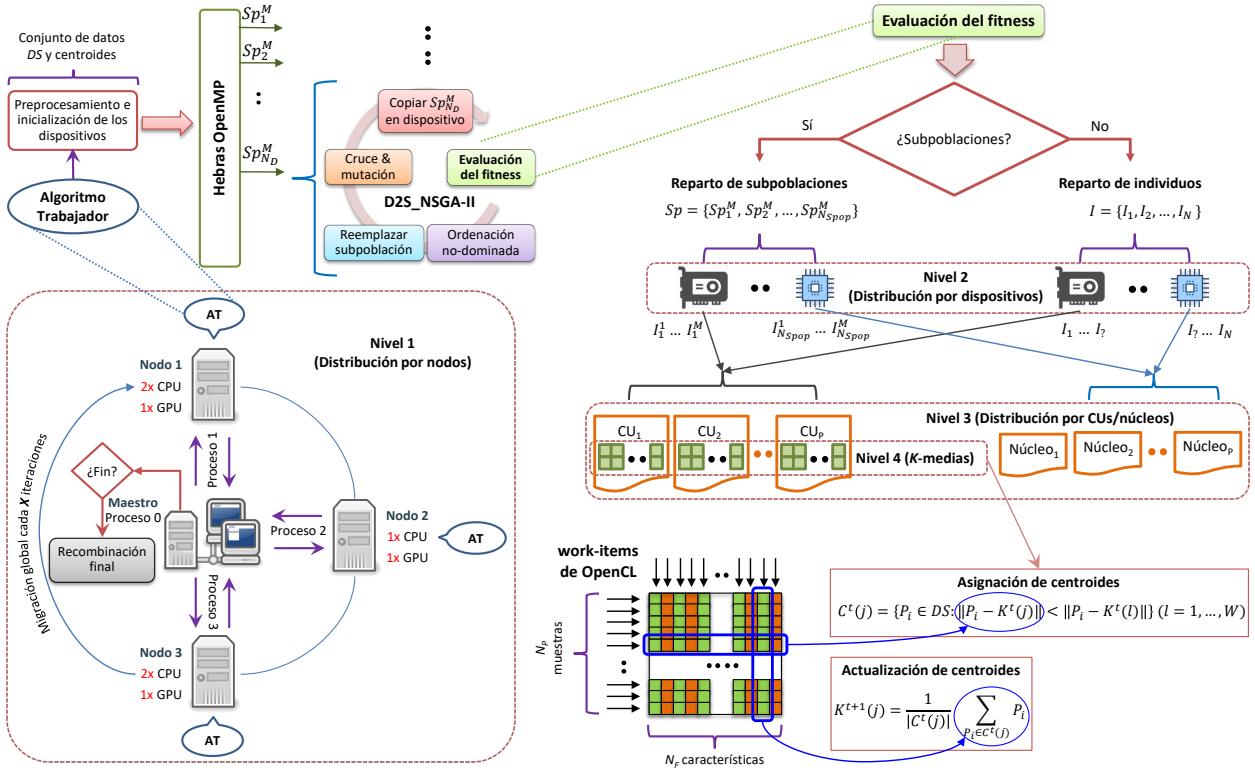


Fig. 1: Esquema MPI-OpenMP del procedimiento evolutivo que proporciona un paralelismo multinivel de hasta 4 niveles según los recursos utilizados para evaluar el fitness de los individuos.

rias ejecuciones del algoritmo K-medias y la distancia euclídea es una operación altamente paralelizable, un procedimiento maestro-trabajador que distribuya individuos o subpoblaciones enteras entre los nodos del clúster podría ser un buen enfoque para extraer el máximo paralelismo.

IV. ENFOQUE PARALELO MULTINIVEL HETEROGÉNEO

El procedimiento maestro-trabajador paralelo propuesto para la selección de características multiobjetivo aprovecha hasta cuatro niveles de paralelismo dependiendo de los dispositivos utilizados para evaluar el fitness de los individuos. Concretamente, en este trabajo se comparan tres versiones diferentes del procedimiento, que difieren en la distribución de la carga de trabajo y también en el uso de OpenCL [20] u OpenMP [21] para paralelizar en la CPU. En lo que sigue, estas versiones se llamarán *v1*, *v2* y *v3*. El pseudocódigo y un análisis extenso para la versión *v1*, la cual se usará en este trabajo como referencia, se proporcionan en [22]. Aquí, por tanto, se describen las versiones *v2* y *v3*, siendo ésta última la más optimizada de las tres.

La versión *v1* utiliza OpenCL, tanto en CPU como en GPU, para implementar el algoritmo K-medias, el cual se encarga de evaluar el fitness de los individuos. La versión *v2* es similar a *v1*, con la diferencia de que el algoritmo K-medias está codificado para CPU con directivas OpenMP en lugar de con OpenCL. Con respecto a la versión propuesta (*v3*), se han añadido algunas mejoras en la distribución de la carga de trabajo con respecto a las versiones *v1* y *v2*. Esto permitirá reducir tanto el tiempo de ejecución como la energía consumida por el procedimiento.

Las siguientes subsecciones detallan los niveles de paralelismo implementados en las tres versiones analizadas así como sus diferencias. La Figura 1 resume todos los pasos de *v3*, desde la creación y distribución de subpoblaciones por el proceso maestro, hasta la evaluación del fitness de los individuos en los procesos que actúan como trabajadores. Todos los pasos de la figura se repiten varias veces, dependiendo del número de subpoblaciones a evolucionar y de cuántas migraciones globales se hayan establecido.

A. Distribución de subpoblaciones entre nodos

El primer nivel de paralelismo corresponde a una distribución dinámica de las subpoblaciones entre los diferentes trabajadores (nodos) del clúster. El enfoque ha sido implementado con la biblioteca de paso de mensajes OpenMPI. Se ha empleado esta estrategia en lugar de una distribución estática para evitar desbalanceos en la carga de trabajo. En primer lugar, si se observa el Algoritmo 1, el maestro (proceso MPI con rango número 0) inicializa los individuos de las subpoblaciones (Línea 2). En la Línea 3, usando una sentencia `if-else` el programa comprueba el número total de procesos MPI que la aplicación está ejecutando. Si la condición se cumple, el maestro será responsable de realizar todo el trabajo, es decir, de evolucionar todas las subpoblaciones utilizando los dispositivos disponibles del nodo al que pertenece y de migrar individuos entre las subpoblaciones (Líneas 6-9). Dado que en este caso el maestro también incluye el rol del trabajador, previamente

Algoritmo 1: Pseudocódigo del algoritmo maestro en *v3*. Las subpoblaciones son distribuidas por el maestro entre todos los nodos trabajadores. Si no hay trabajadores, el maestro evaluará todas las subpoblaciones.

```
1  Función Maestro(N_Spop, M, N_Gm, N_W, N_D)
       Entrada: Número de subpoblaciones, N_Spop
       Entrada: Tamaño de la subpoblación, M
       Entrada: Número de migraciones, N_Gm
       Entrada: Número de trabajadores, N_W
       Entrada: Número de dispositivos, N_D
       Salida  : La métrica de hipervolumen, hv
2      Sp ← obtenerSubpoblaciones(N_Spop, M, DS)
3      si N_W == 0 entonces
4          DS ← obtenerConjuntoDatos()
5          D ← inicializarDispositivos(DS, N_D)
6          para i ← 1 a N_Gm migraciones hacer
7              Sp ← evolucionar(Sp, N_Spop, M, D, N_D)
8              Sp ← migracionGlobal(Sp, N_Spop, M)
9          fin
       // Distribución dinámica de subpoblaciones
10     en otro caso
11         para i ← 1 a N_Gm migraciones hacer
12             tareas ← N_Spop
13             repetir
14                 MPI::Recv(R) ← Del trabajador W_n
15                 env ← min(tareas, R)
16                 MPI::Isend(Sp, env) → A W_n
17                 tareas ← tareas − env
18             hasta que los N_W trabajadores tengan
                 trabajo || tareas == 0;
19             repetir
20                 MPI::Recv(Sp) ← Del trabajador W_n^j
21                 env ← min(tareas, 1)
22                 MPI::Send(Sp, env) → A W_n^j
23                 tareas ← tareas − env
24             hasta que tareas == 0;
25             Sp ← migracionGlobal(Sp, N_Spop, M)
26         fin
           // Finalizar las comunicaciones MPI
27         MPI::Bcast(FIN) → A los N_W trabajadores
28     fin
29     Sp ← agruparSubpoblaciones(Sp, N_Spop, M)
30     hv ← obtenerHipervolumen(Sp, N_Spop, M)
31     devolver hv
32 Fin
```

Algoritmo 2: Pseudocódigo del algoritmo trabajador en *v3*. Las subpoblaciones recibidas del maestro se distribuyen entre los dispositivos. Si sólo se recibe una subpoblación, sus individuos serán repartidos dinámicamente entre los N_D dispositivos.

```
1  Función Trabajador(M, j, N_D)
       Entrada: Tamaño de la subpoblación, M
       Entrada: ID del trabajador, n
       Entrada: Número de dispositivos, N_D
2      DS ← obtenerConjuntoDatos()
3      D ← inicializarDispositivos(DS, N_D)
       // Solicitud de subpoblaciones
4      MPI::Isend(N_D) → Pedir subpoblaciones
5      MPI::Recv(Sp) ← rcv subpoblaciones recibidas
6      repetir
           // rcv trabajadores, W_n^j; ∀j ∈ [1, rcv] ∩ ℕ
7          #pragma omp parallel num_threads(rcv)
8              repetir
9                  Sp_j ← evolucionar(Sp_j, 1, M, D, N_D)
10                 MPI::Isend(Sp_j, 1) → Al maestro
11                 MPI::Recv(Sp_j) ← Nueva subpoblación
12             hasta que no se reciba una nueva Sp_j;
13         fin
14         MPI::Isend(N_D) → Pedir subpoblaciones
15         MPI::Recv(Sp) ← rcv subpoblaciones o FIN
16     hasta que se reciba la señal FIN;
17 Fin
```

debe obtener el conjunto de datos DS e inicializar los dispositivos (Líneas 4 y 5, respectivamente).

En las versiones *v1* y *v2* no existe la alternativa en la que el maestro pueda tener el rol del trabajador, por lo que para estas versiones se necesitan dos procesos MPI como mínimo. Esto no sería relevante si existiera más de un nodo de cómputo, ya que uno de ellos albergaría al proceso maestro y el otro al proceso trabajador. Sin embargo, cuando sólo hay un nodo de cómputo sigue siendo obligatorio un proceso MPI que actúe como maestro y otro que haga de trabajador. Lógicamente, esto presenta un problema de saturación de recursos en la CPU y la memoria, ya que cada proceso MPI se asigna a un núcleo lógico del mismo nodo y, por tanto, hay menos recursos de cómputo disponibles para evaluar el fitness de los individuos. Además, se deben tener en cuenta otras sobrecargas, como el paso de mensajes entre procesos MPI o los requisitos de sincronización, que también afectan al tiempo de ejecución y a la energía consumida. Todos estos problemas han sido corregidos en la versión *v3*, dotando al maestro la capacidad de hacer todo el trabajo.

Si la sentencia `if-else` es falsa, significa que el maestro no evolucionará las subpoblaciones. En este punto, el maestro y los trabajadores estarían sincronizados y listos para comenzar a comunicarse. La distribución de subpoblaciones a evolucionar y la migración global se repiten tantas veces como migraciones globales, N_{Gm}, se hayan establecido (Líneas 11-26). En primer lugar, el maestro envía a todos los nodos un primer paquete de subpoblaciones menor o igual a la cantidad de subpoblaciones solicitadas por el trabajador (Líneas 13-18). Con esto, la idea es hacer que todos los nodos estén ocupados lo antes posible para tratar de reducir el desbalanceo de carga. Posteriormente, el maestro continúa de forma asíncrona atendiendo las peticiones de cada trabajador y distribuye dinámicamente las subpoblaciones restantes hasta que no quede más trabajo por hacer (Líneas 19-24). Acto seguido, procede a realizar una migración global (Línea 25) entre todas las N_{Spop} subpoblaciones para mejorar su diversidad y escapar de los óptimos locales, lo que mejorará la calidad de las soluciones. Una migración global implica crear un nuevo conjunto de subpoblaciones recibiendo soluciones del resto de subpoblaciones. Más concretamente, cada subpoblación aporta como máximo la mitad de las soluciones de su frente de Pareto.

Una vez que todas las N_{Spop} subpoblaciones han evolucionado y se han completado todas las N_{Gm} migraciones globales, el maestro envía la señal FIN a los trabajadores para notificar que no hay más subpoblaciones que evolucionar y que las comunicacio-

nes mediante MPI han terminado (Línea 27). Posteriormente, las soluciones obtenidas por las diferentes subpoblaciones se fusionan en la Línea 29 para componer la subpoblación final, que incluirá a los mejores individuos de cada subpoblación. Finalmente, se calcula la métrica del hipervolumen y se devuelve a la función principal (Líneas 30 y 31).

B. *Distribución de individuos o subpoblaciones entre dispositivos*

Si hay más de un proceso MPI ejecutándose, mientras el proceso maestro se encarga de distribuir las subpoblaciones, los trabajadores (Algoritmo 2) realizan todos los pasos del procedimiento evolutivo para cada subpoblación. Tanto al inicio del procedimiento como tras una migración global, cada trabajador W_n solicita al maestro tantas subpoblaciones como N_D dispositivos estén presentes en el nodo (Líneas 4 y 14), aunque el número de subpoblaciones recibidas, Sp_j, podría ser inferior a N_D si no hay suficientes, tal y como se muestra en las Líneas 5 y 15. Previamente, antes de iniciar las comunicaciones MPI con el maestro, el trabajador obtiene el conjunto de datos DS e inicializa los dispositivos (Líneas 2 y 3).

En la Línea 7, con la directiva `#pragma omp parallel` se crean tantas hebras en la CPU como subpoblaciones recibidas, W_n^j; $\forall j \in [1, rcv] \cap \mathbb{N}$ para paralelizar la función `evolucionar` (Línea 9). Por tanto, cada hebra de la CPU tendrá asignada una única subpoblación Sp_j y llamará a dicha función para aplicar los operadores evolutivos sobre sus individuos (cruce, mutación y ordenación no dominada), los cuales se repetirán según el número de generaciones establecido. Con respecto a la evaluación de individuos, también incluida dentro de la función `evolucionar`, puede llevarse a cabo en las hebras CPU que queden libres o en una GPU, dependiendo de si la hebra CPU asociada con la subpoblación Sp_j administra a la propia CPU o una GPU.

En resumen, esto constituye el segundo nivel de paralelismo: las subpoblaciones recibidas en cada nodo trabajador se asignan a los dispositivos disponibles en dicho nodo, ya sea la CPU u otro acelerador. Si el nodo trabajador sólo recibiese una subpoblación y sólo se crease una hebra OpenMP ($rcv = 1$), el segundo nivel de paralelismo se conservaría porque la función `evolucionar` detecta esta situación y distribuiría los individuos de dicha subpoblación entre todos sus dispositivos. Por lo tanto, existen dos alternativas para repartir de forma dinámica la evaluación de los individuos. En las versiones *v1* y *v2*, el nodo trabajador W_n no pedirá más trabajo al nodo maestro hasta que se hayan evolucionado sus Sp subpoblaciones. Sin embargo, dado que los nodos trabajadores no son homogéneos, esta estrategia provoca un desequilibrio de carga que se acentúa cuando las capacidades de cómputo de sus dispositivos difieren significativamente. Esto quiere decir que algunos de ellos terminarán su trabajo antes que otros, lo que provocará tiempos ociosos en los dispositivos más potentes y, por tanto, una menor aceleración del algoritmo. Es por ello que en la versión *v3*, cada hebra trabajadora W_n^j tiene la capacidad de devolver su subpoblación Sp_j directamente al maestro y solicitar una nueva (Líneas 10-5). El coste de introducir esta mejora es la eliminación de las llamadas migraciones locales implementadas en las versiones *v1* y *v2*, las cuales tenían por objetivo migrar individuos entre las subpoblaciones de cada dispositivo. Se ha comprobado que esta modificación no afecta a la calidad de las soluciones, tal y como se muestra en la Sección V. Por último, el trabajador W_n espera la asignación de más trabajo o la señal FIN (Líneas 14 y 15), lo cual implica que todas las N_{Gm} migraciones globales han sido realizadas por el maestro y que los trabajadores ya pueden terminar (Línea 16).

C. *Distribución de individuos entre elementos de procesamiento*

Los niveles de paralelismo tercero y cuarto ocurren dentro de la función `evolucionar` (Líneas 7 y 9 de los Algoritmos 1 y 2, respectivamente). Independientemente de si se evalúan una o varias subpoblaciones, el tercer nivel trabaja con individuos. En la versión *v1*, la evaluación del fitness se realiza mediante el lanzamiento de *kernels* OpenCL, tal y como fue propuesto en [23] y posteriormente optimizado en [24,25]. Cada dispositivo distribuye individuos entre sus elementos de procesamiento, es decir, núcleos CPU o unidades de cómputo (CU) en el caso de las GPUs.

Por otro lado, la implementación del K-medias en las versiones *v2* y *v3* no está codificada con OpenCL, sino con OpenMP. El objetivo de estudiar ambas implementaciones es analizar cómo afectan al tiempo de ejecución y la energía consumida. Su uso también está motivado por la facilidad de implementación, ya que una única directiva `#pragma omp parallel for` en el bucle que itera sobre el vector de individuos es suficiente para distribuirlos entre las hebras de la CPU. Aunque podría pensarse que el resultado será el mismo, hay que tener en cuenta que los *kernels* OpenCL se compilan y ejecutan en tiempo de ejecución por el driver OpenCL de la CPU, mientras que en las versiones OpenMP (*v2* y *v3*), el K-medias es compilado por el compilador *C++* correspondiente. Esta circunstancia, junto con el compilador utilizado y las posibles optimizaciones que cada uno sea capaz de lograr, determinarán la diferencia entre elegir una opción u otra.

Tal y como se comentó en la Sección III, la evaluación del fitness se lleva a cabo aplicando el algoritmo K-medias sobre cada individuo. En la CPU, dado que cada individuo está asignado a una hebra, el K-medias se ejecuta secuencialmente dentro de ella. Sin embargo, en la GPU, como cada CU está compuesta por múltiples work-items, tanto la asignación de puntos a centroides como la actualización de éstos se pueden acelerar mediante paralelismo de datos, lo que constituiría el cuarto (y último) nivel de paralelismo. Aunque el paralelismo de datos también es posible en la CPU utilizando vectorización, por simplicidad se considerará en futuros trabajos.

Tabla I: Características de los dispositivos CPU-GPU de los nodos utilizados en los experimentos.

Nodo	**CPU**			**GPU**		
	Modelo	**Núcleos/Hebras**	**Frecuencia (MHz)**	**Modelo**	**CUs/Núcleos**	**Frecuencia (MHz)**
Maestro	2x Intel Xeon E5-2620 v2	12/24	2.100	-	-	-
N1				1x Tesla K20c	13/2.496	706
N2	1x Intel Xeon E5-2620 v4	8/16		1x Tesla K40m	15/2.880	745
N3	2x Intel Xeon E5-2620 v4	16/32				

Tabla II: Parámetros del algoritmo NSGA-II implementado.

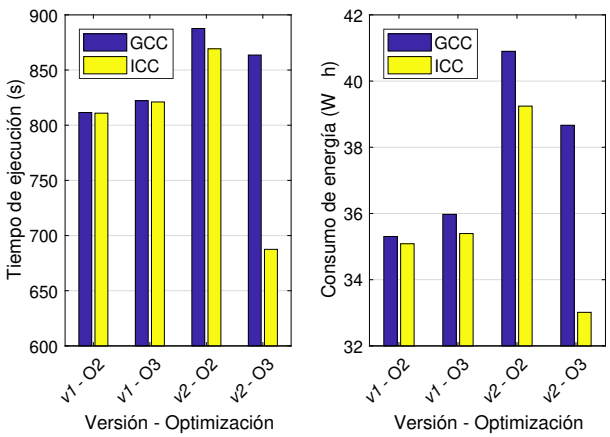
Subpoblación	Individuos totales (N)	3.840
	Número (N_{Spop})	1 a 32
	Tamaño (M)	N/N_{Spop}
Evolución	Generaciones (g)	150
	Torneo	Binario
Migración	Global (N_{Gm})	5
	Local (N_{Lm})	15
Cruce	Tipo	Uniforme
	Probabilidad	0,75
Mutación	Tipo	Inversión del bit
	Probabilidad	0,025

V. TRABAJO EXPERIMENTAL

En esta sección se analiza el rendimiento de los procedimientos en un clúster que ejecuta CentOS (v7.4.1708) y que contiene cuatro nodos NUMA conectados mediante Gigabit Ethernet. Los códigos fuente han sido compilados con el compilador GNU (GCC v4.8.5) salvo para las ejecuciones de la Figura 2, donde también se ha utilizado el compilador de Intel (ICC v19 .0.0.117). La biblioteca OpenMPI utilizada (v1.10.7) tiene soporte para la API MPI v3.0.0. Al utilizar varios nodos, uno de ellos se encargará del proceso maestro y los demás actuarán como trabajadores. Las características CPU-GPU de cada plataforma se muestran en la Tabla I.

Para los experimentos se ha utilizado el conjunto de datos del Laboratorio BCI de la Universidad de Essex [26] correspondiente al sujeto humano número 110. Éste incluye 178 señales EEG de 3.600 características. La métrica de hipervolumen se ha calculado mediante el algoritmo de Zitzler [27], usando (0,0) como punto de referencia. Dado que el valor máximo de las funciones de coste es $f_1 = f_2 = 1$, el valor máximo del hipervolumen será $hv = 1$. Se han evaluado 3.840 individuos distribuidos entre 1 y 32 subpoblaciones a lo largo de 150 generaciones. Las migraciones globales se han realizado cada 30 generaciones, y las locales cada 10 en las versiones en las que estén habilitadas (*v2* y *v3*). Todos los experimentos se han repetido 20 veces para obtener medidas más fiables del comportamiento del procedimiento. La Tabla II resume los valores de los parámetros utilizados en el algoritmo NSGA-II implementado [28].

La potencia instantánea y el consumo de energía de los cuatro nodos del clúster se han medido con un vatímetro basado en Arduino Mega que ha sido desarrollado específicamente para la experimentación.



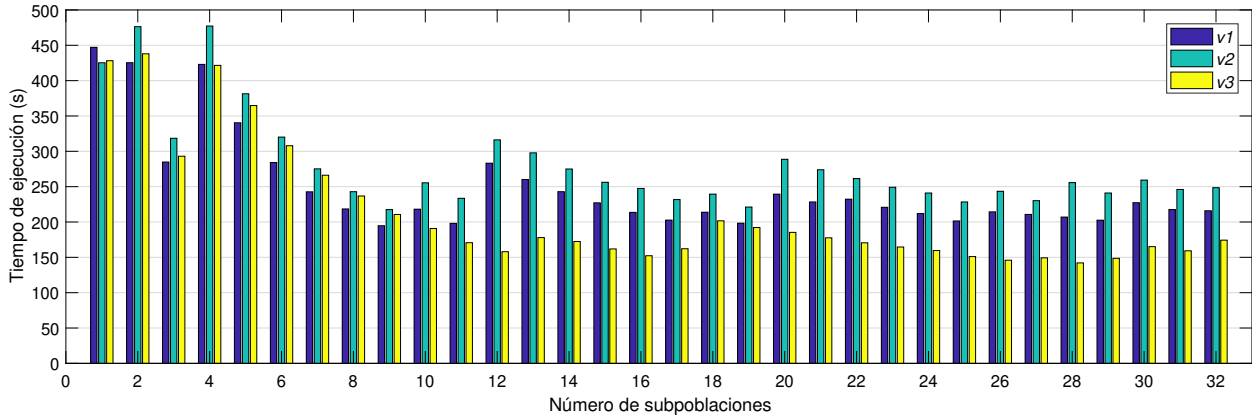
(a) Tiempo de ejecución. (b) Consumo de energía.

Fig. 2: Comparativa de rendimiento entre compiladores al ejecutar las versiones *v1* y *v2* sólo con la CPU del nodo *N3* para evaluar 1 subpoblación.

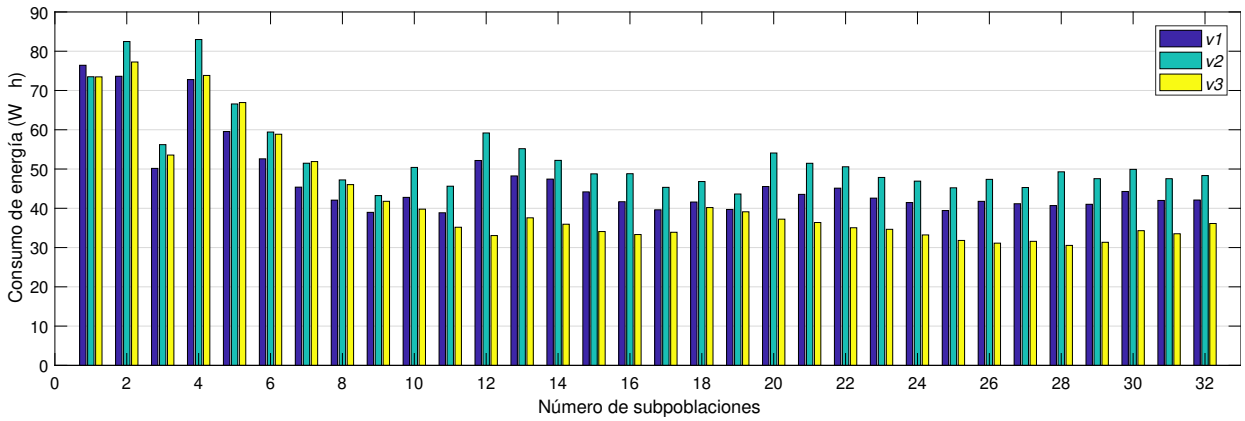
Este medidor es capaz de proporcionar para cada nodo, en tiempo real, una lectura de la potencia instantánea (en Vatios) y otra de la energía consumida acumulada (en W·h). En las medidas también se incluyen la potencia instantánea y la energía consumida por el switch que conecta todos los nodos entre sí, aunque tal y como se mostrará en la Sección V-A, el consumo del switch es mucho menor en comparación con el de los dispositivos.

A. Resultados experimentales

La Figura 2 muestra tanto el tiempo de ejecución como el consumo de energía de las versiones *v1* y *v2* al compilarlas con GCC e ICC y los niveles de optimización `-O2` y `-O3`. Para el experimento sólo se ha utilizado la CPU del nodo $N3$ ya que los compiladores actúan exclusivamente sobre el código host, por lo que las GPUs de ese nodo han sido desconectadas. Poniendo el foco primero en el compilador GCC, la Figura 2 muestra que *v2* no supera a *v1*. El driver de OpenCL parece aplicar mejores optimizaciones en su código que las que GCC puede aplicar en el código OpenMP. Incluso el uso del nivel de optimización `-O3` no es suficiente para lograr el rendimiento de *v1*. Un programador inexperto podría pensar que la compilación con `-O3` no es posible en sistemas heterogéneos ya que el archivo binario generado no se ejecutaría correctamente. Sin embargo, MPI permite al usuario



(a) Tiempo de ejecución.



(b) Consumo de energía.

Fig. 3: Comportamiento energía-tiempo de las versiones *v1*, *v2* y *v3* al utilizar las CPUs y GPUs de todos los nodos para evaluar hasta 32 subpoblaciones. Código compilado con GCC y nivel de optimización `-O2`.

especificar un archivo binario para cada proceso que ejecuta el programa, aunque, dependiendo de la aplicación, esto puede ser bastante tedioso. Por ejemplo, en *v2* el orden de los procesos al iniciar la aplicación es importante ya que cada uno es mapeado a un nodo siguiendo el orden de la lista de hosts.

Por otro lado, lo que se observa al usar el compilador ICC es que todos los resultados obtenidos por GCC han sido mejorados o igualados. Para *v1* se obtienen tiempos de ejecución similares, ya que el compilador no puede actuar sobre el código OpenCL. La diferencia entre ambos compiladores es aún mayor cuando se emplea el nivel de optimización `-O3`. Esto significa que ICC aplica optimizaciones más agresivas, lo que ha permitido que la versión *v2* reduzca el tiempo de ejecución y el consumo de energía de *v1* en aproximadamente un 19 %. Este comportamiento es el esperado, teniendo en cuenta que tanto la CPU como el compilador han sido diseñados por el mismo fabricante (Intel Corporation).

La Figura 3 proporciona el tiempo y consumo de energía tras ejecutar las versiones *v1*, *v2* y *v3* en todos los nodos para evaluar hasta 32 subpoblaciones. Teniendo en cuenta los datos mostrados, está claro que la versión *v3* proporciona los mejores valores en tiempo y energía de 10 a 32 subpoblaciones, demostrando que el nuevo balanceo de carga es más eficiente a medio y largo plazo. En el caso de 1 a 9

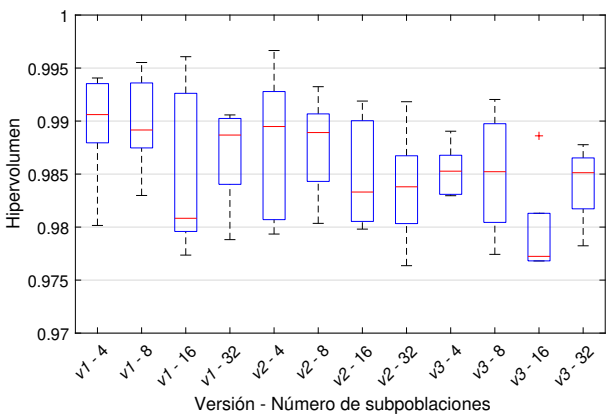
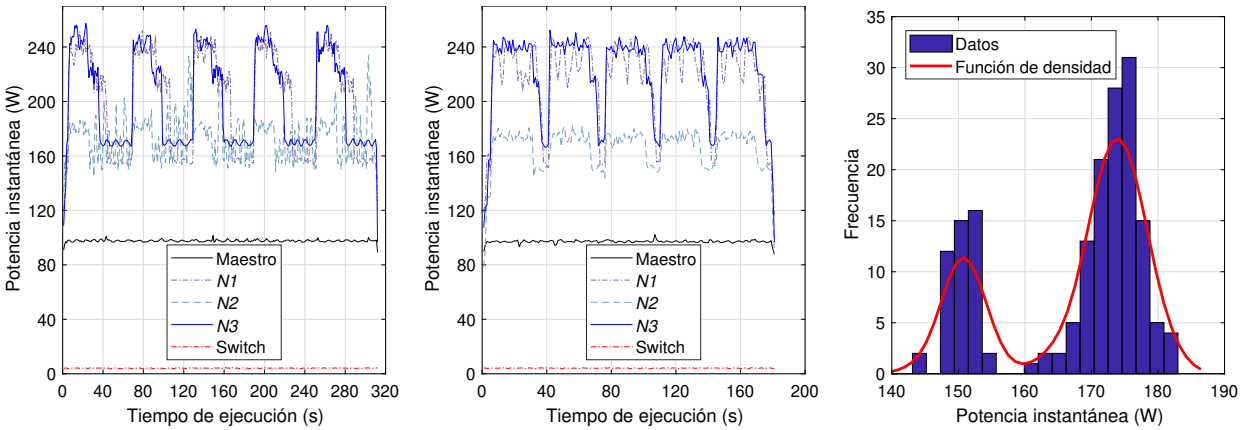


Fig. 4: Hipervolumen obtenido en las versiones *v1*, *v2* y *v3* al utilizar las CPUs y GPUs de todos los nodos para evaluar 4, 8, 16 y 32 subpoblaciones. Código compilado con GCC y nivel de optimización `-O2`.

subpoblaciones, *v3* mejora algunas veces a *v1* pero nunca es superada por *v2*. También se puede observar que el tiempo de ejecución más bajo se ha obtenido al utilizar 28 subpoblaciones. Para este caso, se han calculado las medidas energía-tiempo de la versión secuencial de *v3* usando la CPU del nodo *N3*. Se ha obtenido $T_{seq} = 11.840{,}26$ segundos y $E_{seq} = 626{,}2$ W·h. Por tanto, la versión paralela proporciona un factor de reducción de tiempo de hasta 83 requiriendo sólo alrededor de un 4,9 % de la energía consumida



(a) Todos los nodos. 6 subpoblaciones. (b) Todos los nodos. 32 subpoblaciones. (c) Nodo *N2*. 32 subpoblaciones.

Fig. 5: Evolución temporal de la potencia instantánea e histograma de la versión *v3* al utilizar las CPUs y GPUs de los nodos involucrados. Código compilado con GCC y nivel de optimización `-O2`.

por el procedimiento secuencial. En cuanto a la calidad de las soluciones, la Figura 4 proporciona los diagramas de cajas del hipervolumen obtenido para 4, 8, 16 y 32 subpoblaciones. No obstante, se ha comprobado que los valores obtenidos para el resto de casos son superiores a 0,975. Estos resultados son bastante aceptables teniendo en cuenta que el valor máximo del hipervolumen es $hv = 1$.

Por otro lado, las Figuras 5a y 5b muestran la evolución de la potencia instantánea de cada nodo y del switch para la versión *v3* al evaluar 6 y 32 subpoblaciones. De estas cifras se puede deducir que el sistema aprovecha los recursos que ofrecen los procesadores para disminuir el consumo energético cuando la carga de trabajo es baja. Esto se consigue mediante el uso de la interfaz avanzada de configuración y energía (ACPI) [29], la cual incluye mecanismos para administrar y ahorrar energía y proporciona información sobre la configuración y estados del procesador. En el caso de la Figura 5, el descenso de la carga de trabajo se puede producir por dos causas diferentes: cuando los nodos trabajadores están esperando a que el proceso maestro lleve a cabo una migración global, o bien cuando el procesador deba esperar a que la GPU del mismo nodo finalice su tarea.

También se puede observar que la potencia instantánea consumida por el switch es bastante baja en comparación con la de los nodos, mientras que el nodo maestro presenta unos valores intermedios muy similares a los obtenidos en reposo. La mayor parte de la potencia instantánea corresponde a los nodos $N1$, $N2$ y $N3$, ya que son los que procesan la mayor parte del trabajo. Estos nodos muestran claramente dos escenarios de energía: trabajo y estado ocioso. Sin embargo, el nodo $N2$ no presenta el comportamiento mostrado por los nodos $N1$ y $N3$. En este caso, los datos oscilan en torno a los valores mínimos de los otros dos nodos. Esto se debe a que el nodo $N2$ no dispone de dos CPUs, como sí ocurre en los otros dos nodos, por lo que está justificado que el consumo de energía mostrado sea inferior. La Figura 5c muestra el histograma de la potencia instantánea. Aquí se puede observar que es más frecuente encontrar valores altos de energía, estados que corresponden a cuando los dispositivos del nodo $N2$ están computando. Sin embargo, también existe una frecuencia considerable en valores inferiores, lo cual significa que los dispositivos también pasan tiempo en estado ocioso por tener que esperar a que el nodo maestro realice las migraciones globales.

VI. CONCLUSIONES

Se han analizado diferentes versiones de un MOFS paralelo para clasificación de EEGs que permite aprovechar los clústeres heterogéneos con dispositivos CPU y GPU. Los códigos correspondientes hacen uso de las bibliotecas MPI, OpenMP y OpenCL para conseguir paralelismo mediante paso de mensajes y memoria compartida. Aunque todas las versiones implementan un algoritmo evolutivo maestro-trabajador basado en subpoblaciones, difieren en la estrategia de distribución de la carga de trabajo y en el uso de OpenCL u OpenMP para evaluar a los individuos cuando se utiliza la CPU. En este escenario, se ha puesto de manifiesto la importancia de la elección del compilador en función de la opción de optimización y librería de programación utilizada.

El rendimiento de cada versión es diferente en tiempo y energía en función de las estrategias de equilibrado de carga que implementan. Así, aunque en todas las versiones las subpoblaciones se distribuyen dinámicamente entre los nodos, la granularidad de la información intercambiada entre los nodos maestro y trabajador es más fina para *v3*. Esta diferencia podría explicar la mejora de rendimiento observada para *v3* con respecto a *v1* y *v2*.

Los resultados muestran que los enfoques paralelos propuestos son capaces de disminuir el tiempo de ejecución y consumo de energía. El tiempo de la versión *v3*, al usar todos los nodos para evaluar 28 subpoblaciones, se ha reducido en un factor superior a 83 requiriendo sólo alrededor de un 4,9 % de la energía consumida por el código secuencial equivalente. Aún así, explorar nuevas alternativas y situaciones expe-

rimentales podrían ser útiles para seguir mejorando el procedimiento. Por ejemplo, un análisis detallado del perfil energético de las comunicaciones entre nodos permitiría diseñar nuevas estrategias de cómputo energéticamente eficientes. Además, el algoritmo aquí propuesto podría mejorarse para evitar estados ociosos en los nodos y dispositivos cuando existan pocas subpoblaciones que evolucionar. Actualmente estamos trabajando en una nueva versión en la que la granularidad de la distribución de la carga de trabajo es más fina. Esto es, se descarta el reparto de subpoblaciones en favor de un reparto de individuos más eficiente, aunque ello implique más comunicaciones MPI. Por último, también se está vectorizando para CPU las distancias euclídeas presentes en el algoritmo K-medias.

AGRADECIMIENTOS

Este trabajo ha sido financiado por el Ministerio de Ciencia, Innovación y Universidades y por fondos FEDER a través del proyecto PGC2018-098813-B-C31. También agradecemos al laboratorio BCI de la Universidad de Essex por el acceso a sus conjuntos de datos de EEGs.

Referencias

[1] K. O'brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in hpc systems and applications," *ACM Computing Surveys*, vol. 50, no. 3, pp. 37:1–37:38, 2017.

[2] Y.C. Lee and A.Y. Zomaya, "Energy conscious scheduling for distributed computing systems under different operating conditions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 8, pp. 1374–1381, 2011.

[3] B. Dorronsoro, S. Nesmachnow, J. Taheri, A.Y. Zomaya, E-G. Talbi, and P. Bouvry, "A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems," *Sustainable Computing: Informatics and Systems*, vol. 4, no. 4, pp. 252–261, 2014.

[4] K. Raju and N.C. Niranjan, "A survey on techniques for cooperative cpu-gpu computing," *Sustainable Computing: Informatics and Systems*, vol. 19, pp. 72–85, 2018.

[5] S. Mittal and J.S Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," *ACM Computing Surveys*, vol. 47, no. 2, pp. 19:1–19:23, 2014.

[6] J. Ortega, J. Asensio-Cubero, J.Q. Gan, and A. Ortiz, "Classification of motor imagery tasks for BCI with multiresolution analysis and multiobjective feature selection," *BioMedical Engineering OnLine*, vol. 15, no. 1, pp. 149–164, 2016.

[7] J.J. Escobar, J. Ortega, A.F. Díaz, J. González, and M. Damas, "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*, Barcelona, Spain, September 2018, PBIO'2018, pp. 33–43, ACM.

[8] P. Vidal, E. Alba, and F. Luna, "Solving optimization problems using a hybrid systolic search on gpu plus cpu," *Soft Computing*, vol. 21, no. 12, pp. 3227–3245, 2017.

[9] P. Pospichal, J. Jaros, and J. Schwarz, "Parallel genetic algorithm on the cuda architecture," in *Proceedings of the 13th European Conference on the Applications of Evolutionary Computation*, Istambul, Turkey, April 2010, EvoApplications'2010, pp. 442–451, Springer.

[10] A. Gainaru, E. Slusanschi, and S. Trausan-Matu, "Mapping data mining algorithms on a gpu architecture: A study," in *Proceedings of the 19th International Symposium. Foundations of Intelligent Systems*, Warsaw, Poland, June 2011, ISMIS'2011, pp. 102–112, Springer.

[11] C.A. Coello Coello and M. Sierra, "A study of the parallelization of a coevolutionary multi-objective evolutionary algorithm," in *Proceedings of the 3rd Mexican International Conference on Artificial Intelligence*, Mexico City, Mexico, April 2004, MICAI'2004, pp. 688–697, Springer.

[12] K. Pruhs, R. Stee, and P. Uthaisombut, "Speed scaling of tasks with precedence constraints," *Theory of Computing Systems*, vol. 43, no. 1, pp. 67–80, 2008.

[13] E. Rotem, U.C. Weiser, A. Mendelson, R. Ginosar, E. Weissmann, and Y. Aizik, "H-earth: Heterogeneous multicore platform energy management," *IEEE Computer Magazine*, vol. 49, no. 10, pp. 47–55, 2016.

[14] S. Nesmachnow, B. Dorronsoro, J.E. Pecero, and P. Bouvry, "Energy-aware scheduling on multicore heterogeneous grid computing systems," *Journal of Grid Computing*, vol. 11, no. 4, pp. 653–680, 2013.

[15] G.L. Valentini, W. Lassonde, S.U. Khan, N. Min-Allah, S.A. Madani, J. Li, L. Zhang, L. Wang, N. Ghani, J. Kolodziej, H. Li, A.Y. Zomaya, C-Z. Xu, P. Balaji, A. Vishnu, F. Pinel, J.E. Pecero, D. Kliazovich, and P. Bouvry, "An overview of energy efficiency techniques in cluster computing systems," *Cluster Computing*, vol. 16, no. 1, pp. 3–15, 2013.

[16] A. Marowka, "Energy consumption modeling for hybrid computing," in *Proceedings of the 18th International Conference on Parallel Processing, Euro-Par 2012*, Rhodes Island, Greece, August 2012, Euro-Par'2012, pp. 54–64, Springer.

[17] T. Allen and R. Ge, "Characterizing power and performance of gpu memory access," in *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing*, Salt Lake City, Utah, USA, November 2016, E2SC'2016, pp. 46–53, IEEE Press.

[18] J.J. Escobar, J. Ortega, A.F. Díaz, J. González, and M. Damas, "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.

[19] Free Software Foundation, "Gnu gprof documentation," https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_node/gprof_toc.html, Accessed: 2022-05-11.

[20] Khronos Group, "Khronos opencl registry," https://www.khronos.org/registry/cl/, 2015, Accessed: 2015-11-30.

[21] OpenMP Community, "Openmp specifications," http://www.openmp.org/specifications/, Accessed: 2016-11-21.

[22] J.J. Escobar, J. Ortega, A.F. Díaz, J. González, and M. Damas, "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*, Kyoto, Japan, July 2018, GECCO'2018, pp. 1862–1869, ACM.

[23] J.J. Escobar, J. Ortega, J. González, and M. Damas, "Assessing parallel heterogeneous computer architectures for multiobjective feature selection on eeg classification," in *Proceedings of the 4th International Conference on Bioinformatics and Biomedical Engineering*, Granada, Spain, April 2016, IWBBIO'2016, pp. 277–289, Springer.

[24] J.J. Escobar, J. Ortega, J. González, and M. Damas, "Improving memory accesses for heterogeneous parallel multi-objective feature selection on eeg classification," in *Proceedings of the 4th International Workshop on Parallelism in Bioinformatics*, Grenoble, France, August 2016, PBIO'2016, pp. 372–383, Springer.

[25] J.J. Escobar, J. Ortega, J. González, M. Damas, and A.F. Díaz, "Parallel high-dimensional multi-objective feature selection for eeg classification with dynamic workload balancing on cpu-gpu," *Cluster Computing*, vol. 20, no. 3, pp. 1881–1897, 2017.

[26] J. Asensio-Cubero, J.Q. Gan, and R. Palaniappan, "Multiresolution analysis over simple graphs for brain computer interfaces," *Journal of Neural Engineering*, vol. 10, no. 4, pp. 21–26, 2013.

[27] E. Zitzler and L. Thiele, "Multiobjective optimization using evolutionary algorithms - a comparative case study," in *Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, Amsterdam, The Netherlands, September 1998, PPSN V, pp. 292–301, Springer.

[28] J.J. Escobar, "Hpmoon," https://github.com/efficomp/Hpmoon, Accessed: 2022-05-20.

[29] Advanced Configuration and Power Interface (ACPI), "Acpi specification," https://uefi.org/specs/ACPI/6.4/, Accessed: 2022-05-11.