Facultad de Ciencias

PCBE: Desarrollo de un algoritmo de planificación basado en energía

PCBE: a novel energy-based configurable HPC scheduler

Trabajo de Fin de Máster
para acceder al

MÁSTER EN INGENIERÍA INFORMÁTICA

Autor: Luis Cruz Varona

Director\es: José Luis Bosque Orero

Septiembre - 2023

Índice general

Resumen

High Performance Computing es una rama de la computación que hace uso de clústers de gran tamaño, para dar servicio a aplicaciones científicas muy complejas. Para conseguir un rendimiento tan alto, estos clústers suelen estar compuestos por nodos heterogéneos, con procesadores con capacidades muy diferentes. Es por esto que, los sistemas HPC, especialmente los más modernos, tienen un consumo energético muy elevado, llegando a alcanzar los 20MW de potencia máxima. Sin embargo, este gran consumo tiene un impacto económico y ambiental muy importante. Es por esto, por lo que la investigación de herramientas y componentes que hagan mejor utilización de los recursos de un clúster es más importante que nunca. Una de las posibles áreas donde se pueden encontrar ganancias de consumo y eficiencia energética es en los *resource managers* o *workload managers* de un clúster, los encargados de planificar la cola de los trabajos que van llegando al sistema.

Es en este contexto en el que se ha desarrollado PCBE. El Planificador Configurable Basado en Energía/Eficiencia energética (PCBE) es un algoritmo de planificación novedoso, capaz de optimizar el consumo o eficiencia energética de un clúster HPC. Para ello, es capaz de realizar la planificación en base a una estimación del consumo (o eficiencia) de los trabajos que le llegan en los diferentes nodos del clúster, en vez del tiempo que van a tardar en ejecutarse, como hacen las técnicas de planificación tradicionales. En este proyecto se describe el proceso seguido para diseñar e implementar PCBE, usando como plataforma IRMASim, un simulador de sistemas HPC creado por la Universidad de Cantabria. Asimismo, durante el proyecto se han realizado experimentos, probando las diferentes configuraciones de PCBE y comparándolo con algoritmos de planificación tradicionales, como *First Come First Serve* y *Shortest Job First*. Como resultado del mecanismo de estimación de la energía, PCBE es capaz de mejorar alrededor de un 10 % el consumo energético de un clúster, respecto a FCFS y SJF, mejorando también el resto de aspectos del clúster medidos: eficiencia energética y throughtput.

Abstract

High Performance Computing is a discipline of computer science that makes use of clusters with a high number of nodes, to give service to complex scientific applications and simulations. In order to achieve such high performance, these types of clusters are usually composed of nodes with very different capabilities. Thanks to this heterogeneous architecture, their power consumption can be quite high, reaching in some instances up to 20MW of peak power. This, however, has a significant economic and environmental impact. In order to reduce this consumption, research regarding new tools and components that make better use of the computational power available is necessary. One of these areas of research that may help improve energy consumption and efficiency in an HPC cluster is the development of new workload/resource managers, the pieces of software in charge of scheduling the incoming jobs to the nodes in a cluster.

In this project, a novel energy based scheduling algorithm is proposed to minimize energy consumption in clusters. PCBE (configurable energy/efficiency-based scheduler) is a scheduling algorithm capable of estimating the energy consumption and efficiency of each of the jobs that arrive to a cluster, assigning them to the relevant nodes, and thus improving overall consumption when compared to more traditional scheduling algorithms (that take into account only the jobs' execution time). In this document, the process used to design and implement PCBE is described, as well as the framework used to test it, IRMASim, an HPC cluster simulator developed at Universidad de Cantabria. On top of this, the experiments used to compare PCBE against more traditional scheduling algorithms, such as First Come First Serve and Shortest Job First, are presented. As a result of the novel energy estimation method used by PCBE, the algorithm is able of achieving 10 % lower energy consumption, when compared with FCFS and SJF, while also improving other metrics such as cluster throughput and energy efficiency.

Capítulo 1

Introducción

1.1. High Performance Computing

High Performance Computing (HPC) es una rama de la computación caracterizada por utilizar clústers con gran número de recursos, que atienden un gran número de aplicaciones con un alto grado de paralelismo. Estos clústers suelen tener una gran cantidad de nodos con funcionalidades muy dispares, usando en el mismo sistema HPC procesadores de alto rendimiento y de bajo consumo, algunos de ellos con aceleradores para aplicaciones específicas como GPUs o FPGAs, haciendo que este tipo de arquitecturas sean muy heterogéneas.[3, 26]

Estos sistemas HPC se suelen usar para dar servicio a una gran gama de aplicaciones científicas con requerimientos muy variados, como la simulación del doblado de proteínas [23], el estudio y predicción de fenómenos meteorológicos [33], o aplicaciones de búsqueda por contenido en vídeo [4]. Para poder garantizar que todas las aplicaciones se puedan atender en un tiempo razonable, los clúster hacen uso de un componente software conocido como *resource/workload manager*, que se encarga de planificar los trabajos que le van llegando.

Sin embargo, el problema de asignar los trabajos que llegan al clúster de manera que no se desperdicie potencia de cómputo es *NP-Hard* [19, 5]. Es por esto que los algoritmos de planificación usados por los *workload managers* generalmente se contentan con dar una solución lo suficientemente buena (que puede no ser la óptima); sacrificando, de esta manera, precisión en favor de un menor tiempo de ejecución y mayor simplicidad.

Por otro lado, los clústers HPC tienen un consumo energético muy elevado, debido a su gran cantidad de procesadores. Algunos de ellos pueden llegar a tener una potencia máxima de entre 15 y 20 MW [22]. Este consumo energético, puede tener un impacto no solo económico (debido al coste de la electricidad necesaria para mantener dicho clúster funcionando), sino también medioambiental, dependiendo del origen de la electricidad para la zona en la que se encuentre el clúster[22, 7].

Para reducir el consumo energético de estos clústers existen una gran cantidad de técnicas, como el uso de DVFS en los cores de los nodos[13], o el uso de procesadores de bajo consumo [7]. Sin embargo, otro de los componentes de un clúster que puede afectar a su consumo es el *workload manager*. Mediante el uso de técnicas de planificación más avanzadas (como el uso de redes neuronales [17]) que tengan en cuenta la energía, estos gestores de recursos pueden realizar planificaciones más óptimas, obteniendo así reducciones significativas del consumo en un clúster [7].

1.2. Objetivos

El objetivo principal de este proyecto es desarrollar un algoritmo heurístico que permita a un clúster HPC realizar la planificación de los trabajos que le llegan en función de su posible consumo energético. Además, se debe determinar qué criterios de selección de trabajos y nodos proporcionan a dicho algoritmo un mejor rendimiento. Este algoritmo debe tener además, un coste computacional razonable (no mayor a $O(n^2)$), que permita su uso en situaciones reales.

Asimismo, es necesario probar el rendimiento de dicho algoritmo, para poder caracterizar su comportamiento y poder determinar si realmente se obtienen beneficios por usarlo. Para ello es necesario realizar pruebas en entornos diferentes:

- Pruebas sencillas en las que se pueda determinar la combinación de parámetros óptima.
- Pruebas realistas que permitan comparar el rendimiento del algoritmo contra algunos de los planificadores más usados.

Para conseguir los objetivos propuestos, es necesario construir el algoritmo sobre una plataforma que permita la experimentación con varios planificadores distintos y cargas de trabajo a medida. Para esto, será necesario usar IRMASim, una herramienta que permite simular clústers HPC de varios tamaños, para cargas de trabajo arbitrarias. IRMASim, además, permite extraer datos relativos a la planificación y métricas pertinentes al clúster, como duración de la planificación, consumo energético, ...

1.3. Plan de trabajo

Para el desarrollo de este proyecto, se han planificado varias fases, siendo la primera un estudio previo. Este estudio debe comprender las diferentes técnicas que existen para desarrollar algoritmos, así como los diferentes algoritmos de planificación existentes, haciendo hincapié en aquellos que tienen en consideración (o realizan la planificación respecto a) la energía consumida por cada trabajo/el clúster entero. Además, es necesario estudiar la arquitectura de la plataforma sobre la que se va a desarrollar el planificador (IRMASim), teniendo en cuenta los requisitos detallados en la sección anterior.

Tras realizar el estudio de las posibles alternativas, se comenzará la siguiente fase: el desarrollo del planificador; escogiendo para ello la técnica algorítmica que se determine más adecuada. Una vez elegida la técnica, se hará un diseño inicial del algoritmo, en el que se basará la implementación posterior. Esta implementación deberá ser depurada de manera exhaustiva para asegurar que no haya comportamientos inesperados en el planificador.

Por último, se realizarán experimentos, diseñados para probar las diferentes áreas del algoritmo en diferentes situaciones, permitiendo analizar en profundidad su comportamiento. Asimismo, se deberá probar el planificador en situaciones realistas, usando clústers de gran tamaño y colas de trabajos complejas, comparándolo con algoritmos de planificación tradicionales. Tras realizar estas pruebas, será necesario un análisis de los resultados, que determine si el uso del planificador implementado tiene beneficios, especialmente en lo que a consumo energético del clúster se refiere.

1.4. Estructura del documento

La estructura del resto del documento es la descrita a continuación:

- En el capítulo 2 se explican los conceptos usados durante el resto del trabajo, como algoritmos de planificación y técnicas de diseño de algoritmos. Además, se hace una introducción de la plataforma usada para desarrollar PCBE, el simulador IRMASim[19]. Por último, se mencionan algunos trabajos relacionados con éste, como pueden ser algoritmos de planificación alternativos que tienen en cuenta de alguna manera la energía del clúster.

- En el capítulo 3 se detallan las fases de diseño e implementación de PCBE, entrando en detalle en el mecanismo de estimación de la energía usado y realizando un estudio teórico de su complejidad temporal, para asegurarse de que no excede $O(n^2)$.
- En el capítulo 4, por otro lado, se describen los experimentos definidos para probar el algoritmo, así como las herramientas implementadas para realizarlos y la metodología seguida en cada uno de ellos.
- En el capítulo 5 se muestran y analizan los resultados obtenidos de realizar las pruebas definidas en el capítulo 4. Para ello se entra en detalle en el comportamiento de PCBE, estudiando cómo realiza las planificaciones y cómo se compara respecto a otros algoritmos.
- Por último, en el capítulo 6 se explican las conclusiones extraídas de los experimentos, proponiendo, además, posibles trabajos futuros a desarrollar relacionados con PCBE, permitiendo ampliar su funcionalidad.

Todo el código desarrollado para este proyecto, incluyendo las herramientas usadas para los experimentos (además de los resultados obtenidos de los mismos), se puede encontrar en los repositorios: Implementación de PCBE, Experimentos de PCBE.

Capítulo 2

Background

Antes de explicar en detalle el desarrollo del planificador y los resultados obtenidos de los experimentos realizados, es importante introducir algunos de los conceptos que se utilizarán durante el resto del trabajo.

En concreto, se hablará sobre planificación de trabajos (y los posibles algoritmos que se pueden usar para ello), y consumo energético, y se hará una breve introducción al framework usado para el desarrollo, IRMASim. Por último, algunos trabajos previos relacionados con este proyecto.

2.1. Planificación de trabajos

En clusters HPC, la gestión de los trabajos entrantes y cómo se asignan sus recursos es muy importante. Esta gestión es típicamente realizada por programas conocidos como "job schedulers" (también llamados "workload managers" o "resource managers", entre los que se encuentran "SLURM, MOAB/-Torque, PBS y Cobalt"[14][15].

Para hacer una utilización correcta de los equipos disponibles, estos programas deben realizar una planificación óptima de los trabajos a ejecutar constantemente. Esta planificación se encarga de asignar dichos trabajos dentro de los nodos disponibles en el clúster de manera que todos acaben siendo atendidos.

Esta asignación se suele realizar respecto a un objetivo que se quiere minimizar o maximizar, como por ejemplo: el rendimiento del clúster (o cantidad de trabajos que puede ejecutar en un tiempo concreto) , el tiempo de ejecución de los trabajos individuales (makespan) o, como es en el caso de este trabajo, el consumo energético de los nodos usados [35, 36].

Sin embargo, esta planificación de los trabajos no es trivial; para encontrar la más óptima, es necesario buscar entre todas las posibles soluciones la que mejor se adapte a una serie de condiciones preestablecidas. El tamaño de este espacio de búsqueda viene determinado por la cantidad de aspectos que se quieran considerar, como por ejemplo: tiempo de llegada del trabajo, número de procesadores de un nodo del clúster...

Dependiendo de los aspectos a considerar al realizar la planificación, ésta puede ser más o menos complicada. Sin embargo, la complejidad temporal teórica para este tipo de problemas (de búsqueda combinatoria u optimización) es muy elevada, independientemente del tamaño del espacio de búsqueda. Es por esto que la planificación de trabajos se suele denominar como un problema *NP-Hard* [5]. Para solucionar este problema existen varios algoritmos de planificación.

2.1.1. Algoritmos de planificación

La mayoría de los planificadores usan alguno de los siguientes algoritmos para realizar la planificación, debido a varias razones, como que no son muy difíciles de implementar y son relativamente rápidos, produciendo resultados suficientemente buenos. Estos algoritmos son:

FCFS

Un algoritmo muy simple pero bastante usado [14], es el conocido como *First-Come-First-Serve*. Este algoritmo atiende los trabajos según su orden de llegada al clúster, de manera que los que más tiempo llevan en la cola son ejecutados. Este algoritmo es muy sencillo y relativamente "justo", ya que todos los trabajos serán atendidos en orden y se les dedicarán los recursos necesarios.

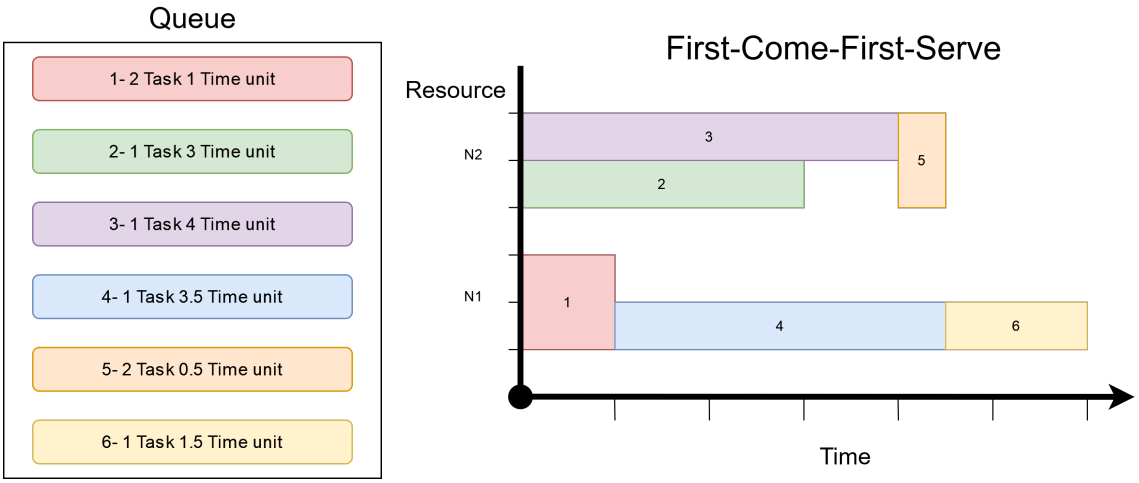

Figura 2.1: Ejemplo de una aplicación del algorimto FCFS sin backfill para una cola de trabajos (figura de la derecha), en un clúster de 2 nodos con 2 cores cada nodo. El número de cada trabajo indica el orden de llegada de éstos a la cola.

SJF

Otro algoritmo bastante popular es *Shortest-Job-First*, en el que los trabajos son asignados a un nodo del clúster en función del tiempo que van a tardar en completarse. Este algoritmo da prioridad a los trabajos con menor tiempo de ejecución, y por lo tanto, puede generar un problema de *starvation* en el que los trabajos más largos nunca lleguen a ejecutarse, ya que siempre habrá otro más corto que será planificado primero. Asimismo, el tiempo de ejecución de un trabajo no siempre se puede saber a priori. Para solucionar esto existen algunas técnicas, como la que emplea SLURM, que pide al usuario una estimación del tiempo de ejecución del trabajo.[37, 27]

Backfill

Una alternativa a FCFS un poco más compleja, pero también muy usada es backfill[15]. Este algoritmo mantiene ciertas características de FCFS, como atender los trabajos según su orden de llegada. Sin embargo, cuando backfill encuentra que no puede planificar el primer trabajo de la cola, pasa al siguiente, y así hasta que encuentre uno que "quepa" en el clúster, al contrario que FCFS, que esperará a que haya espacio para el primer trabajo de la cola antes de seguir planificando.[9]

Existen dos tipos de algoritmo de backfill: EASY y conservador (CONS). La variante EASY realiza una planificación que garantiza que el primer trabajo de la cola no se vea retrasado, introduciendo (si puede) entre los huecos trabajos cortos. Por otro lado, la versión CONS garantiza que ningún trabajo de la cola se vea retrasado, aunque eso signifique no introducir entre los huecos esos trabajos cortos.[9]

Figura 2.2: Ejemplo de una aplicación del algorirmo SJF para la cola de trabajos de la figura anterior, en un clúster con 2 nodos y 2 cores por nodo.

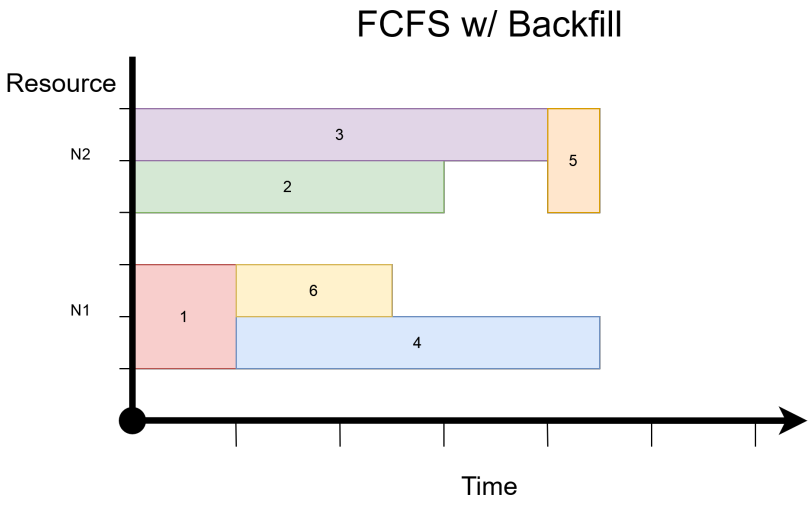

Figura 2.3: Ejemplo de una aplicación del algoritmo FCFS con backfill para la cola de trabajos usada en los 2 diagramas anteriores, en un clúster con 2 nodos y 2 cores por nodo.

Ambas versiones tienen sus ventajas: EASY maximiza la utilización de los recursos para una cola dada, pero a cambio puede retrasar la ejecución de ciertos trabajos de la cola; por otro lado, CONS hace una utilización menos eficiente de los recursos, pero a cambio garantiza que ningún trabajo sea discriminado negativamente.[9]

2.1.2. Técnicas alternativas

A pesar de que éstos son los algoritmos más usados, a veces puede ser necesario desarrollar un algoritmo más complejo que tenga en cuenta otros factores (a parte de tiempo de llegada y tiempo de ejecución), como la cantidad de memoria utilizada por cada trabajo[18]. Para este fin, existen ciertas técnicas algorítmicas que pueden ser usadas para desarrollar un planificador a medida.

Búsqueda en profundidad (DFS)

Los algoritmos de búsqueda en profundidad buscan todas las combinaciones posibles dentro del espacio de búsqueda, eligiendo siempre la mejor opción. Estos algoritmos producen siempre la mejor solución

posible, a costa de tener una complejidad temporal muy elevada. Existen varias técnicas para intentar reducir el tiempo de la exploración del espacio de búsqueda, como el podado de soluciones peores a la encontrada hasta el momento o el descarte de soluciones idénticas (para espacios de búsqueda donde el orden de exploración de los puntos del espacio no influye en la solución). Además, existe una variante a este tipo de algoritmos, la búsqueda a lo ancho, o *breadth-first-search* (BFS), que modifica el orden de búsqueda de las soluciones. [31]

Algoritmos voraces

Por otro lado, están los algoritmos voraces. Estos algoritmos escogen en cada iteración de la búsqueda la solución aparentemente más óptima, sin valorar cómo puede afectar esto a futuras invocaciones. Su coste de cómputo es bastante inferior al de la búsqueda en profundidad, pero a cambio pueden producir soluciones subóptimas. [32]

Algoritmos genéticos

Los algoritmos genéticos son algoritmos generacionales que suelen emplear una población de agentes (generados aleatoriamente) para producir las soluciones a un problema. Para cada generación, el algoritmo escoge a los agentes con las mejores soluciones (de acuerdo a una función de evaluación o *fitness*), y los combina y modifica ligeramente, utilizándolos como punto de partida para generar la siguiente generación o iteración del algoritmo.

Los resultados que producen este tipo de algoritmos son bastante dependientes del número de generaciones empleadas; para un número adecuado de iteraciones, estos algoritmos pueden producir soluciones suficientemente buenas (mejores que las de los voraces), manteniendo un coste temporal inferior al de la búsqueda en profundidad.[30]

Algoritmos basados en partículas (PSO, ACO, etc . . .)

Los algoritmos basados en partículas son similares a los géneticos. Éstos también emplean una población de agentes (o partículas), pero ésta no se modifica entre generaciones, se deja que las partículas exploren el espacio de búsqueda, y se espera a que, mediante una función de evaluación, converjan en una solución lo suficientemente buena.[12][10]

Al igual que los algoritmos genéticos, los basados en partículas pueden producir resultados mejores que los algoritmos voraces en un tiempo no mucho mayor; aunque esto es dependiente del número de iteraciones realizadas.

Reinforcement learning (RL)

Reinforcement Learning usa una red neuronal para generar soluciones a un problema. Esta red es entrenada mediante "recompensas" y "castigos" de acuerdo a las acciones que realiza sobre el espacio de búsqueda y de si éstas acercan a la red a una solución óptima o no. De esta manera, la red aprende a extraer patrones del espacio de búsqueda y encuentra un método generalizado para resolver los problemas que usan el mismo espacio de búsqueda.

Sin embargo, el proceso de entrenamiento es largo, complicado y requiere de una colección inicial de datos (que, dependiendo del problema, puede ser muy grande). Por otro lado, una vez entrenado el modelo, su tiempo de ejecución es relativamente corto, pudiendo producir resultados mejores que los de los algoritmos voraces y genéticos. [17][15]

2.2. Métricas de consumo y eficiencia energética

Dado que el proyecto tiene como objetivo desarrollar un planificador que intente minimizar el consumo energético de un clúster, es necesario introducir las métricas de energía y eficiencia que se van a usar para los cálculos y estimaciones descritas en los capítulos posteriores a este.

El consumo energético de cualquier equipo informático se puede describir como el trabajo realizado por el sistema (o potencia consumida por dicho sistema) multiplicado por el tiempo durante el cual se ha estado realizando dicho trabajo. Las ecuaciones 2.1 y 2.2 muestran una versión simplificada de este cálculo, en el que se asume que la potencia consumida por el sistema es constante a lo largo del tiempo. En estas ecuaciones (2.1 2.2), E denota el consumo energético de un trabajo j que se ejecuta en un nodo n, donde T_j es el tiempo de ejecución del trabajo j en el nodo n, y P_n es la potencia utilizada por el nodo n para ejecutar el trabajo j.

$$E = T \cdot P \tag{2.1}$$

$$E_{jn} = T_j \cdot P_n \tag{2.2}$$

Generalmente, esta simplificación no es adecuada para modelar de manera precisa el consumo de un sistema informático, ya que éste fluctúa en función de su utilización. Existen técnicas, como DVFS, que pueden variar la potencia de un CPU modificando parámetros como el voltaje y la frecuencia del mismo[13]. Sin embargo, el desarrollo del planificador se ha desarrollado en IRMASim[19], que modela los CPUs de los nodos como componentes con una frecuencia fija.

Asimismo, IRMASim[19] modela el consumo energético de un nodo como el consumo de sus cores, por lo que, el consumo de potencia de un nodo se puede modelar como las ecuaciones 2.3 y 2.4; donde la potencia de un nodo n viene denotada por P_n, Ps_n es su potencia estática y Pd_n es su potencia dinámica, siendo Ta_j el número de tareas del trabajo j. Por otro lado, la potencia dinámica de un core del nodo, Pd_n, se calcula usando la capacitancia del nodo, C_n, el voltaje al que corre el core, V_n, y la frecuencia del nodo, F_n.

$$P_n = Ps_n + (Pd_n \cdot Ta_j) \tag{2.3}$$

$$Pd_n = \frac{1}{2} C_n \cdot V_n^2 \cdot F_n \tag{2.4}$$

En esta ecuación (2.3) se considera que la potencia de un nodo viene dada por la potencia estática, que consume por estar encendido, más la potencia dinámica, la potencia que consume al realizar un trabajo, de todos los cores que actualmente están ejecutando una tarea. Además, la ecuación 2.4 describe una manera de calcular la potencia dinámica de cada core del nodo Pd_n. Sin embargo, debido a que IRMASim[19] no modela cambios en la frecuencia ni voltaje de los CPUs de un clúster, esta ecuación no se usará, asumiendo que tanto la potencia estática como dinámica de un core del clúster son constantes.

Por otro lado, con optimizar y medir el consumo energético de un clúster HPC no es suficiente; puede darse que, aunque se haya consumido poca energía, se haya hecho en más tiempo del necesario, ralentizando el clúster en favor de un menor consumo. Para poder evaluar este aspecto, es necesario planificar y evaluar los resultados obtenidos de dicha planificación teniendo en cuenta la eficiencia energética del sistema. Como métrica elegida para medir la eficiencia energética, se ha escogido el *Energy Delay Product* [6]. Esta métrica (representada en las ecuaciones 2.5 y 2.6) calcula la eficiencia como el consumo de energía realizado a lo largo de un periodo de tiempo. El Energy Delay Product se puede representar como E, la energía consumida, por T, el periodo de tiempo durante el cual se ha realizado el consumo, siendo P la potencia utilizada.

$$EDP = E \cdot T \tag{2.5}$$

$$EDP = P \cdot T^2 \tag{2.6}$$

2.3. IRMaSim

Además de introducir conceptos de planificación, o *scheduling*, algoritmos y energía, es necesario explicar de manera breve cuál es el framework que se ha utilizado para desarrollar el proyecto y cómo funciona de manera superficial.

El framework utilizado para desarrollar el planificador basado en energía ha sido IRMASim. IRMASim es un proyecto desarrollado por el grupo de Arquitectura y Tecnologías de Computadores de la Universidad de Cantabria. Este programa, escrito en python, es capaz de simular el comportamiento de un clúster HPC al que llegan trabajos de forma asíncrona. [19]

El simulador calcula el tiempo de ejecución de cada trabajo dentro del clúster, así como la energía consumida y el slowdown en la cola de trabajos (como resultado de la competición entre los trabajos por ciertos recursos, como el ancho de banda de la memoria principal, que genera contención en los accesos a memoria). Para ello hace uso de ecuaciones que modelan los tiempos de ejecución de trabajos en el clúster, teniendo en cuenta factores como saturación del ancho de banda de memoria del procesador[19]. Cabe destacar, sin embargo, que IRMASim no simula la red que conecta los diferentes nodos, por lo que para este proyecto se han considerado cargas de trabajo que se ejecutan en un sólo nodo.

Gracias a la arquitectura de IRMASim, implementar y probar algoritmos de scheduling es sencillo, y es por esto que se ha elegido implementar el algoritmo en esta plataforma. A continuación se describe de manera sencilla la arquitectura de IRMASim.

2.3.1. Arquitectura interna de IRMASim

IRMASim implementa la simulación de clusters HPC usando la arquitectura representada en el diagrama de clases 2.4. Cabe destacar que este diagrama ha sido simplificado para ilustrar los componentes más importantes de IRMASim.

El componente más importante es la clase *Simulator*, que realiza la simulación, calculando cada uno de los eventos del clúster (llegada de trabajos, comienzo y finalización de un trabajo, ...) y el delta de tiempo entre éstos. Asimismo, Simulator es el responsable de invocar al *Workload Manager*, WM, el componente encargado de realizar la planificación de los trabajos. Más adelante se describe en más detalle como funciona el WM.

Para modelar tanto el clúster como los trabajos que le llegan, el simulador se apoya en una lista de *resources* que contiene todos los nodos a simular y una *job_queue* que contiene todos los trabajos que van a llegar al clúster durante la simulación. Un nodo está compuesto por sus cores, además de su frecuencia. Cada core tiene su propia potencia dinámica y estática, así como la cantidad de operaciones por segundo que puede realizar. Por otro lado, un trabajo o *job* está representado por su identificador (id), el ancho de banda de la memoria que va a usar, el número total de operaciones que va a realizar, el momento en el que llega, empieza y termina de ejecutarse, y la estimación del tiempo de ejecución que provee el usuario. Asimismo, un trabajo está compuesto por varias subtareas, que se ejecutarán cada una en un core del nodo al que se asigne. De todos estos parámetros, hay varios que son internos al simulador y que, por lo tanto, el algoritmo de scheduling no podrá ver (para ser más fiel a una situación real), entre estos están el momento de llegada de un trabajo o la cantidad de ancho de banda e instrucciones que va a consumir dicho trabajo.

Para configurar una simulación, IRMASim hace uso de varios ficheros de configuración JSON, que definen el entorno de la simulación (qué tipo de scheduling a utilizar, semilla para los valores aleatorios de la simulación, ...), la arquitectura del clúster a simular (cuántos nodos posee, de qué tipo son, cuál es su frecuencia y potencia, el número de cores por nodo, etc...) y los trabajos a ejecutar dentro del clúster (definiendo cada unos de los aspectos de dichos trabajos).

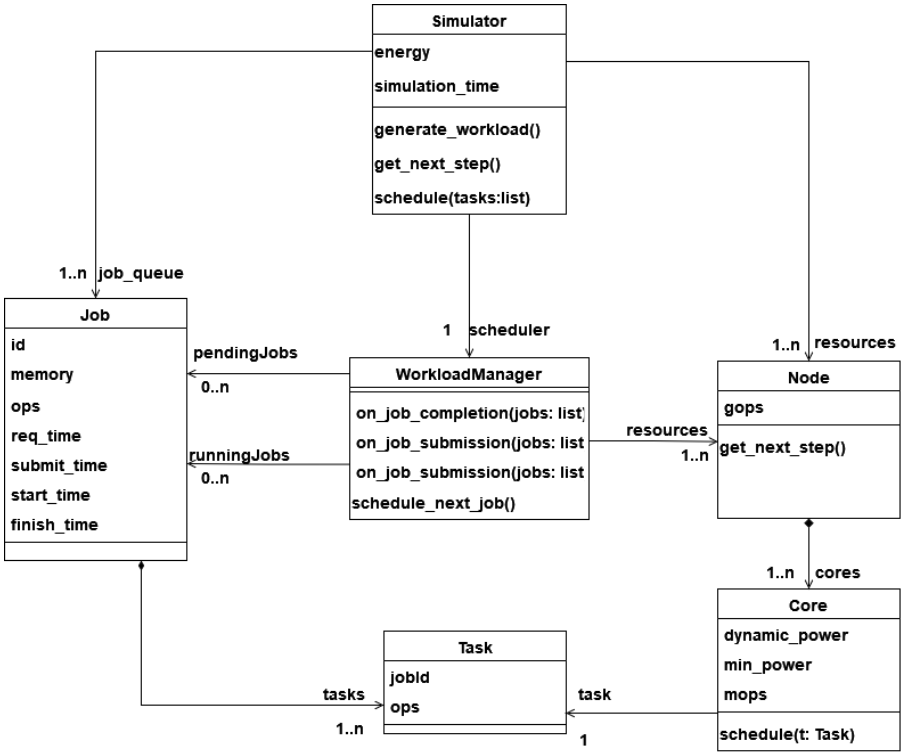

Figura 2.4: Diagrama de clases simplificado de IRMASim

El Workload Manager

Aunque en el diagrama 2.4 se ha representado el WM como una clase más, interna al simulador, éste es más bien una superclase que se usa como base para implementar los diferentes tipos de algoritmos de planificación que soporta IRMASim. Las implementaciones de los Workload Manager disponibles en IRMASim son muy variadas, desde algoritmos puramente heurísticos, hasta sistemas de redes neuronales basados en reinforcement learning[17].

Este sistema permite prototipar e implementar en el simulador nuevos algoritmos de planificación de manera muy sencilla, siempre y cuando éstos se ciñan a las operaciones y datos disponibles en la superclase. Es debido a esta simplicidad y desacople del WM con el resto del simulador por lo que se ha elegido desarrollar el algoritmo de scheduling en esta plataforma.

2.4. Trabajos relacionados

Existen varios trabajos anteriores que se centran en proponer algoritmos de planificación para clusters HPC. Entre ellos se encuentran varios que hacen uso de las diversas técnicas descritas en este capítulo, algunos de los cuales tienen en cuenta el consumo energético del clúster como factor de la planificación.

La mayoría de estos trabajos utilizan el consumo energético como un factor secundario, intentando optimizar el tiempo de ejecución dentro de un presupuesto energético. Por ejemplo, Raca et al.[29] proponen un planificador que optimiza la distribución de tareas en un clúster heterogéneo, minimizando

el tiempo de ejecución para un límite superior de energía. Por otro lado, Benoit et al.[2] proponen un algoritmo capaz de planificar un conjunto de tareas que empiezan a la vez, teniendo en cuenta el consumo energético de éstas.

En EATSDCD[1], Barzegar et al. proponen un algoritmo de planificación que minimiza el tiempo de ejecución para tareas con dependencias temporales, una vez decidido el orden en el que atender las tareas, el planificador intenta minimizar el consumo energético de la planificación, apoyándose para ello en herramientas, como DVFS. Asimismo, Nejad et al. [25] proponen un algoritmo que tiene en cuenta las capacidades de los nodos del clúster, así como la capacidad de enfriamiento de los mismos, para asignar trabajos de manera eficiente.

Además de estos planificadores basados en técnicas heurísticas, existen otros que utilizan conceptos más complejos. Como [11], de Mejri et al., que describe un algoritmo voraz capaz de planificar trabajos (y decidir qué equipos apagar durante periodos de inactividad), teniendo en cuenta el consumo energético del clúster y usando parámetros que han sido optimizados mediante *Particle Swarm Optimization* (PSO). Cocaña et al.[8], por otro lado, describen un sistema basado en algoritmos géneticos, capaz de encontrar la mejor manera de colocar trabajos planificados en el hardware disponible, optimizando varios parámetros, entre ellos el consumo energético.

Asimismo, también están tomando popularidad las técnicas basadas en *Deep Reinforcement Learning*, que hacen uso de redes neuronales pre-entrenadas para realizar la planificación de los trabajos. Fan et al.[15] describen un planificador para clústers HPC basado en RL capaz de aprender a optimizar un objetivo concreto, pudiendo hacer ajustes de manera dinámica durante su ejecución. Por otro lado, Fomperosa et al. [17] introducen a IRMASim[19] un planificador basado en machine learning que es capaz de determinar la mejor combinación de trabajo-recurso para realizar un scheduling óptimo.

Por último, éstas técnicas basadas en machine learning también están siendo aplicadas a planificadores *energy-aware*. Por ejemplo, Qin et al.[28] presentan un sistema basado en RL que es capaz de planificar trabajos intentando minimizar tanto el tiempo total de ejecución de la cola, como la energía consumida por el clúster.

Como se puede ver, la mayoría de estos trabajos tienen en cuenta el consumo energético como una variable secundaria a optimizar en la planificación, o como una restricción del sistema que hay que satisfacer. Es en este contexto, en el que presentamos este nuevo planificador, capaz no solo de optimizar el consumo energético del clúster, sino también su eficiencia energética.

Capítulo 3

Diseño e implementación de PCBE

En un clúster HPC, varios componentes (como los nodos, o la interconexión entre éstos) trabajan de manera conjunta para ejecutar las cargas de trabajo lanzadas al mismo. Uno de los componentes más importantes dentro del clúster es el gestor de colas. El gestor de colas es una herramienta software que hace uso de un algoritmo para realizar la planificación de una cola de trabajos. Este algoritmo define en qué orden atienden los trabajos que van llegando y en qué nodo se ejecutarán. Diferentes algoritmos tienen impacto en diferentes características del clúster, como el tiempo que tarda en ejecutar trabajos o la energía que consume. Sin embargo, como se ha comentado en el capítulo anterior, el problema de decidir qué trabajos y nodos priorizar es NP-Hard [5]. Es por esto que, para desarrollar un nuevo algoritmo de planificación, es necesaria una fase de planificación y diseño minucioso previa.

En esta sección se ilustra el proceso seguido para diseñar e implementar el Planificador Configurable Basado en Energía/Eficiencia Energética (PCBE). PCBE es capaz de realizar planificaciones basándose en una estimación del consumo energético (o eficiencia energética, usando el *Energy Delay Product*, EDP) de los trabajos de la cola. Ésto es diferente de la gran mayoría de algoritmos de planificación (como FCFS o SJF), que se sirven de una estimación del tiempo de ejecución del trabajo para realizar la planificación.

Para implementar y probar una versión funcional de PCBE, se ha usado el simulador IRMASim, que ha sido extendido para soportar y exportar ciertas métricas de energía, y sobre el que se ha desarrollado el algoritmo diseñado, mediante la creación de un nuevo Workload Manager (WM).

3.1. Diseño

Como paso previo al desarrollo del workload manager, que hiciera la planificación basada en energía, fue necesario diseñarlo; eligiendo para ello el tipo de algoritmo (de los descritos en el capítulo 2) más adecuado. Este algoritmo debía ser relativamente "rápido"(que su complejidad temporal no fuera superior a $O(n^2)$) y ser determinista, es decir, producir resultados repetibles (para facilitar la depuración y experimentación), además de cumplir los requisitos descritos a continuación.

3.1.1. Requisitos del planificador

Para que sea efectivo, un algoritmo de planificación debe ser capaz de asignar los trabajos que van llegando al clúster de manera eficaz, de acuerdo a una serie de parámetros. En este caso, cuando nos referimos a un trabajo, hablamos de un programa que tiene una o varias subtareas (o subprocesos) a ejecutar. Este trabajo tiene, además, una serie de necesidades computacionales y de memoria (el número de instrucciones que va a ejecutar, la cantidad de RAM que va a ocupar, el número de accesos a caché que va a realizar, etc. . . .) que pueden impactar en qué nodo es planificado y cuánto tendrá que competir por los recursos del nodo con el resto de trabajos ejecutándose en él. Asimismo, cada trabajo llega en un momento arbitrario de la planificación y su tiempo de ejecución (y en nuestro caso consumo y eficiencia

energética) no puede ser sabido *a priori*. Sin embargo, ciertos gestores de recursos, como SLURM [34], incluyen un campo para cada trabajo de la cola, en el que el usuario que ha lanzado el trabajo puede incluir una estimación de su duración. PCBE debe hacer también uso de una técnica similar, que permita al usuario definir una estimación del consumo energético de su trabajo.

Para poder realizar una planificación eficaz de los recursos de un clúster, el algoritmo a diseñar debe ser capaz de realizar una estimación (lo suficientemente buena) del consumo energético o eficiencia de cada uno de los trabajos que llegan al clúster en cada uno de los nodos. Esta estimación será la usada para decidir en qué orden se atienden los trabajos y a qué nodo se envían, intentando minimizar así el consumo energético (o el EDP, si se quiere optimizar la eficiencia) del clúster. Además, el algoritmo debe ser capaz de determinar si el nodo que se está considerando para un trabajo en concreto tiene los suficientes cores libres como para poder alojar todas sus subtareas.

Asimismo, PCBE debe tener un tiempo de ejecución lo suficientemente corto como para no impactar negativamente el rendimiento del clúster que está planificando. Por último, no hay que olvidar que este algoritmo no debe discriminar trabajos según su consumo energético, de tal manera que haya ciertos trabajos que nunca sean atendidos. Ésto podría resultar en una mejoría del consumo de energía del clúster, sacrificando, sin embargo, su rendimiento para las tareas que puedan ser más intensivas.

A fin de satisfacer todos los aspectos aquí descritos, se pasó a valorar cada una de las técnicas disponibles para diseñar un algoritmo de tal carácter. En la siguiente sección se explica cuál de esas técnicas se ha usado finalmente y por qué, descartando otras posibles alternativas que son comúnmente usadas.

3.1.2. Elección del algoritmo

Tras estudiar detenidamente las alternativas explicadas en el capítulo 2, se eligió diseñar un algoritmo voraz, ya que es el que mejor se ajustaba a los requerimientos anteriormente descritos.

Primero, éste debía ser rápido a la hora de decidir dónde colocar cada trabajo, ya que, de lo contrario, el tiempo de espera de los trabajos podría verse afectado negativamente. Este factor ya descarta a los algoritmos DFS, que sufrirían para colas de trabajos largas, en clústers grandes (un clúster de 10 nodos con una lista de 10 trabajos pendientes implicaría buscar en un espacio de 10^{10} posibles soluciones, asumiendo un algoritmo DFS sin optimizaciones). De la misma manera, los algoritmos evolutivos necesitarían un número adecuado de iteraciones hasta empezar a producir resultados válidos, que varía según el tamaño del espacio de búsqueda.

Por otro lado, las soluciones que produzca el algoritmo no tienen que ser necesariamente las mejores posibles para un instante dado de tiempo, ya que un clúster HPC es un sistema dinámico en el que el espacio de búsqueda cambia constantemente, con lo que la solución que se calcula como óptima en el instante T_n puede no serlo en T_{n+1}. Es por esto que, los algoritmos que producen soluciones sub-óptimas, pero suficientemente buenas, son aceptables en este ámbito.

Por último, idealmente (aunque no es estrictamente necesario), el algoritmo debía producir resultados repetibles para la misma entrada: esto ayuda con la depuración del mismo y, además, garantiza la consistencia de las soluciones producidas. Por desgracia, los algoritmos generacionales (como los genéticos o los basados en agentes/partículas) no cuentan con esta propiedad, ya que (al usar poblaciones inicializadas aleatoriamente) sus resultados pueden variar y no siempre producirán la misma solución para una entrada en concreto, es decir, no son deterministas.

3.1.3. Algoritmo propuesto

El diseño elegido finalmente se puede observar en el algoritmo 1, y se puede dividir en las siguientes dos partes: la selección de la pareja trabajo-nodo, y la asignación de dichos trabajos a los nodos seleccionados. Por otro lado, el algoritmo recibe un parámetro de entrada *jobsPendientes*, que se trata de una lista con los jobs que han llegado al clúster pero que no se han asignado. Esta lista llega al algoritmo ordenada en función de un criterio (en nuestro caso, la energía que consumirá un trabajo en un nodo de referencia).

Algoritmo 1: Algoritmo de encolado desarrollado.

```
1  Resultado jobsAsignados: Lista(Job)
2  Funcion Schedule jobsPendientes: ListaOrdenada(Job), resources: Lista(Node)
3      jobsAsignados = new Lista(Job)
4      Para cada j: Job en jobsPendientes haz
5          nodoSeleccionado = null
6          energiaMin = Infinito
7          Para cada n: Node en resources haz
8              Si (energia(n, j) ≤ energiaMin) ∧ (n.CoresDisponibles ≥ j.Tasks) entonces
9                  nodoSeleccionado = n
10                 energiaMin = energia(n,j)
11             fin
12         fin
13         Si nodoSeleccionado entonces
14             Para la tupla c: Core en nodoSeleccionado.CoresDisponibles, t: Task en j.Tasks haz
15                 c.asignarTarea(t)
16             fin
17             jobsAsignados.anhade(j)
18         fin
19     fin
20     Para cada j: Job en jobsAsignados haz
21         jobsPendientes.quita(j)
22     fin
23     devuelve jobsAsignados
24 fin
```

En la parte de la selección de la pareja nodo-trabajo, el algoritmo recorre todos los trabajos que todavía no han sido asignados y escoge el nodo más adecuado para cada uno (si es que hay alguno disponible). Para elegir el nodo, el algoritmo determina si el trabajo seleccionado entraría en dicho nodo (el número de tareas que lo componen no excede el número de cores disponibles en el nodo). Además, se calcula el consumo energético del trabajo en dicho nodo, y si es menor que el de los otros nodos examinados hasta el momento, se le escoge como nodo al que asignar el trabajo. En caso de que el algoritmo no encuentre ningún nodo con los suficientes cores disponibles como para ejecutar el job, su asiganción se pospondrá a llamadas futuras del algoritmo. Cabe destacar que, si bien la plataforma en la que se decidió desarrollar el algoritmo (IRMASim) permite saber el grado de contención de memoria que hay en cada nodo del clúster[19], se decidió no diseñar ningún mecanismo en el algoritmo que lo tuviera en cuenta para mantener el algoritmo relativamente simple.

Por otro lado, en la parte de asignación de tareas, el algoritmo utiliza el bucle especial, **Para la tupla**, que realiza un mapeo 1 a 1 de los cores disponibles en el nodo con las tareas del trabajo, limitando el tamaño de la tupla (o lista de tamaño [n x 2]) al del componente más pequeño (que en nuestro caso siempre será el número de tareas, debido a la comparación realizada a la hora de seleccionar un nodo). Además de ir asignando cada tarea a un nodo, cuando termina, el algoritmo marca el trabajo como

asignado. En un paso final, el algoritmo quita de la lista de pendientes todos los trabajos marcados como asignados, para evitar discrepancias en el estado del simulador.

Cálculo de energía

Como se puede ver, el algoritmo 1 hace uso de las ecuaciones descritas en la sección 2.2. En este caso, partimos de la suposición de que el algoritmo conoce el tiempo de ejecución de los trabajos en cada nodo y que no hay ningún otro trabajo ejecutándose en dicho nodo. Estas suposiciones son suficientes a la hora de modelar el algoritmo, sin embargo, como se verá en la sección de la implementación, no son adecuadas para el uso en el simulador, ya que habrá ciertas métricas que no le estarán disponibles al algoritmo (como el tiempo de ejecución de los trabajos en cada nodo, algo que en la realidad puede no saberse *a priori*).

Asimismo, se mencionaba en el apartado anterior que la lista de jobsPendientes que recibe el algoritmo viene ya ordenada en función de su consumo energético en un nodo de referencia (o nodo *baseline*). De nuevo, esta es una suposición útil a la hora de elaborar el concepto del algoritmo, pero no se ajusta a la realidad del simulador, en el que dicha métrica no estará disponible (aunque será calculada por el Workload manager, como se verá más adelante).

3.1.4. Starvation de trabajos

Antes de pasar a la implementación del algoritmo, se detectó un fallo de diseño en el modelo propuesto. El algoritmo podía causar *starvation* de ciertos trabajos. *Starvation* es un fenómeno que se puede dar en algunos algoritmos de planificación por el que algunos de los trabajos de la cola no llegan nunca a ser asignados a los recursos del clúster, debido a que siempre llegan trabajos con más prioridad. En el caso del diseño propuesto, los trabajos con el mayor consumo energético no serían encolados hasta que los trabajos con menor consumo fueran asignados al clúster y liberados. Para una herramienta de planificación en un clúster HPC, este comportamiento no es deseable, ya que significa que ciertos trabajos no son asignados, y por tanto completados, nunca.

Para solucionar este problema, primero se hizo un pequeño estudio de las posibles estrategias disponibles para mitigarlo. Por ejemplo, Fan et al. [15], utilizan un sistema basado en Reinforcement Learning para solucionar el problema de *starvation*. Por otro lado, Moschakis et al. [24] proponen un sistema de mitigación de *starvation* de trabajos basado en la reserva de recursos para los trabajos que van llegando.

Tras ello, se ideó una primera intuición sobre una posible estrategia para dar preferencia a trabajos más antiguos usando como parámetro de ordenación de la lista de trabajos pendientes no solo la energía a consumir, sino también el momento de llegada. Este método tenía sus ventajas, como su simplicidad (haciendo así que el tiempo de ejecución del algoritmo no creciera demasiado) o su capacidad para agrupar trabajos respecto a su tiempo de llegada, escogiendo entre ellos el de menor energía, pero manteniendo cierta separación entre trabajos distantes temporalmente.

Sin embargo, este método se descartó debido a que la ordenación de la lista por los dos parámetros no garantizaba un umbral temporal estricto en el que un trabajo fuera asignado al excederlo. Asimismo, la ordenación de la lista por dos parámetros con unidades y magnitudes diferentes (julios en caso de la energía y segundos en el del tiempo de llegada) podía dificultar las ordenaciones, al no ser comparables estos dos tipos de unidades.

Como método alternativo a dicha primera intuición, se decidió añadir una nueva parte al algoritmo diseñado anteriormente, que asignara los trabajos que excedieran un umbral de tiempo desde que aparecieron hasta el momento actual. De esta manera, los trabajos más antiguos tendrían prioridad sobre el resto, asegurando así que ningún trabajo se quede sin ser planificado. La modificación realizada al diseño se puede ver en el algoritmo 2. Como se puede ver, esta solución es algo más pesada que la anterior, ya

que requiere recorrer la lista de trabajos pendientes 2 veces, una para asignar los que exceden el umbral temporal y otra para asignar el resto en función de su consumo energético. Asimismo, es importante destacar, para este diseño, la importancia de definir un umbral temporal correcto, ya que de lo contrario, cualquiera de los 2 aspectos del algoritmo podrían sufrir, ya sea la parte de mitigación de la *starvation* o la parte de asignar trabajos por su consumo energético.

Algoritmo 2: Mecanismo para evitar starvation en la cola de trabajos

```
1  Funcion Schedule jobsPendientes: ListaOrdenada(Job), resources: Lista(Node)
2      jobsAsignados = new Lista(Job)
       // Umbral temporal en segundos, se ha establecido en esta figura en 300s (5
          minutos) como ejemplo
3      tiempoUmbral = 300
4      Para cada j: Job en jobsPendientes haz
5          Si (sim.TiempoActual − j.Tiempo) ≥ tiempoUmbral entonces
               // asignaNodo se utiliza como abreviatura de la lógica utilizada en la
                  siguiente parte, ya que es igual
6              asignaNodo(j, resources)
7          fin
8      fin
9      Para cada j: Job en jobsAsignados haz
10         jobsPendientes.quita(j)
11     fin
12     jobsAsignados = new Lista(Job)
13     Para cada j: Job en jobsPendientes haz
14         ...
15     fin
16     ...
17 fin
```

3.2. Implementación

Una vez completada la fase de diseño del algoritmo, se pasó a realizar la implementación de un Workload Manager de IRMASim que hiciera uso de él. Éste WM (Workload Manager) debía permitir asignar trabajos a los nodos del clúster en función no solo de su posible consumo energético, sino también de la eficiencia energética, dependiendo de cuál de las 2 métricas se quiera optimizar.

El proceso de desarrollo del WM se puede dividir en tres partes fundamentales: la extensión del simulador para que incluya ciertas métricas de energía, el desarrollo de una prueba de concepto basándose en un workload manager ya implementado en IRMASim, y por último, la refactorización de la prueba de concepto, reescribiéndola para eliminar dependencias innecesarias.

3.2.1. Extensiones a IRMASim

Como comienzo al desarrollo, se extendió la funcionalidad del simulador, haciendo que exportara métricas de energía y eficiencia energética al terminar su ejecución. Cabe destacar que, mientras que la métrica de energía ya era calculada por el simulador internamente, la eficiencia energética no, con lo que fue necesario implementar su cálculo en el runtime del simulador.

Asimismo, además de extender el reporte de métricas del simulador, fue necesario extender el modelo de datos de los trabajos, añadiendo una variable nueva: la estimación de la energía requerida para

completar un trabajo. El nuevo dato se añadió para poder organizar los trabajos según su consumo energético cuando llegan al simulador.

Esta nueva variable (representada como *req_energy* en el fichero de configuración de los trabajos) simboliza una estimación que el usuario hace del consumo energético de cada una de las tareas del trabajo que está lanzando al clúster. *Req_energy* está basado en el parámetro "–power" del gestor de colas *slurm*, con el que se pueden especificar flags relacionadas con el consumo de potencia del trabajo.[34]

3.2.2. Implementación del Workload Manager

Una vez implementadas las extensiones descritas, se pasó a desarrollar el WM que hiciera el scheduling basado en energía. Este WM debía cumplir con los siguientes requisitos:

- Debía emplear el algoritmo diseñado para hacer el scheduling (aunque con ciertas modificaciones).
- Debía permitir la asignación de trabajos no solo en base a su consumo energético sino también a su eficiencia energética en los nodos del clúster.
- Para realizar la asignación, debía hacer uso sólo de las métricas y parámetros expuestos por IRMASim a los WM, siendo para ello necesario reemplazar la parte del algoritmo que calculaba la energía de cada nodo para un trabajo por ecuaciones que hicieran estimaciones de dicho consumo.

Como paso inicial en el desarrollo, se partió de uno de los WM ya existentes en IRMASim, en el que se incluyó una versión simplificada del algoritmo de cálculo de energía y eficiencia energética. Tras examinar el comportamiento de esta versión inicial de PCBE, se reescribió, realizando algunas optimizaciones para mejorar el tiempo de ejecución y eliminando las partes del código del workload manager usado como base que no eran necesarias.

En el diagrama de la figura 3.1 se describe el workload manager finalmente implementado, con todos sus componentes y métodos relevantes, así como el flujo del programa cuando éste es invocado. Para entender mejor su funcionamiento, en la siguiente sección se explica más en profundidad la imlementación final.

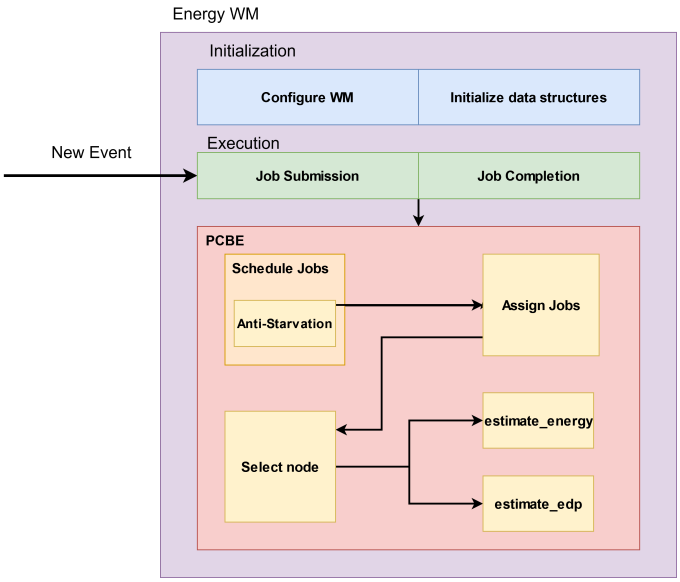

Figura 3.1: Diagrama de los diferentes componentes del WM desarrollado.

Implementación final

Inicialización. La primera pieza del Workload Manager que se ejecuta cuando se inicia una simulación es el constructor de la clase. Durante la inicialización del WM, éste consulta las opciones pasadas al simulador y a partir de ahí configura su comportamiento. Entre los parámetros que se configuran a partir de dichas opciones están: el objetivo a optimizar, ya sea energía o eficiencia, cómo priorizar los trabajos, mayor o menor consumo/eficiencia, y qué tipos de nodo elegir, más o menos potentes/eficientes primero. La implementación de esta configurabilidad se puede ver en el diagrama de la figura 3.2, en la que se ilustra cómo, a partir del fichero "options.json" el simulador le pasa al WM los parámetros necesarios. El WM utiliza el campo "criterion" para decidir si PCBE debe optimizar el consumo o la eficiencia energética del clúster. Con este criterio definido, las políticas de selección de nodo, "node_policy", y de selección de trabajos, "job_policy" se escogen, cambiando su comportamiento en función del objetivo a optimizar. El funcionamiento del método "estimate_energy" (y "estimate_edp") se explicará más adelante, en la sección 3.2.2.

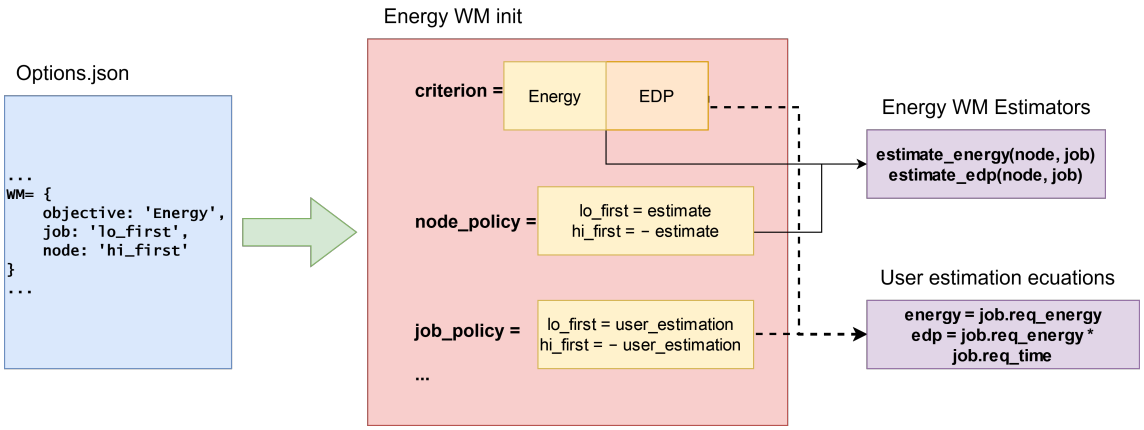

Figura 3.2: Diagrama de la configuración que realiza el WM durante su inicialización

Asimismo, durante esta inicialización, el WM crea las estructuras de datos que va a usar durante su ejecución para saber qué trabajos están pendientes ("pending_jobs") y cuáles se están ejecutando ("running_jobs"). Estas estructuras de datos se aprovechan de un constructo de la librería de python *Sorted Containers*, la *SortedList*, que implementa un árbol binario como estructura de datos para mantener sus elementos ordenados respecto a una clave [20]. La inicialización de una *SortedList* vacía es muy rápida, y el insertar elementos también; sin embargo, debido a la estructura de datos subyacente, la implementación de *SortedList* no permite usar una key dinámica que cambie con el tiempo. Además de la inicialización de estas dos listas, también se crean datos de apoyo adicionales, usados por el algoritmo entre iteraciones, como un diccionario con los trabajos asignados a cada nodo del clúster o el atributo auxiliar "min_freq" usado durante las estimaciones de consumo/eficiencia. En la figura 3.3 se pueden ver estas inicializaciones.

Una vez terminada la inicialización y configuración del WM, el simulador pasa a ejecutar el buble principal.

Ejecución. En el diagrama de la figura 3.4 se puede ver la secuencia de llamadas que se realiza en el simulador con el Workload Manager final implementado. Cabe destacar que los eventos *job_arrived* y *job_finished* son eventos internos a *Simulator* que dispara esta clase, pero que se han representado como externos para ilustrar mejor cuál es el flujo de llamadas en IRMASim. Asimismo, la parte del *Algoritmo* es interna al Workload Manager.

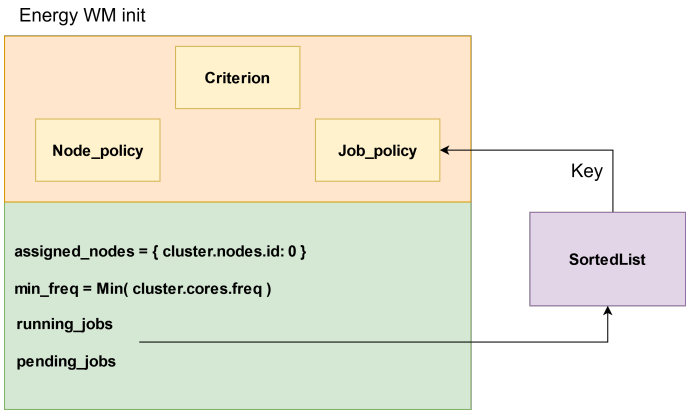

Figura 3.3: Diagrama de la creación de las diferetes estructuras de datos creadas por el WM durante su inicialización.

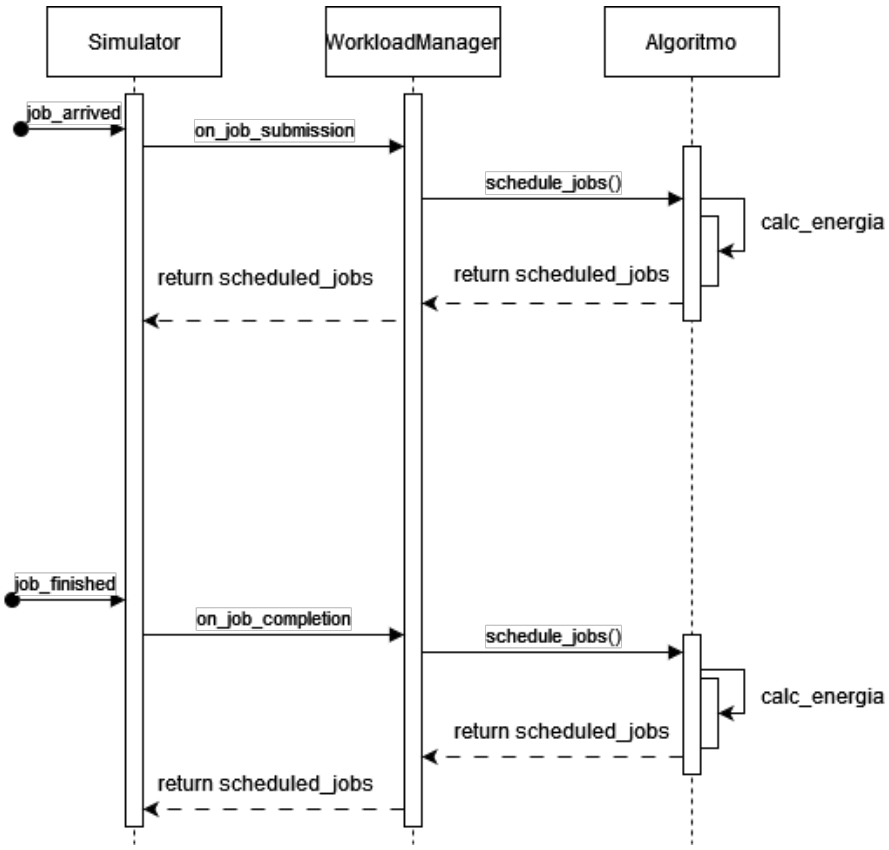

Figura 3.4: Diagrama de secuencia del proceso de scheduling en el simulador

Como se puede ver en el diagrama 3.4, cuando un trabajo nuevo "entra" al clúster/simulador (el simulador sabe en qué momentos va a llegar cada trabajo desde el inicio, pero espera a dicho instante para simular que ha llegado uno) éste notifica al WM, invocando su método *on_job_submission()*. Éste, al recibir el evento, añade el trabajo pertinente a la lista de trabajos por asignar y llama al método *schedule_jobs()* que se encarga de asignar los trabajos pendientes, usando para ello unas ecuaciones que aproximan el consumo energético o eficiencia en los nodos (dependiendo de qué se quiera optimizar y de qué manera). A continuación, *schedule_jobs()* obtiene en qué nodo es más conveniente ejecutar cada trabajo (si es que puede ser ejecutado en ese instante) y los asigna, marcándolos para borrarlos de la lista de pendientes y añadirlos a la lista de trabajos ejecutándose (interna al WM). Una vez terminado esto, tanto *schedule_jobs()* como el WM terminan su ejecución, devolviendo el control al simulador, que

se encargará de simular el nuevo estado del clúster hasta que llegue un nuevo evento.

Asimismo, cuando un trabajo termina de ejecutarse, el simulador invoca al WM mediante su método *on_job_completion()*. Para responder a este evento, el WM quita de su lista de trabajos ejecutándose el que se acaba de completar y, si todavía quedan trabajos pendientes, invoca al algoritmo para asignar nuevos trabajos.

La implementación tanto de *on_job_submission()* como de *on_job_completion()* es similar; ambos métodos modifican las listas internas del WM cuando llega su respectivo evento del simulador y, acto seguido, pasan a asignar nuevos trabajos al clúster (si es posible).

Para asignar nuevos trabajos, ambos métodos hacen una llamada a PCBE, que se encuentra implementado en el método *schedule_jobs()*. Este método (haciendo uso de otros métodos auxiliares) examina los trabajos pendientes de asignar al clúster y decide cuál de ellos encolar y a qué nodo, siguiendo la estrategia descrita anteriormente en el algoritmo 1. El código de la implementación se puede ver en 3.1, donde este método realiza las operaciones de más alto nivel, e incorpora el mecanismo *anti-starvation* explicado en la sección 2. En la llamada a *assign_job()* es donde el algoritmo realiza las operaciones para decidir el mapeo trabajo-nodo.

```
class Energy(WM):
    def schedule_jobs(self):
        # List of jobs to assign
        to_assign = []

        # Anti-starvation mechanism
        for j in self.pending_jobs:
            if (self.simulator.simulation_time - j.submit_time) >= 60:
                to_assign = self.assign_job(j, to_assign)

        for job in to_assign:
            self.pending_jobs.remove(job)
            self.running_jobs.add(job)

        to_assign = []

        # Energy based scheduling
        for j in self.pending_jobs:
            to_assign = self.assign_job(j, to_assign)

        for job in to_assign:
            self.pending_jobs.remove(job)
            self.running_jobs.add(job)
```

Código 3.1: Implementación a alto nivel del algoritmo de scheduling

Assign_job() (ilustrado en el código 3.2) es la parte del algoritmo que realiza la asignación de los trabajos pendientes a los nodos del clúster (si es que puede encontrar uno donde dicho trabajo pueda ejecutarse). Una vez elegido el nodo, usando el método auxiliar *select_node()*, el método asigna cada una de las tareas a uno de los cores vacíos del nodo, hasta encolarlas todas (esto es lo que hará que en el siguiente paso del simulador se simule el trabajo recién asignado). Tras esto, el método marca dicho trabajo como asignado y lo contabiliza en el diccionario que guarda los trabajos ejecutándose en cada nodo.

```
class Energy(WM):
    def assign_job(self, j, assigned_list):
        # Search for the best node
        selected_node = self.select_node(j)

        # If a suitable node is found, schedule
        if selected_node is not None:
```

```
            for task in j.tasks:
                for core in selected_node.cores():
                    if core.task is None:
                        ...
                        break
            # Mark as scheduled
            assigned_list.append(j)

            # Update the number of running jobs in chosen node
            self.assigned_nodes[selected_node.id] += 1
        return assigned_list
```

Código 3.2: Implementación de la asignación de trabajos a nodos del clúster.

El método auxiliar *select_node()* es el encargado de, dado el trabajo a asignar y el estado actual del clúster, encontrar el mejor nodo para dicho trabajo. Para ello se apoya en unas ecuaciones que realizan una estimación de la energía/eficiencia que el trabajo tendrá en cada nodo y que varían dependiendo de la configuración escogida. El código 3.3 ilustra el método de selección de nodo, en el que se busca, el nodo con el "score" más bajo que tenga suficientes cores disponibles como para alojar todas las subtareas del trabajo.

```
class Energy(WM):
    def select_node(self, job):
        selected_node = None
        best = float('inf')

        for node in self.resources:
            if node.count_idle_cores() >= job.ntasks_per_node:
                estimate = self.node_estimation(node, job)
                if estimate < best:
                    best = estimate
                    selected_node = node

        return selected_node
```

Código 3.3: Implementación de la selección de nodo para un trabajo.

Estimación de consumo energético y eficiencia

Como se puede ver en el código 3.3, la estimación del "score" del nodo se hace mediante la función *node_estimation()*, cuya implementación varía dependiendo de si se optimiza energía o eficiencia y de qué tipo de nodos se quiera priorizar. Sin embargo, todas las implementaciones hacen uso de las mismas ecuaciones para realizar las estimaciones.

Para realizar la estimación del consumo energético que un trabajo tendrá en un nodo concreto, es necesario calcular primero la potencia dinámica que consumirán todas las subtareas del trabajo y la fracción de la potencia estática que consumirá por ejecutarse en dicho nodo (teniendo en cuenta el número de tareas que ya se están ejecutando en ese mismo nodo). Las ecuaciones para calcular estas potencias se pueden ver en la figura 3.5

$$Pd_j = Pd_c \cdot Ta_j \tag{3.1}$$

$$Ps_j = \frac{Ps_n}{Js_n + 1} \tag{3.2}$$

Figura 3.5: Estimación del consumo de potencia dinámica y estática de un job j asignado a un nodo n. Ps representa la potencia estática, Pd la potencia dinámica, Ta el número de subtareas de un job y Js el número de jobs ejecutándose en un nodo.

Asimismo, es necesario calcular el tiempo de ejecución que tomará el job en ejecutarse en dicho nodo. Para ello, el algoritmo usa la estimación del tiempo de ejecución, que provee el usuario con el job, como baseline, calculando el speedup relativo del job. Para calcular el speedup del trabajo en un nodo, se utiliza no solo el speedup de frecuencia del nodo considerado respecto al más lento del clúster, sino también la cantidad de instrucciones por ciclo (o *dpflops*) que es capaz de realizar. Las ecuaciones usadas para esta estimación se pueden ver en la figura 3.6.

$$T_j = Treq_j \cdot Sp_n \tag{3.3}$$

$$Sp_n = \frac{F_{min}}{F_n} \cdot \frac{1}{dpflops_n} \tag{3.4}$$

Figura 3.6: Estimación del tiempo de ejecución de un job j asignado a un nodo n. T representa el tiempo de ejecución, $Treq$ el tiempo baseline de ejecución y Sp el speedup. Por otro lado, F representa la frecuencia de un nodo, con F_{min} representando la frecuencia más baja del clúster, y $dpflops$ simboliza las operaciones por ciclo que el nodo es capaz de realizar.

Una vez se tienen estos dos componentes, se puede estimar la energía que consumirá el trabajo usando la ecuación descrita anteriormente en 2.1 o el EDP, usando la ecuación 2.5. Los códigos 3.4, 3.5 y 3.6 ilustran los métodos implementados en el WM para realizar los cálculos descritos en esta sección.

El código completo de PCBE, implementado como un workload manager de IRMASim, se puede encontrar en este repositorio.

```
class Energy(WM):
    def estimate_speedup(self, node):
        # Get info about node properties
        node_info = node.cores()[0]

        freq_speedup = (self.min_freq / node_info.clock_rate)
        inverted_dpflops = ((node_info.clock_rate * 1e3) / node_info.mops)
        return freq_speedup * inverted_dpflops
```

Código 3.4: Implementación del cálculo del speedup del trabajo en un nodo.

```
class Energy(WM):
    def node_energy(self, job: Job, node):
        # Get info about node properties
        node_info = node.cores()[0]

        # Get dynamic power of job in node
        dyn_fraction = node_info.dynamic_power * job.ntasks

        # Get static fraction of power for job in node
        running_node_jobs = self.assigned_nodes[node.id]
        static_fraction = node_info.static_power*len(node.cores())
        static_fraction /= (running_node_jobs + 1)

        # Get the job time in the node
        node_time = job.req_time * self.estimate_speedup(node)

        energy = node_time * (dyn_fraction + static_fraction)
        return energy
```

Código 3.5: Implementación del cálculo de la energía de un trabajo en un nodo.

```
class Energy(WM):
    def node_edp(self, job: Job, node):
        node_time = job.req_time * self.estimate_speedup(node)

        energy = self.node_energy(job, node)
```

```
        return energy * node_time
```

Código 3.6: Implementación del cálculo de la eficiencia energética de un trabajo en un nodo.

Complejidad temporal

Dado el tipo de aplicación para el que se ha desarrollado el algoritmo y WM descritos durante este capítulo, es importante realizar un análisis de la complejidad temporal de todos los componentes implementados, para poder predecir cómo escalará el rendimiento de las diferentes piezas con el tamaño del clúster y el número de trabajos a planificar. Este análisis de complejidad se realizará siguiendo un enfoque *bottom-up*, utilizando la notación *Big-O*, que se usa para denotar la complejidad temporal para peor de los casos.

Estimación de energía. Tanto el cálculo del speedup (código 3.4) como las estimaciones de energía (código 3.5) y de eficiencia (código 3.6) son ecuaciones simples que no iteran sobre colecciones. Aunque sí que es cierto que se acceden a elementos y propiedades internos a ciertas colecciones, asumimos que la implementación de estas colecciones en python permite un tiempo de acceso constante a sus propiedades. Es por estas razones que este componente del algoritmo tiene una complejidad temporal constante, es decir, $O(1)$.

Selección de nodo. Dentro del método de la selección de nodo (código 3.3), se encuentra una llamada a la estimación de energía (que como se ha explicado en el párrafo anterior tiene complejidad $O(1)$), dentro de un bucle que recorre todos los nodos del clúster. Dentro de este mismo bucle, se encuentra otro anidado que recorre los cores del nodo. Por lo tanto, la selección de nodo tiene una complejidad temporal lineal, es decir $O(n)$.

Asignación de trabajo. Durante la asignación de trabajos, el algoritmo hace la llamada a *select_node()*, que tiene una complejidad de $O(n)$. Asimismo, dentro del método, hay un bucle que itera sobre todas las sub-tareas del trabajo y cores del node. Por lo tanto, ésta parte del algoritmo también tiene una complejidad lineal, $O(n)$.

Scheduling de trabajos nuevos. Por último, el método *schedule_jobs()* hace dos invocaciones a *assign_job()* (con complejidad $O(n)$) dentro de los bucles que recorren la lista de trabajos pendientes. Además de esto, se realizan dos modificaciones de las listas internas del WM. Según la documentación de la librería de *SortedList*[20], las operaciones de inserción y eliminación tienen ambas un coste de $O(logn)$. Dado que estas operaciones se ejecutan dentro de un bucle que itera sobre los jobs a asignar (y que ésta será como máximo el tamaño de la lista de trabajos pendientes), la complejidad de estas partes es:

$$O(n) \cdot (O(logn) + O(logn) =$$
$$O(2 \cdot n \cdot logn) =$$
$$O(n \cdot logn)$$

Finalmente, juntando todas las partes del método, obtenemos que la complejidad temporal final del algoritmo es

$$2 \cdot O(n \cdot logn) + 2 \cdot (O(n) * O(m)) =$$
$$2 \cdot O(n \cdot logn) + 2 \cdot (O(n) * O(n)) =$$
$$2 \cdot O(n \cdot logn) + 2 \cdot O(n^2) =$$
$$O(n \cdot logn) + O(n^2) =$$
$$O(n^2)$$

Atención de eventos. Como el algoritmo es invocado cuando llega un evento del simulador (llega un nuevo trabajo a asignar o un trabajo ha terminado de ejecutarse), es necesario examinar también su complejidad temporal, para poder determinar la complejidad final del WM. En este caso, ambos métodos son muy parecidos y se tratarán como iguales. Cuando se recibe un evento, se modifican las listas del WM (que como ya se ha explicado en el párrafo anterior, tiene como mucho una complejidad de $O(n \cdot logn)$) y luego se ejecuta el algoritmo, con complejidad $O(n^2)$. La complejidad temporal final de estos métodos es:

$$O(n \cdot logn) + O(n^2) =$$
$$O(n^2)$$

Como se puede ver, la complejidad temporal de PCBE tiene un crecimiento cuadrático. Si bien este resultado no es ideal y deja espacio para realizar algunas optimizaciones, su tiempo de ejecución, como se ha comprobado durante la realización de los experimentos de escalado (capítulos 4 y 5), es aceptable, incluso para espacios del problema grandes (alrededor de 1000 trabajos y 300 nodos).

Capítulo 4

Experimentos

Una vez diseñado e implementado el nuevo Workload Manager para IRMASim, es necesario probar el rendimiento de las posibles opciones que permite, para poder elegir la más óptima. Asimismo, es necesario comparar PCBE con otros planificadores más establecidos, como FCFS o SJF. Para ello se han elaborado un conjunto de experimentos, que examinan el comportamiento del algoritmo con los diferentes parámetros, y cómo escala el rendimiento de éste frente a simulaciones complejas (con un número elevado de trabajos en clústers grandes).

4.1. Herramientas

Antes de empezar a diseñar los experimentos, fue necesario crear algunas herramientas que facilitaran el proceso de prueba de los planificadores y de extracción de resultados. Estas herramientas se implementaron en python, el mismo lenguaje que usa IRMASim, para simplificar su desarrollo y la ejecución de los experimentos.

La primera herramienta desarrollada fue un script que generaba las definiciones de clústers HPC a usar con el simulador. El script ya había sido implementado por el grupo de ATC de la Universidad de Cantabria en Perl, con lo que sólo fue necesario traducirlo a python. Este primer script: "generate_platform", toma como parámetros de entrada las definiciones de los nodos de un clúster (como una combinación de 3 valores separados por comas) y, haciendo uso de una tabla con datos de diferentes procesadores, genera un fichero de configuración de IRMASim (en formato JSON) que contiene la especificación del clúster. Además, para los experimentos de escalado fue necesario modificarlo ligeramente para que permitiera generar nodos con un mayor número de cores.

Además de ese script, se escribieron otros dos, completamente nuevos. Estos generan los otros ficheros de configuración necesarios para que IRMASim realice una simulación: la especificación de los trabajos a simular, y las opciones del simulador (qué Workload Manager usar y con qué políticas).

El primero de estos scripts, "generate_workload", tiene un mecanismo similar al de generate_platform. Toma como parámetros de entrada una lista con los diferentes trabajos a simular (definidos por 3 valores, el tiempo de llegada, el número de subtareas, y el perfil del trabajo: High_Intensity, Med_Intensity, Low_Intensity). Tras recibir esta lista, el script compone un archivo JSON que contiene la especificación de todos los trabajos compatible con IRMASim.

Por otro lado, el segundo script, "generate_options", genera el fichero de configuración global que se usará al ejecutar el simulador. Para ello, recibe como argumentos el workload manager a emplear (Energía, EDP o Heurístico) así como las políticas de selección de trabajos y nodos. Este script asume que ya existen los otros 2 ficheros de configuración de IRMASim necesarios para definir una simulación, y que éstos tienen los nombres: "platform.json" y "workload.json".

Por último, se desarrolló una última herramienta para ejecutar los experimentos diseñados de manera desatendida. El script utiliza un fichero de configuración para generar el entorno de la simulación de IRMASim (usando los programas descritos anteriormente), y después de esto ejecuta cada uno de los experimentos (también contenidos en el fichero de configuración). Asimismo, permite (mediante argumentos) usar una plataforma o traza de trabajos pregenerada, ignorando las especificaciones que puedan contener los experimentos.

Además, cada vez que un experimento termina, el script recoge los datos que exporta IRMASim y los agrupa, procesándolos y generando gráficas comparativas que faciliten el análisis de los resultados. Si las trazas de trabajos usadas son lo suficientemente pequeñas (no más de 100 tareas en total), la herramienta genera gráficos ilustrando la asignación que los diferentes WM han hecho de los trabajos, para permitir un análisis más detallado de los diferentes algoritmos.

4.2. Metodología

Tras haber implementado tanto el WM como las herramientas a usar durante la realización de los experimentos, se pasó a diseñar pruebas para caracterizar PCBE y compararlo con los otros algoritmos disponibles en el simulador.

Para ello se decidió realizar dos tipos de experimentos: experimentos de comparación, que permitieran examinar las diferencias entre las políticas de selección de trabajos y nodos disponibles en las dos variantes de PCBE, la basada en energía (PCBE_n) y la basada en eficiencia energética (PCBE_f); y experimentos de escalado, que permitieran analizar el comportamiento de ambas frente algoritmos más tradicionales, para trazas de trabajos de gran tamaño en plataformas con muchos recursos.

4.2.1. Experimentos de comparación

Como primer paso para caracterizar PCBE (y compararlo con otras heurísticas de planificación), se decidió realizar unos experimentos iniciales. Éstos estaban enfocados a determinar cuál es la mejor combinación de políticas de selección que ofrecen PCBE_n y PCBE_f, y a examinar su comportamiento, utilizando para ello trazas de tamaño reducido que permitieran un fácil análisis.

Plataforma escogida

Para realizar estos experimentos, se decidió utilizar una plataforma existente, para limitar el número de variables del experimento. Esta plataforma es la usada en algunos de los ejemplos de uso de IRMASim y está compuesta por 2 nodos básicos. Cada nodo contiene un procesador con 4 cores, sin embargo, estos 2 procesadores son totalmente distintos: uno de ellos tiene una alta frecuencia (3.4 GHz) y consumo energético, mientras que el otro es más lento (1.5GHz) pero consume menos energía. Esta arquitectura de clúster es ideal para estos experimentos, ya que permite identificar con facilidad si los trabajos han sido asignados al nodo energéticamente eficiente o al más rápido.

Workloads generados

Por otro lado, se diseñaron trazas de trabajos sintéticas para los experimentos. En concreto, se diseñó una traza por cada combinación de políticas (2 políticas de selección de trabajos y otras 2 de selección de nodos) y algoritmo de planificación (2 algoritmos de planificación: energía y EDP). En total se generaron $2^3 = 8$ trazas únicas, cada una de ellas diseñada para dar los mejores resultados posibles en una de las posibles combinaciones.

Todas las trazas seguían un patrón concreto: debían tener la misma cantidad de subtareas totales (en concreto 22), debían saturar todos los cores del clúster y debían llegar todos los trabajos entre los instantes 0 y 2 de tiempo. Asimismo, los trabajos podían ser de uno de tres perfiles:

- "high_intensity" (con un número de instrucciones elevado)
- "low_intensity"(con un número de instrucciones reducido)
- "medium_intensity" (un punto intermedio entre los otros 2 perfiles)

Cabe destacar que, el tipo de perfil no determina el número de subtareas que podía tener el trabajo.

Pruebas diseñadas

Como experimentos, se decidió realizar 8 pruebas (una por cada traza diseñada) que compararan el rendimiento de las diferentes políticas entre ellas y contra dos de los algoritmos heurísticos de planificación más usados: First Come First Serve [15] y Shortest Job First[27]. Las 8 pruebas se dividieron en 2 grupos, las que examinaban el rendimiento del algoritmo que prioriza consumo energético (PCBE_n) y las que examinaban el algoritmo de eficiencia energética (PCBE_f).

Como métricas recogidas en cada prueba, se recogieron datos sobre el consumo energético total del clúster durante la simulación, la eficiencia energética, y el tiempo total de ejecución de los trabajos en el clúster. Asimismo, se recogieron *logs* detallados sobre la asignación de los trabajos, para luego generar gráficas detalladas que permitieran analizar el comportamiento de cada uno de los planificadores. En la tabla 4.1 se pueden ver las diferentes políticas probadas para cada planificador y en la figura 4.1 se puede ver un ejemplo de las gráficas de trabajos diseñadas.

Planificador	Políticas de trabajos	Políticas de nodos	Combinaciones Probadas
PCBE_n	Menor consumo primero (Lj) Mayor consumo primero (Hj)	Menor consumo primero (Ln) Mayor consumo primero (Hn)	LjLn LjHn HjLn HjHn
PCBE_f	Menor EDP primero(Lj) Mayor EDP primero(Hj)	Menor EDP primero(Ln) Mayor EDP primero(Hn)	LjLn LjHn HjLn HjHn
Heurístico	Primer trabajo Trabajo más corto	Primer nodo disponible	FCFS SJF

Tabla 4.1: Tabla con las políticas disponibles en los planificadores. En la última columna se detalla cuáles han sido las combinaciones de políticas probadas para cada planificador (primera columna) en los experimentos de comparación.

4.2.2. Experimentos de escalado

Una vez completados los experimentos de comparación, se pasó a realizar pruebas que ilustrasen cómo se comportaban los planificadores desarrollados para conjuntos de trabajos más grandes y complejos, en plataformas más heterogéneas. Además, en estas pruebas se decidió comparar dichos planificadores con algunos algoritmos heurísticos.

Sin embargo, como paso previo al diseño de las pruebas, se tuvieron que implementar otras 2 herramientas más, que ayudaran a tratar las trazas de trabajos que se iban a usar. El primero de estos scripts, "add_energy", coge una traza sin estimaciones de energía (el parámetro "req_energy" del planificador) y las añade a cada trabajo, estimando para cada uno la cantidad de energía que va a consumir. Para generar esta estimación de energía, se usó la ecuación 2.1, usando como el tiempo T, la estimación del tiempo de ejecución del trabajo ("req_time"), y como potencia P, la potencia dinámica del nodo menos potente que puede generar el script "generate_platform" (que en este caso eran $\approx 3W$). Por otro lado, el segundo script, "split_workload", parte una traza dada en tantos trozos, de igual tamaño, como se le especifique (por defecto 30).

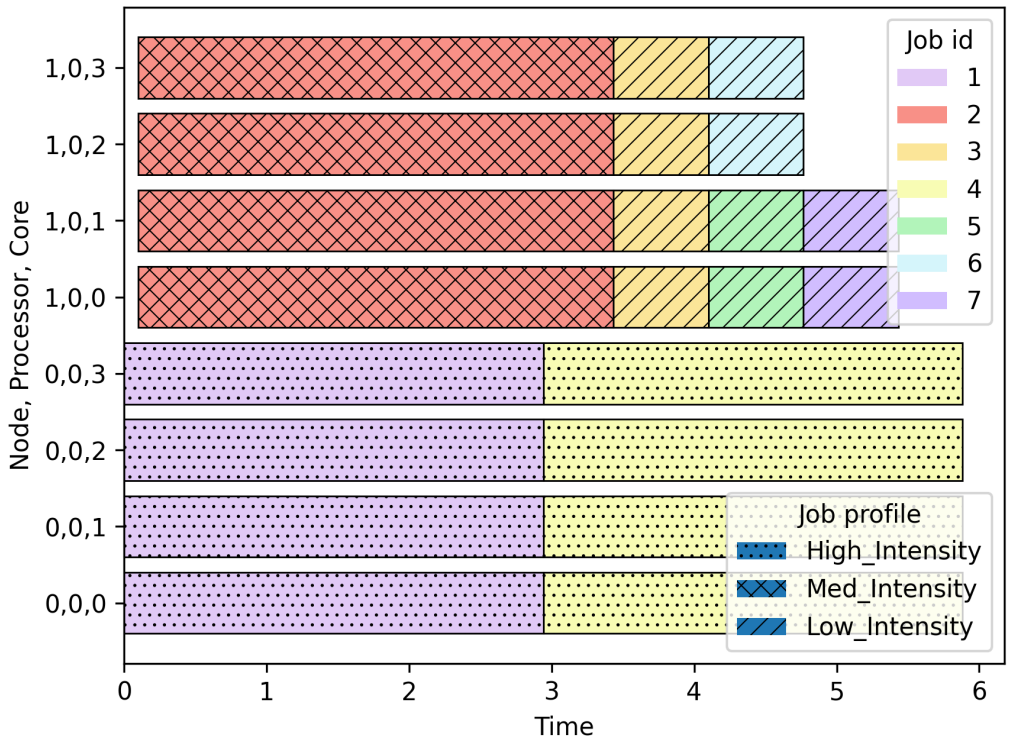

Figura 4.1: Ejemplo de uno de los diagramas de planificación generados, donde el eje x es el tiempo, en las unidades internas del simulador, el eje y denota cada uno de los cores del clúster (con la nomenclatura "nodo, procesador del nodo, core del procesador"). El orden de llegada de los trabajos está representado por su id/color, y el tipo de trabajo por el relleno de la barra.

Plataforma generada

Para estos nuevos experimentos, se decidió usar una plataforma de mayor tamaño, en parte para reflejar mejor las especificaciones de un posible clúster HPC real, y en parte porque, debido a la traza que se iba a usar, era necesario que esta plataforma tuviera un mayor número de cores que pudieran atender todos los trabajos que iban a llegar durante la simulación.

Esta plataforma se generó usando el script "generate_platform" y estaba compuesta por 9 nodos de alto rendimiento (con una frecuencia de 4.2 GHz y 8 cores) y 3 nodos más eficientes (con una frecuencia de 3.0 GHz y 64 cores). Se eligieron estos números de nodos porque eran los máximos que se podían utilizar antes de que quedaran recursos sin utilizar y el tiempo de llegada de los trabajos de la traza se convirtiera en el factor limitante del experimento.

Workload escogido

Para el workload a usar durante estas pruebas, se eligió una traza que contenía 891 trabajos con un número variable de hasta 64 subtareas. Esta traza se obtuvo de *"Logs of Real Parallel Workloads from Production Systems"* de la Universidad Hebrea de Jerusalén[16], en concreto se usó la traza *"The KIT ForHLR II log"*, modificada ligeramente, restringiendo el número máximo de subtareas de los trabajos a 64. Aunque existen 2 versiones de este workload: uno con bajo volumen de accesos a memoria y otro con un alto volumen de accesos a memoria, se decidió usar sólo el de bajo volumen, ya que se querían comparar las diferentes planificaciones que realizaban los algoritmos en un entorno "ideal", en el que el rendimiento del clúster se viera afectado lo menos posible por el ancho de banda de la memoria de los

nodos simulados. Asimismo, la traza fue dividida también en 20 trozos (de unos 45 trabajos cada uno) para observar el comportamiento de los planificadores en las diferentes situaciones que contenía. Se eligió esta cantidad debido a que este tamaño de segmento era el adecuado (en la mayoría de situaciones) para saturar el clúster, y por lo tanto poder observar mejor las diferencias en la planificación de los algoritmos

Pruebas realizadas

Para estos experimentos se diseñaron 2 pruebas diferentes, usando la misma traza de trabajos y plataforma. En ambas pruebas se compararon los planificadores de energía y eficiencia (usando la configuración que había obtenido mejores resultados en las pruebas anteriores: priorizar los trabajos más largos enviándolos a los nodos de menor consumo), con varios algoritmos heurísticos.

Entre los algoritmos heurísticos probados están FCFS y SJF, así como otros dos que realizan la planificación priorizando los trabajos más cortos, mandándolos a los nodos con mayor IPC o a los de menor potencia. En la tabla 4.2 se pueden ver los algoritmos escogidos finalmente. Estos 2 últimos planificadores se eligieron porque, de las posibles políticas que poseía el WM heurístico de IRMASim, las elegidas eran las que más se acercaban a las desarrolladas para los planificadores de energía y se quería ver qué diferencias podía haber entre ambas.

La primera de las pruebas consistió en ejecutar toda la traza de golpe, para poder observar la diferencia de los resultados obtenidos por todos los algoritmos. De esta manera, se podría comparar los planificadores desarrollados con los algoritmos más establecidos y ver si se obtiene, para situaciones más realistas, un posible beneficio sobre éstos últimos, tanto en términos de tiempo, como de energía y eficiencia energética.

Para la segunda prueba, se partió la traza de trabajos en 20 segmentos, y se volvieron a comparar los planificadores escogidos para la anterior prueba. Esto se hizo con el objetivo de estudiar de manera más detallada el rendimiento de PCBE a lo largo de la ejecución de la traza, pudiendo analizar así en qué situaciones se comporta mejor que los heurísticos y en qué situaciones es peor.

Por último, se recogieron métricas tan solo de tiempo, consumo y eficiencia energética para las 2 pruebas, generando para la primera gráficas de barras que ilustraran las diferencias entre todos los algoritmos. Para la segunda prueba, se generaron nubes de puntos con las métricas obtenidas en cada segmento para cada planificador.

Planificador	Políticas de trabajos	Políticas de nodos	Combinaciones Probadas
PCBE$_n$	Menor consumo primero (Lj) Mayor consumo primero (Hj)	Menor consumo primero (Ln) Mayor consumo primero (Hn)	HjLn
PCBE$_f$	Menor EDP primero(Lj) Mayor EDP primero(Hj)	Menor EDP primero(Ln) Mayor EDP primero(Hn)	HjLn
Heurístico	Primer trabajo (Fj) Trabajo más corto (Sj)	Primer nodo disponible(Fn) Nodo con mayor IPC (Qn) Nodo de menor potencia (Sn)	FjFn (FCFS) SjFn (SJF) SjQn SjSn

Tabla 4.2: Tabla con algunas de las políticas disponibles en los planificadores. En la última columna se detalla cuáles han sido las combinaciones de políticas usadas en las pruebas de escalado.

Por último, tanto las gráficas generadas, como los datos brutos, además de los ficheros que definen todas las pruebas realizadas durante ambos experimentos (y las herramientas usadas, a excepción de IRMASim) se pueden encontrar en el repositorio de Github irmasim experiments.

Capítulo 5

Evaluación

Tras definir y ejecutar los experimentos, se recogieron los resultados experimentales, procesándolos para que fuera más fácil analizarlos y poder extraer conclusiones respecto al rendimiento de PCBE. En este capítulo se encuentran algunos de los resultados obtenidos, así como un análisis detallado.

5.1. Resultados de los experimentos de comparación

Los primeros experimentos realizados fueron los enfocados a comparar el rendimiento de las diferentes políticas disponibles en PCBE. Esto tenía como objetivo, entre otros, determinar cuál era la mejor combinación de políticas para cada uno de los algoritmos (tanto el que optimiza energía como el que optimiza EDP). En la figura 5.1 se encuentran los resultados de las pruebas del algoritmo de energía, mientras que en la figura 5.2 se encuentran los del algoritmo de EDP. Cada una de estas figuras está acompañada por una tabla que contiene los resultados numéricos, las tablas 5.1 y 5.2.

Resultados de energía

Prueba	Energy short to eff (S_l)	Energy short to pow (S_h)	Energy long to eff (L_l)	Energy long to pow (L_h)	FCFS	SJF
Energía #1	197.11 J	189.58 J	195.27 J	187.86 J	187.86 J	189.57 J
Energía #2	209.6 J	207.45 J	201.02 J	210.24 J	209.17 J	207.45 J
Energía #3	262.94 J	250.46 J	262.94 J	256.27 J	256.27 J	254.98 J
Energía #4	269.52 J	293.70 J	289.20 J	289.16 J	291.86 J	293.70 J

Tabla 5.1: Tabla con los resultados de las diferentes pruebas de consumo energético. En este contexto, *short* se refiere a la priorización de los trabajos con el menor consumo estimado, mientras que *long* se refiere a lo contrario. Por otro lado, *pow* se refiere a la priorización de los nodos con mayor consumo energético, y por lo tanto *eff* hace referencia a lo contrario. En verde está resaltado el mejor resultado de cada fila.

Como se puede ver (en la figura 5.1 y la tabla 5.1), en cada una de las pruebas de energía una de las políticas tiene menor consumo que el resto. Esto es debido a que las pruebas se diseñaron para elaborar situaciones que favorecieran cada una de las políticas. Sin embargo, en las pruebas de energía números 1 y 3 se puede ver cómo la combinación de políticas que manda los trabajos largos a los nodos con mayor consumo (L_h) tiene el mismo rendimiento que el algoritmo FCFS, quedándose muy cerca de su rendimiento en las pruebas 2 y 4 (alrededor de un 1 % de diferencia del consumo).

Por otro lado, la combinación de políticas que manda los trabajos más cortos al nodo de menor consumo (S_l) obtiene el mejor resultado respecto al resto en la prueba de energía 4, con una reducción del consumo del 6 % frente a la segunda mejor planificación. Seguido de esta buena puntuación, se encuentra la combinación que manda los trabajos más largos (L_l) al nodo más eficiente en la prueba número 2, con una reducción del 3 % respecto del segundo mejor.

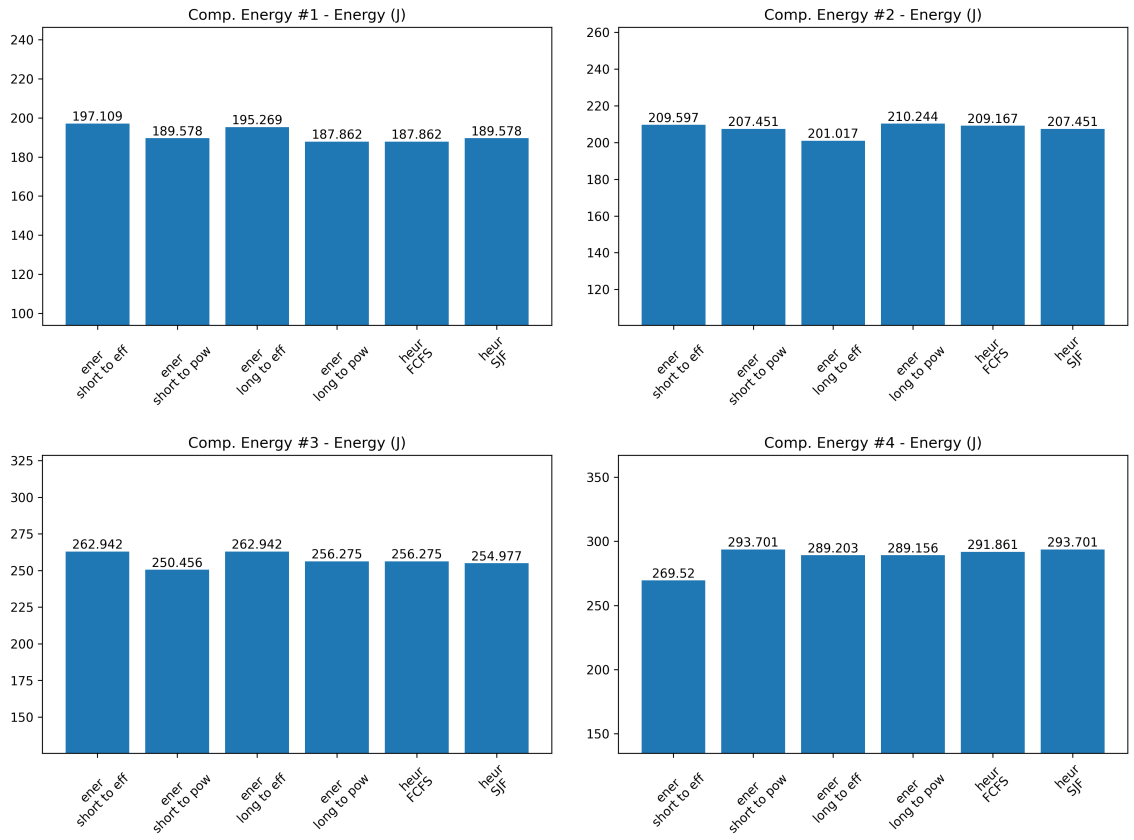

Figura 5.1: Gráfico de barras que ilustra los resultados de las diferentes pruebas de consumo energético. En este contexto, *short* se refiere a la priorización de los trabajos con el menor consumo estimado, mientras que *long* se refiere a lo contrario. Por otro lado, *pow* se refiere a la priorización de los nodos con menor consumo energético, y por lo tanto *eff* hace referencia a lo contrario.

Además de todo esto, también se compararon los gráficos de EDP y tiempo para cada combinación y prueba. Tras analizarlos, se determinó que la combinación de políticas que mejores resultados obtenía era L_l. Esta combinación obtenía los mejores resultados para estos otros 2 apartados en 3 de las 4 pruebas realizadas (a veces ganando al segundo mejor por hasta 5 %). Esto demuestra que, aún para situaciones en las que no obtiene el menor consumo energético, esta combinación puede tener beneficios al usarla de manera general. Cabe destacar también que, durante el diseño de las pruebas, fue necesario modificar alguno de los workloads, ya que, aún para situaciones que en teoría beneficiaban a otra política, ésta seguía saliendo ganadora, dificultando el diseño de los experimentos.

Esto tiene sentido, ya que mandar los trabajos que se cree que más van a consumir al nodo en el que se estima que su consumo pueda ser menor (mientras que el resto de trabajos se encajan donde pueda) es una estrategia que puede tener ventajas en términos de consumo energético, si bien puede tener efectos negativos en la eficiencia energética del sistema.

Resultados de eficiencia energética

Por otro lado, como se puede ver en la figura 5.2 y en la tabla 5.2, los resultados obtenidos en las pruebas del algoritmo que optimiza el EDP de un clúster son similares. Esta vez, la política que manda los trabajos con mayor eficiencia al nodo más eficiente (S_l) es la que obtiene un resultado idéntico al de SJF en la prueba diseñada para beneficiarla.

Al igual que en la sección anterior, la política S_l es la que obtiene un mejor delta respecto al segundo resultado en su prueba, siendo esta vez del 7 % (en la número 4). Asimismo, la política que asigna los

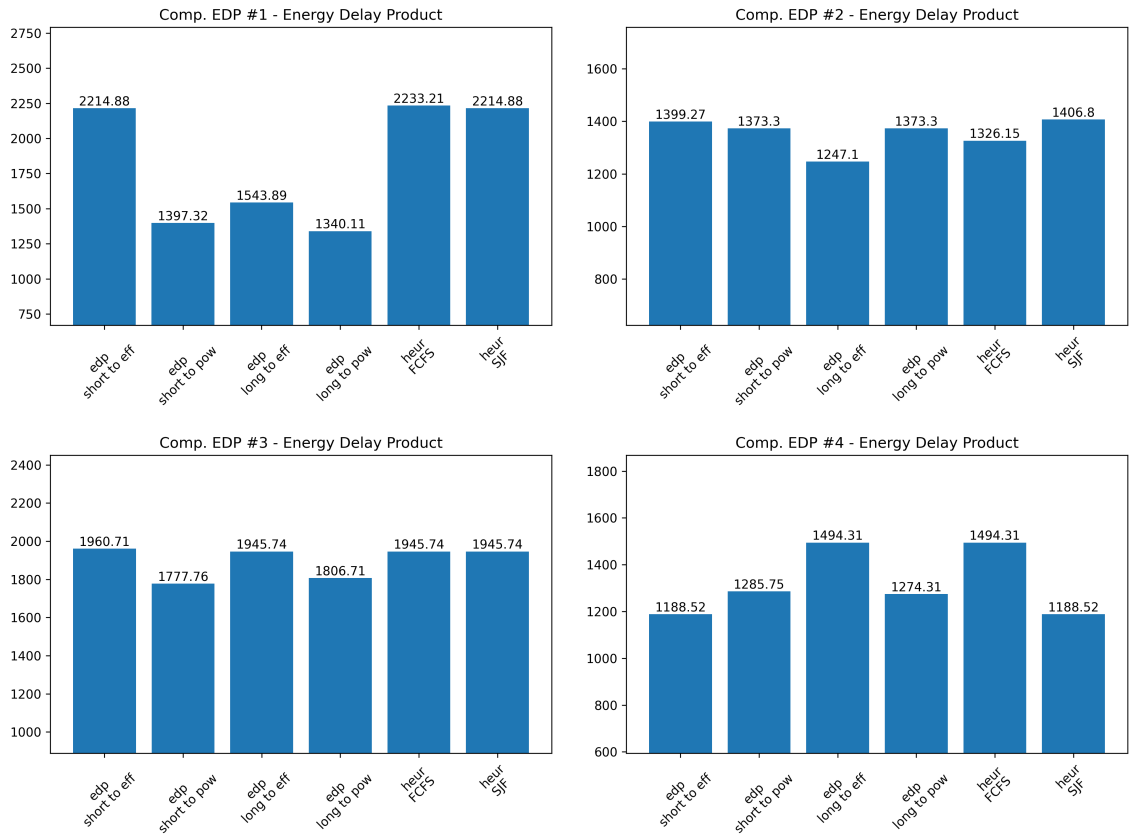

Figura 5.2: Gráfico de barras ilustrando los resultados de las diferentes pruebas de eficiencia energética. En este contexto, *short* se refiere a la priorización de los trabajos con la mejor eficiencia estimada, mientras que *long* se refiere a lo contrario. Por otro lado, *pow* se refiere a la priorización de los nodos con menor eficiencia energética, y por lo tanto *eff* hace referencia a lo contrario.

Prueba	EDP short to eff (S_l)	EDP short to pow (S_h)	EDP long to eff (L_l)	EDP long to pow (L_h)	FCFS	SJF
EDP #1	2214.88	1397.31	1543.890	1340.11	2233.21	2214.88
EDP #2	1399.27	1373.30	1247.09	1373.30	1326.15	1406.80
EDP #3	1960.70	1777.75	1945.74	1806.71	1945.74	1945.74
EDP #4	1188.52	1285.75	1494.31	1274.31	1494.31	1188.52

Tabla 5.2: Tabla con los resultados de las diferentes pruebas de eficiencia energética. En este contexto, *short* se refiere a la priorización de los trabajos con la mejor eficiencia estimada, mientras que *long* se refiere a lo contrario. Por otro lado, *pow* se refiere a la priorización de los nodos con mejor eficiencia energética, y por lo tanto *eff* hace referencia a lo contrario. En verde está resaltado el mejor resultado de cada fila.

trabajos con menor eficiencia energética a los nodos más eficientes (L_l) vuelve a quedar en segundo lugar, esta vez con una mejoría del 6 % (en la número 2). Cabe destacar también, que en la prueba número 1 se encuentran los resultados con mayor variación.

Por último, la misma configuración de políticas que en la sección anterior, L_l, obtiene resultados favorables en los otros apartados de algunas de las otras pruebas (aunque esta vez en menor medida); sacando mejores resultados en energía en 2 de ellas, y obteniendo el menor tiempo total en una prueba ajena a la suya. De nuevo, durante el diseño de las pruebas, esta combinación salía favorecida en la mayoría de ocasiones, si bien esta vez en menor medida, pudiendo encontrar con más facilidad escenarios que favoreciesen al resto de combinaciones de políticas.

De nuevo, esto demuestra que, incluso a la hora de optimizar por eficiencia energética, enviar los trabajos con peor eficiencia a los nodos más eficientes (L_l) es la mejor estrategia, pudiendo obtener

beneficios no solo en términos de eficiencia, sino también en términos de energía e incluso tiempo de ejecución.

5.2. Análisis de la configuración elegida

Antes de pasar a explicar los experimentos de escalado, es necesario hacer un análisis más detallado de algunas de las pruebas, ya que sus resultados son interesantes y pueden ayudar a caracterizar mejor el comportamiento del algoritmo. En concreto, se presenta aquí un análisis de las pruebas de energía y EDP número 2, así como la prueba de energía número 4.

Prueba de energía #2

Como ya se explicó en la sección anterior, en la prueba número dos, el planificador que optimiza energía priorizando los trabajos de mayor consumo y los nodos de menor consumo es el que obtiene mejores resultados. Sin embargo, no sólo los obtiene en cuanto a energía, sino que también encuentra la planificación que tarda menos tiempo, y por lo tanto tiene mejor eficiencia energética (como se puede ver en la figura 5.3). Cabe destacar también que este planificador obtiene además resultados dramáticamente mejores que los algoritmos FCFS y SJF en términos de tiempo y de eficiencia energética, lo cual tiene sentido, ya que esta última variable depende directamente de las otras dos.

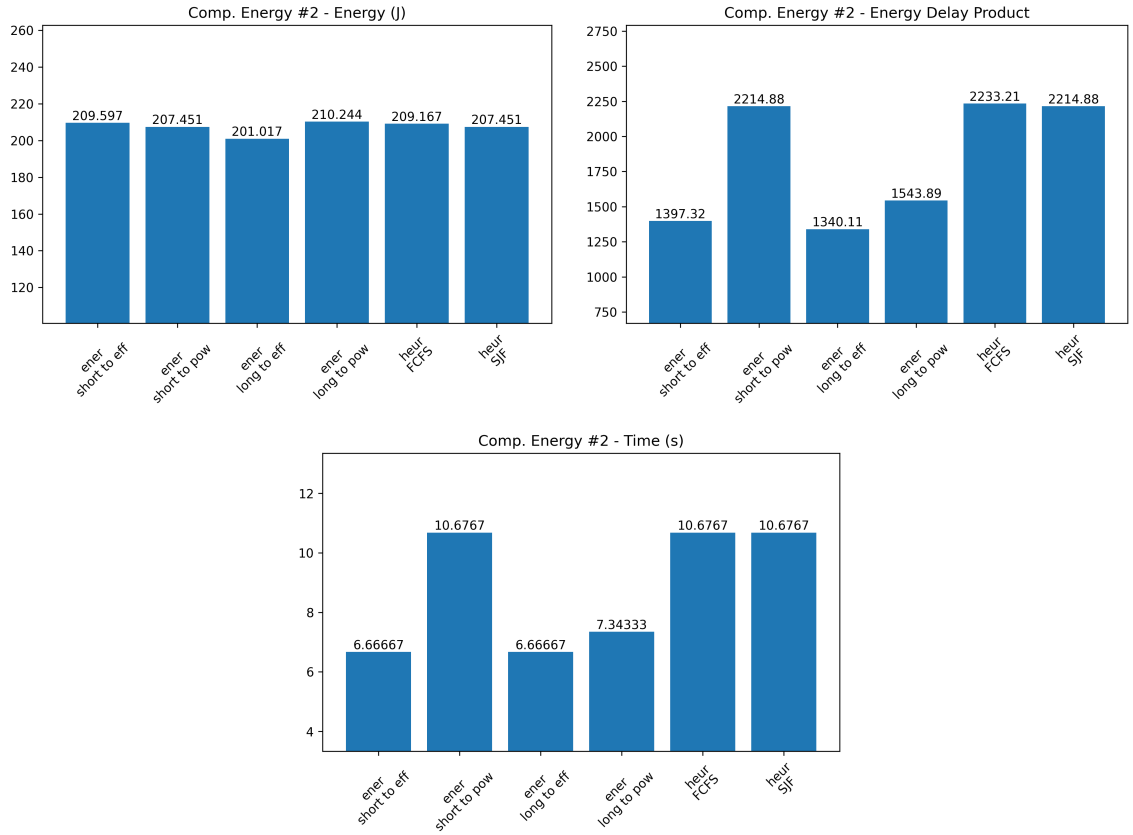

Figura 5.3: Gráficos con las métricas de energía (sup. izq), eficiencia (sup. der) y tiempo(inf), obtenidas en la prueba del planificador de energía número 2. En este contexto, *short* se refiere a la priorización de los trabajos con el menor consumo estimado, mientras que *long* se refiere a lo contrario. Por otro lado, *pow* se refiere a la priorización de los nodos con mayor consumo energético, y por lo tanto *eff* hace referencia a lo contrario.

De la misma manera, si se estudia la gráfica de asignación de trabajos para este planificador en la prueba 2, podemos ver cuál ha sido la asignación por la que ha obtenido tan buenos resultados. Como

se puede ver en la figura 5.4, el planificador asigna al nodo 1 (que es el de menor potencia) al trabajo 1 (que es de alto consumo energético), de esta manera reduce el consumo de este trabajo y por lo tanto el total de energía usada por el clúster.

Por otro lado, los trabajos que incurren en menos consumo energético son asignados al nodo 0 (el de mayor potencia de cómputo), de esta manera, el tiempo que dedica el clúster para atenderlos se reduce, lo que tiene un efecto positivo también en la eficiencia del sistema. Por último, como se puede ver, en ningún momento el clúster deja recursos sin utilizar (salvo cuando la cola queda vacía y terminan de llegar trabajos) lo que ayuda también a reducir el tiempo de espera de los trabajos y la eficiencia energética del clúster.

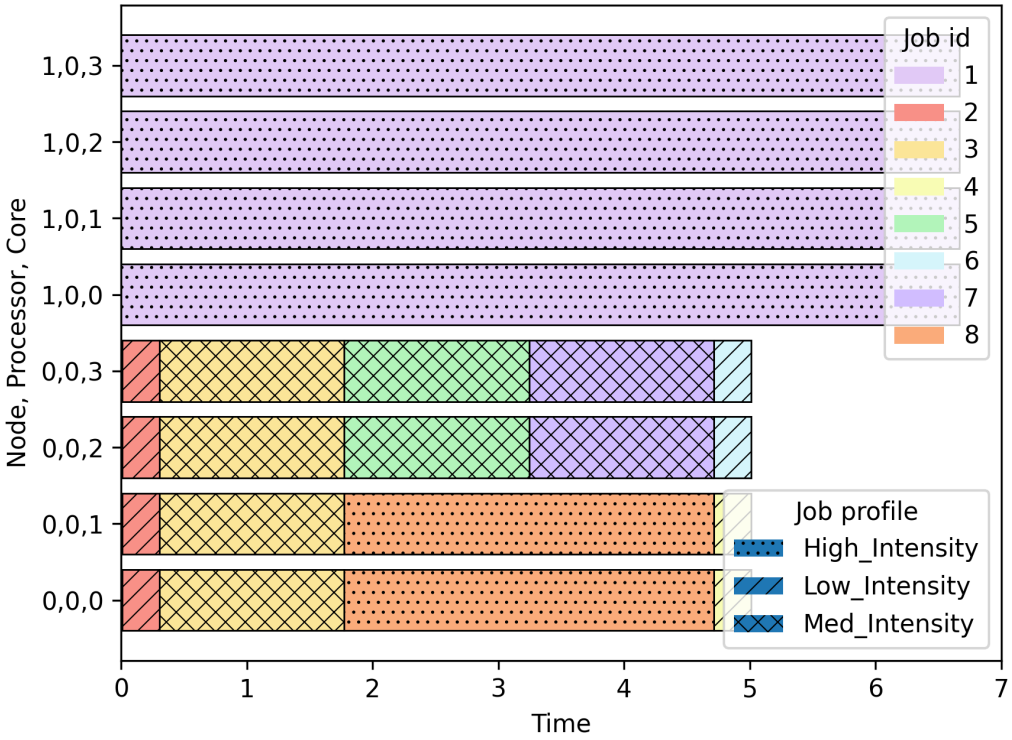

Figura 5.4: Gráfica que ilustra las asignaciones del planificador de energía configurado para priorizar los trabajos de mayor consumo y los nodos de menor consumo energético. En este caso, los cores del nodo 1 ("1,0,x") pertenecen al procesador de menor potencia, mientras que los del nodo 0 ("0,0,x") pertenecen al nodo de mayor potencia.

Prueba de EDP #2

Al igual que con la prueba de energía número 2, en la prueba 2 de EDP, las políticas que envían los trabajos con menor eficiencia energética al nodo con mayor eficiencia resultan en una mejor eficiencia energética del clúster completo. Sin embargo, esta vez (como se puede ver en la figura 5.5), esta combinación no es la que obtiene el mejor consumo energético, aunque se queda cerca, siendo el segundo mejor resultado. Sin embargo, esta configuración obtiene un tiempo mejor que el resto de planificadores, siendo este resultado un 5 % más bajo que el siguiente mejor. En este caso, los resultados no son tan diferentes como en la prueba anterior, pero aún así demuestran que optimizar por eficiencia energética puede ser también beneficioso para otros aspectos del clúster, como su rendimiento.

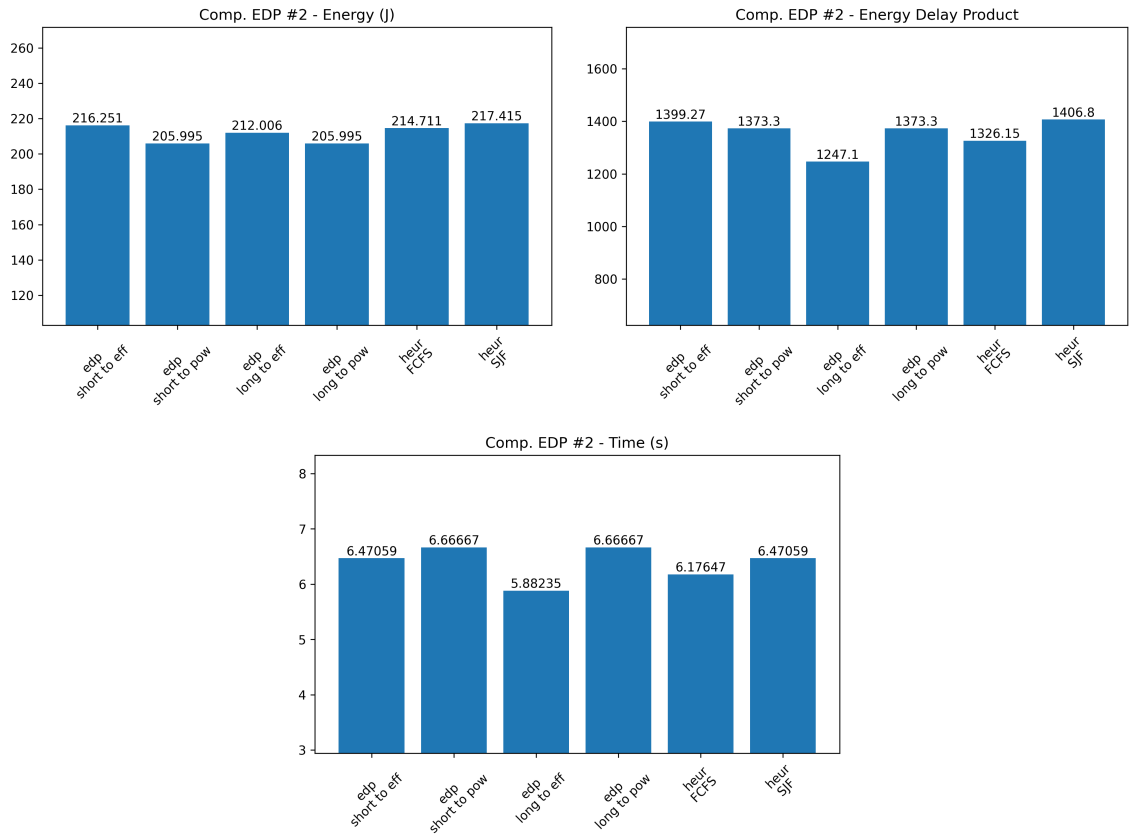

Figura 5.5: Gráficos con las métricas de energía (sup. izq), eficiencia (sup. der) y tiempo(inf), obtenidas en la prueba del planificador de eficiencia número 2. En este contexto, *short* se refiere a la priorización de los trabajos con la mejor eficiencia estimada, mientras que *long* se refiere a lo contrario. Por otro lado, *pow* se refiere a la priorización de los nodos con menor eficiencia energética, y por lo tanto *eff* hace referencia a lo contrario.

Por otro lado, estudiando la asignación de trabajos, ilustrada en la figura 5.6, podemos ver como el planificador asigna al nodo con más potencia de cómputo (el nodo 0) la realización de los trabajos más pesados (los de alta intensidad), mientras que delega al nodo menos potente la realización de los trabajos más livianos.

En este caso, esta decisión es la mejor, ya que asignar los trabajos más pesados al nodo potente reducirá el tiempo que el clúster pasa dedicando a los trabajos largos, por lo tanto la eficiencia energética mejorará, a costa del consumo. Esto es lo que se puede ver en las gráficas de la figura 5.5.

Prueba de energía #4

Por último, la prueba del planificador de energía 4 muestra como, a pesar de ser la que mejor consumo energético obtiene, la política que envía los trabajos de menor consumo a los nodos de menor consumo obtiene resultados significativamente peores en las otras métricas utilizadas. Como se puede ver en la figura 5.7, esta configuración tiene el peor tiempo y por tanto peor eficiencia, de todos los algoritmos.

Esto se debe a que, al intentar enviar los trabajos que menos consumen al nodo con menor consumo (que generalmente suele ser el de menor potencia), su tiempo de ejecución aumenta, reduciendo así el rendimiento del clúster. Por otro lado, estos trabajos pequeños ocupan los nodos de menor consumo, con lo que solo dejan libres los nodos que mayor consumo tienen para atender el resto de trabajos, lo que resulta en un mayor consumo, general del clúster. Como se puede ver, mientras que esta configuración consigue el objetivo de reducir el consumo energético del clúster, lo hace a cambio de sacrificar

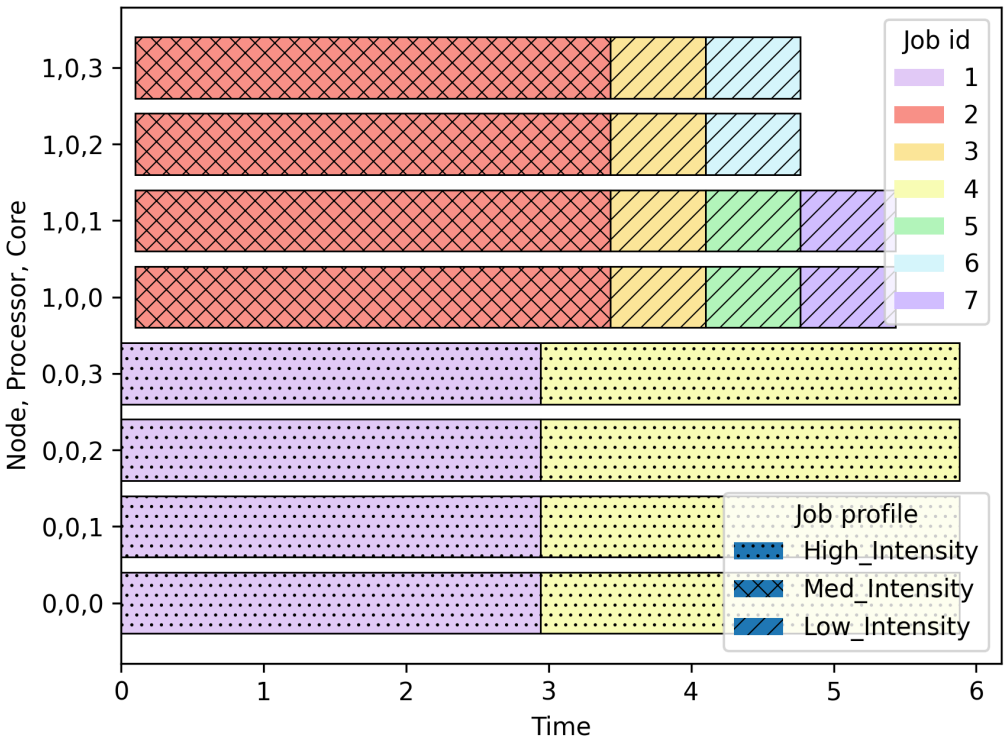

Figura 5.6: Gráfica que ilustra las asignaciones del planificador de eficiencia, configurado para priorizar los trabajos de mayor consumo y los nodos de menor consumo energético. En este caso, los cores del nodo 1 ("1,0,x") pertenecen al procesador de menor potencia, mientras que los del nodo 0 ("0,0,x") pertenecen al nodo de mayor potencia. Cabe destacar que el pequeño espacio de tiempo al principio de la ejecución en el que el nodo 1 no trabaja se debe a cómo llegan los trabajos a la cola, llegando el trabajo 2 0.1 unidades de tiempo más tarde que el 1.

rendimiento y por lo tanto eficiencia, y esto puede no ser siempre deseable.

5.3. Resultados de los experimentos de escalado

Tras terminar los experimentos comparativos y decidir cuál era la mejor combinación de políticas para los planificadores de energía y eficiencia energética, se pasó a ejecutar los experimentos de escalado, usando la metodología descrita en el capítulo 4. A continuación se muestran los resultados obtenidos de las pruebas, empezando por las que hacen uso de la traza de trabajos completa.

5.3.1. Resultados de la ejecución completa

La primera de las pruebas ejecutadas fue la que hace uso de la traza de trabajos sin segmentar. Con esto, se pretendía estudiar si los planificadores desarrollados obtenían ganancias respecto a otras técnicas heurísticas para situaciones más realistas, usando espacios del problema más grandes.

Como se puede ver en la figura 5.8 y la tabla 5.3, los planificadores diseñados obtienen unos resultados sustancialmente mejores que los algoritmos heurísticos en todas las áreas estudiadas: energía, tiempo y eficiencia. Ambos algoritmos obtienen unos resultados un 7 % mejores que FCFS, y un 11 % mejores que SJF (y el resto de los planificadores heurísticos) en cuanto a energía. Por otro lado, en términos de tiempo, los planificadores desarrollados obtienen un 11 % de speedup respecto a FCFS y un 16 %

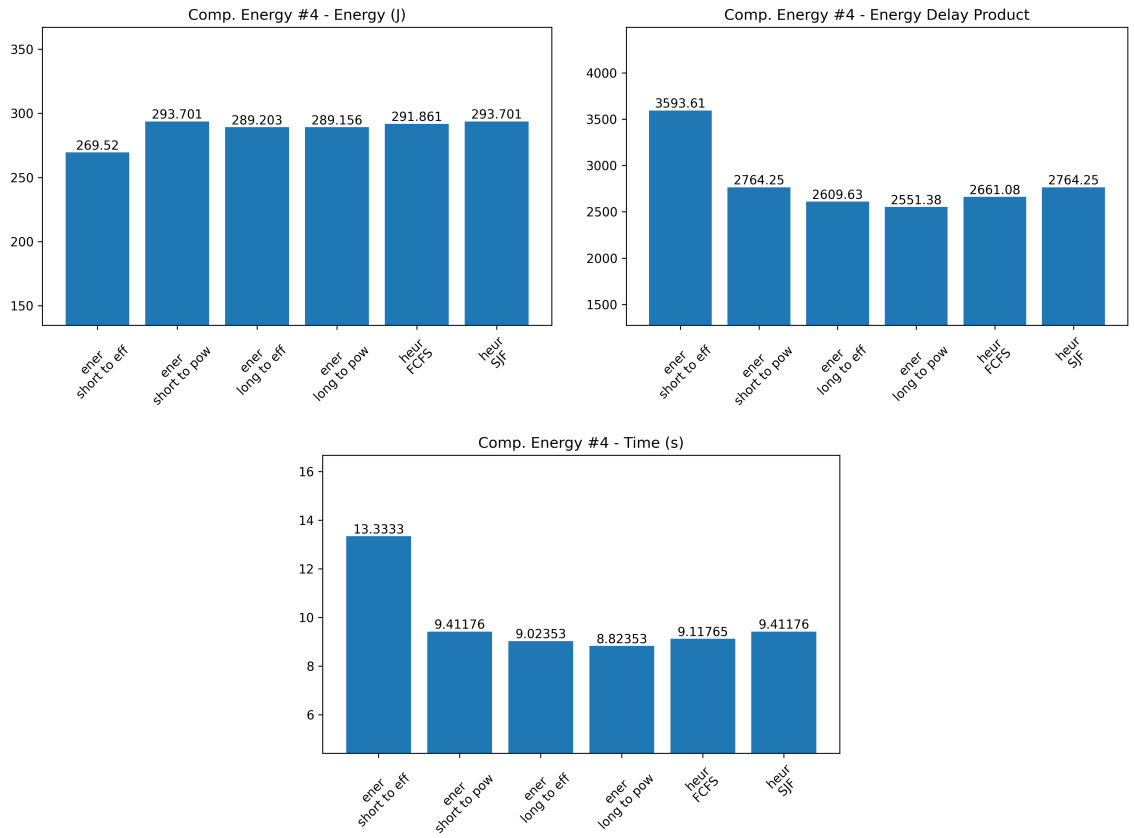

Figura 5.7: Gráficos con las métricas de energía (sup. izq), eficiencia (sup. der) y tiempo(inf), obtenidas en la prueba del planificador de energía número 4. En este contexto, *short* se refiere a la priorización de los trabajos con el menor consumo estimado, mientras que *long* se refiere a lo contrario. Por otro lado, *pow* se refiere a la priorización de los nodos con menor consumo energético, y por lo tanto *eff* hace referencia a lo contrario. En verde está resaltado el mejor resultado de cada columna.

respecto a SJF. De la misma manera, en cuanto a eficiencia energética, se obtienen unas ganancias del 16 % respecto a FCFS y del 25 % respecto a SJF. Cabe destacar, además que el planificador de energía es el que obtiene los mejores resultados, aunque el delta de rendimiento entre ambos planificadores es de menos del 1 %.

Métrica	FCFS	SJF	Heur. Short to IPC	Heur. Short to Low Power	Energy	EDP
Energía (J)	$1{,}61 \cdot 10^9$	$1{,}69 \cdot 10^9$	$1{,}70 \cdot 10^9$	$1{,}68 \cdot 10^9$	$1{,}5057 \cdot 10^9$	$1{,}5064 \cdot 10^9$
Tiempo (s)	$2{,}35 \cdot 10^6$	$2{,}51 \cdot 10^6$	$2{,}39 \cdot 10^6$	$2{,}50 \cdot 10^6$	$2{,}108 \cdot 10^6$	$2{,}111 \cdot 10^6$
EDP	$3{,}78 \cdot 10^{15}$	$4{,}25 \cdot 10^{15}$	$4{,}07 \cdot 10^{15}$	$4{,}21 \cdot 10^{15}$	$3{,}175 \cdot 10^{15}$	$3{,}181 \cdot 10^{15}$

Tabla 5.3: Tabla ilustrando los resultados obtenidos en cada una de las áreas (energía, tiempo, EDP) del experimento de escalado. La columna FCFS se corresponde con la barra de "heur first to first" en las gráficas de la figura 5.8, la columna SJF se corresponde con la barra "heur short to first" en la misma figura y "Heur. Short to IPC" se corresponde con la barra "heur short to flops". En verde se representan los mejores resultados de cada una de las áreas.

Estos resultados son sorprendentes, ya que ambos planificadores ganan a los algoritmos tradicionales hasta para métricas de tiempo, que PCBE no tiene en cuenta al realizar las planificaciones. PCBE$_n$ ha obtenido los mejores resultados no solo de energía, sino también de eficiencia (algo que correspondería más al planificador encargado de optimizar el EDP del clúster). Para poder entender mejor resultados, es necesario analizar los que se obtuvieron de la siguiente prueba.

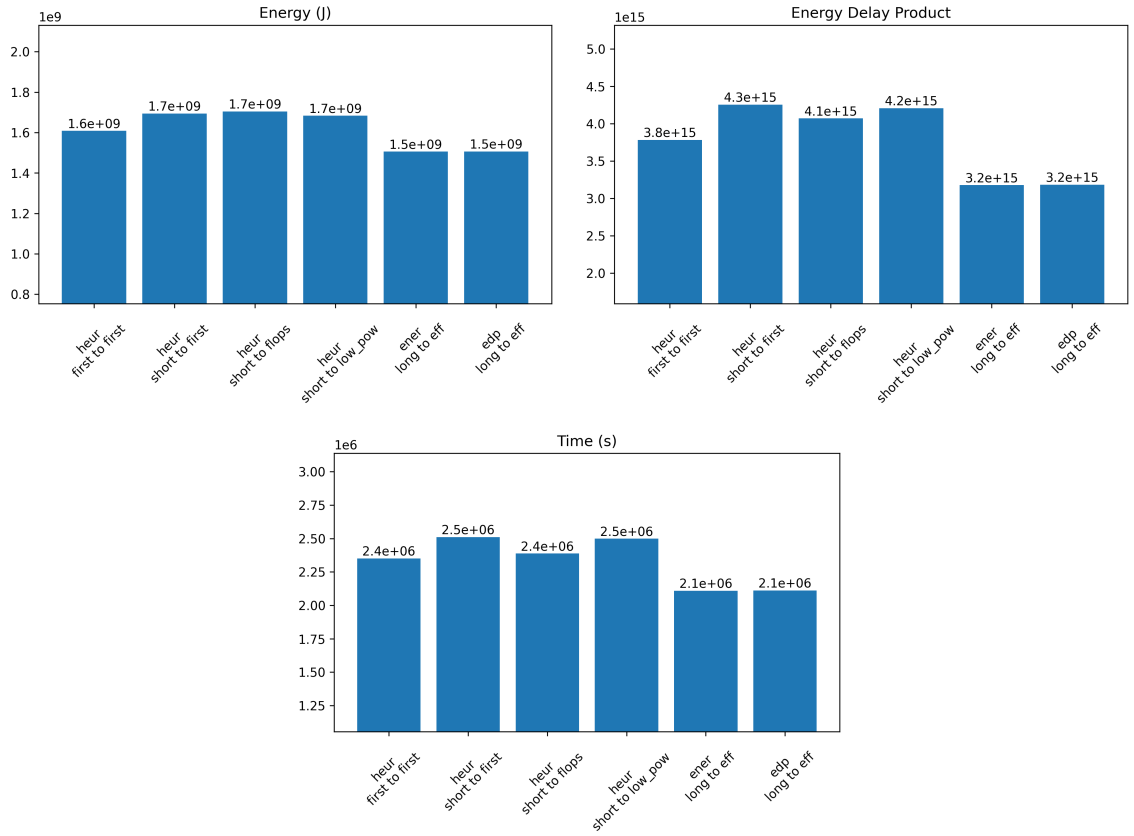

Figura 5.8: Gráficos con las métricas de energía (sup. izq), eficiencia (sup. der) y tiempo(inf), obtenidas en el experimentos de escalado. En este contexto, *short* se refiere a la priorización de los trabajos con el menor consumo estimado, mientras que *long* se refiere a lo contrario. Por otro lado, *pow* se refiere a la priorización de los nodos con menor consumo energético, y por lo tanto *eff* hace referencia a lo contrario.

5.3.2. Resultados de la ejecución por segmentos

Los resultados obtenidos en la prueba de escalado descrita en la sección anterior muestran como PCBE es capaz de obtener mejores resultados que técnicas heurísticas como FCFS tanto en consumo energético como eficiencia y tiempo. Sin embargo, para poder comprender mejor el por qué de estos resultados es necesario analizar el comportamiento de los algoritmos en mayor detalle. Con este fin se ha realizado esta prueba, en la que la traza de trabajos usada para el experimento de escalado se ha partido en 20 trozos, o segmentos. A continuación se detallan los resultados de esta prueba.

Energía

La figura 5.9 muestra, como una nube de puntos, los consumos energéticos de los algoritmos para cada uno de los segmentos analizados. Como se puede ver, a excepción de una de las técnicas heurísticas, todos los planificadores tienen un consumo muy similar, en la mayoría de los segmentos. En los segmentos 3, 11 y 12, sin embargo, PCBE obtiene mejores resultados, llegando a reducir hasta un 17 % el consumo energético del clúster respecto de FCFS (el mejor planificador heurístico).

Tiempo

En la figura 5.10 se ve el tiempo consumido por el clúster para ejecutar las tareas de cada segmento utilizando los diferentes algoritmos. Esta vez, el tiempo utilizado por todos los planificadores es exactamente el mismo en todos los segmentos, salvo los números 11, 12 y 13. En éstos, el algoritmo PCBE reduce el tiempo alrededor de un 9 % respecto de FCFS (el mejor planificador heurístico).

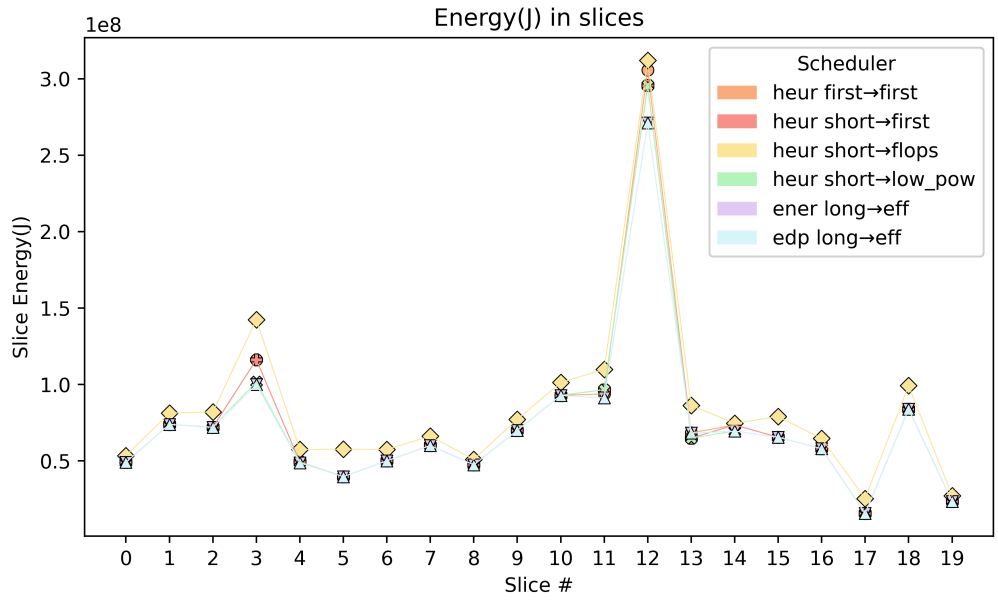

Figura 5.9: Consumo energético obtenido de cada uno de los algoritmos en cada uno de los segmentos de la traza.

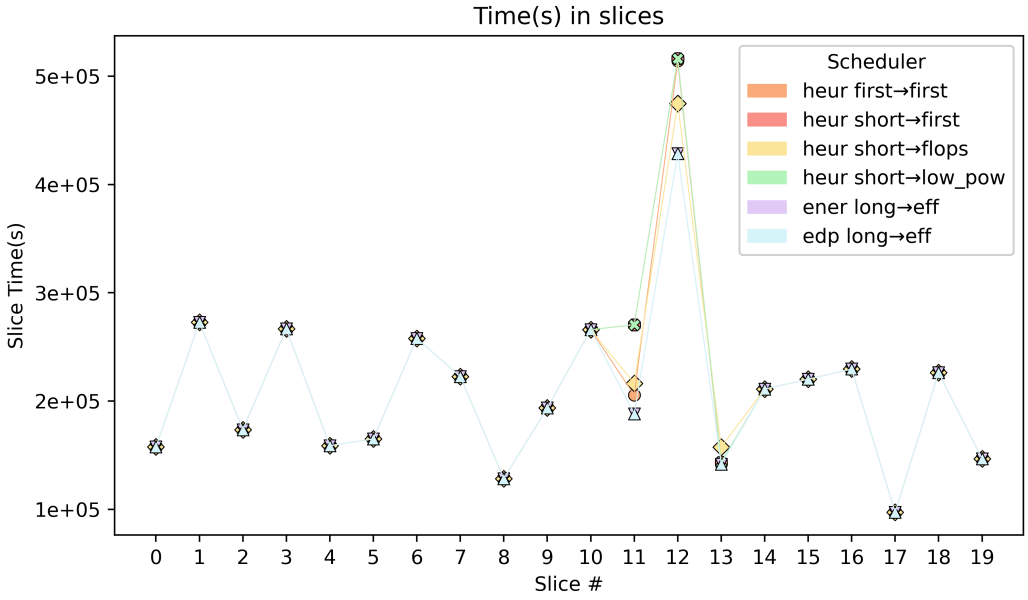

Figura 5.10: Tiempos obtenidos de los planificadores en cada uno de los segmentos de la traza.

Eficiencia energética

Por último, la figura 5.11 ilustra la eficiencia energética del clúster en cada segmento, usando los diferentes planificadores. Podemos observar como, al igual que en las figuras de tiempo y energía, PCBE obtiene mejores resultados en los segmentos 3, 11, y 12; manteniendo una eficiencia similar en el resto de segmentos. En estos segmentos, PCBE obtiene una eficiencia energética alrededor de un 15 % mejor que el siguiente mejor planificador, FCFS.

La razón por la que PCBE obtiene mejores resultados que el resto en estos segmentos se debe a la composición de trabajos de éstos. Los segmentos 3 y 12 tienen un conteo de subtareas mayor al del

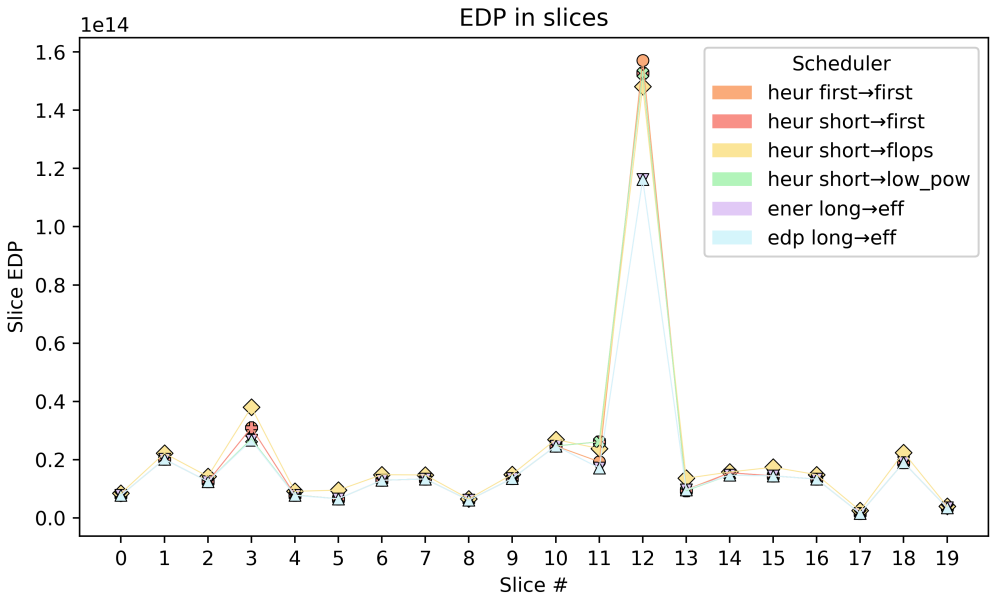

Figura 5.11: Eficiencia energética de los planificadores probados para cada uno de los segmentos de la traza.

resto de segmentos, exhibiendo una buena mezcla de trabajos con muchas subtareas (entre 30 y 64) y trabajos con pocas subtareas (menos de 10). Esto parece indicar que PCBE es capaz de asignar de manera más eficiente los trabajos, aprovechando los huecos que quedan entre trabajos "grandes" (con muchas subtareas) para asignar trabajos más "pequeños".

Como se puede ver en los resultados obtenidos durante la realización de estos experimentos, PCBE obtiene (en la mayoría de situaciones) mejores resultados que los planificadores heurísticos más comúnmente usados: FCFS y SJF [37, 27, 14], a veces superándolos, en las diferentes áreas estudiadas, hasta un 15 %. Por otro lado, aunque PCBE permite optimizar tanto consumo energético como eficiencia energética, parece que ambas configuraciones obtienen resultados muy similares (sino exactamente los mismos), al menos para las pruebas realizadas durante este proyecto.

Capítulo 6

Conclusiones

6.1. Conclusiones

El consumo energético de un clúster HPC puede tener un impacto económico y ambiental importante, por lo que es importante asegurar que se hace una buena utilización de los recursos, para así poder optimizar la energía consumida. Una de las maneras de obtener mejorías en el consumo de los sistemas HPC es mediante el uso de técnicas de planificación de trabajos más avanzadas, que tengan en cuenta la energía como parámetro a optimizar.

En este trabajo se ha presentado el diseño de PCBE, un planificador configurable basado en energía y eficiencia energética, capaz de asignar trabajos a los recursos de un clúster teniendo en cuenta su consumo o eficiencia energética. Además, se ha implementado una primera versión utilizando como plataforma el simulador IRMASim. Por último, además de implementarlo se ha realizado una evaluación en profundidad del mismo, comparándolo con algoritmos de planificación más tradicionales, como FCFS y SJF.

Como se puede ver por los resultados extraídos de los experimentos realizados en el capítulo anterior, se han alcanzado los objetivos definidos al principio del proyecto. Por un lado, se han implementado dos versiones de PCBE: una basada en energía (PCBE_n) y otra basada en eficiencia energética (PCBE_f). Se han implementado dos versiones (que optimizan dos métricas diferentes) debido a que el consumo y la eficiencia energética son objetivos que a veces pueden estar enfrentados. Realizar un consumo muy pequeño durante un tiempo muy largo puede no ser ideal, ya que se pueden estar desperdiciando recursos del clúster. PCBE (con las configuraciones probadas7 finalmente) consigue resultados mejores que algoritmos de planificación tradicionales, como FCFS y SJF. PCBE obtiene ganancias de alrededor del 15 % respecto a éstos, manteniendo el mismo rendimiento en los peores casos.

Asimismo, se ha podido implementar un mecanismo en el planificador que previene la *starvation* de trabajos que no cumplen los requisitos energéticos impuestos por PCBE. Este mecanismo, se basa en técnicas de envejeciemiento de los trabajos en la cola, de forma que prioriza aquellos trabajos que llevan más de un tiempo umbral esperando en la cola a ser planificados, asegurando de esta manera que el rendimiento del clúster no se vea perjudicado para ciertos perfiles de trabajos.

Por último, se ha conseguido implementar PCBE de manera que su complejidad temporal no sobrepase el umbral de $O(n^2)$, manteniendo por ello un tiempo de ejecución razonable para las colas de trabajos probadas. Esto ha ayudado a que el tiempo de ejecución del simulador en el que se ejecuta, IRMASim, no se vea afectado por el uso de este nuevo algoritmo.

Gracias a los experimentos realizados se han podido obtener ciertas conclusiones en cuanto a la planificación orientada a energía se refiere:

- No siempre es necesario sacrificar el rendimiento de un clúster para obtener un menor consumo energético; un algoritmo que sea consciente del consumo de cada componente del clúster puede hacer una mejor asignación de los trabajos que planificadores tradicionales, como FCFS. De esta manera, se asegura una utilización mejor de los recursos disponibles.
- Mientras que los algoritmos tradicionales, como FCFS y SJF, son capaces de realizar planificaciones casi óptimas para colas de trabajos pequeñas (con una complejidad temporal relativamente baja), éstos se dejan posibles ganancias no solo de consumo energético y eficiencia sino también de tiempo, para colas de trabajos más complejas en clústers más grandes y heterogéneos.
- Dependiendo de la composición de los trabajos que van llegando al clúster, la estrategia escogida para optimizar el consumo energético puede no ser la adecuada, resultando en un rendimiento inferior al deseable. Sin embargo, PCBE proporciona un mecanismo para seleccionar la estrategia a utilizar antes de empezar la ejecución, permitiendo así adaptarse mejor a diferentes situaciones si se conoce de antemano la naturaleza de los trabajos que se van a atender.

Como se puede ver, el desarrollo de PCBE se puede considerar un éxito, ya que, no solo consigue mejorar en todas las métricas a algoritmos de planificación tradicionales, sino que además ha ayudado a caracterizar mejor qué aspectos de la planificación basada en energía pueden no ser intuitivos, como que no siempre es necesario sacrificar rendimiento en el clúster para reducir su consumo energético.

6.2. Trabajo futuro

Aunque los resultados en este trabajo son prometedores, todavía quedan muchos aspectos en los que se puede profundizar en cuanto a PCBE se refiere. En primer lugar, sería deseable realizar más experimentos usando trazas reales en sistemas complejos, para poder analizar más el comportamiento del planificador en sus múltiples configuraciones. Asimismo, comparar PCBE con otras técnicas y algoritmos de planificación podría ayudar a caracterizar mejor el rendimiento de éste; enfrentándole quizás a planificadores más complejos basados en *reinforcement learning* (como el que proponen Fomperosa et al. [17]) o a otras técnicas heurísticas, como *Earliest Deadline First* (EDF) [21].

Por otro lado, dado que diferentes configuraciones parecen tener sus ventajas para ciertas situaciones, se puede modificar PCBE para que pueda decidir sobre la marcha cuál es la combinación de políticas que puede reportar las mayores ganancias de energía para cada momento dado. Además de esto, sería beneficioso realizar un estudio del mecanismo de prevención de la starvation implementado, determinando cuál puede ser el umbral óptimo (que asegura que se alcanza el mejor consumo energético mientras se asegura que ningún trabajo se vea discriminado). De esta manera, se podría incluso implementar un mecanismo que vaya ajustando dicho umbral de acuerdo a los perfiles de trabajo que esté atendiendo el clúster.

Por último, es necesario hacer un análisis en profundidad del tiempo que toma PCBE en realizar la planificación de los trabajos para diferentes situaciones, para localizar posibles cuellos de botella y poder de esta manera mejorar la complejidad temporal del algoritmo, minimizando de esta manera el tiempo que el clúster se pasa planificando los trabajos.

Además de posibles mejoras del algoritmo, también se podría extender IRMASim, cuyo modelo de consumo energético tiene ciertas limitaciones. Entre los posibles cambios estarían añadir soporte para una versión básica de DVFS, que permita probar cómo se comporta PCBE para nodos con potencias que fluctúan; o modelar la comunicación entre nodos, para así poder ejecutar trabajos con tareas en varios nodos.

Bibliografía

[1] Behnam Barzegar, Homayun Motameni y Ali Movaghar. «EATSDCD: A green energy-aware scheduling algorithm for parallel task-based application using clustering, duplication and DVFS technique in cloud datacenters». En: *Journal of Intelligent Fuzzy Systems* 36.6 (jun. de 2019), págs. 5135-5152. DOI: 10.3233/jifs-171927.

[2] Anne Benoit et al. «List and shelf schedules for independent parallel tasks to minimize the energy consumption with discrete or continuous speeds». En: *Journal of Parallel and Distributed Computing* 174 (abr. de 2023), págs. 100-117. DOI: 10.1016/j.jpdc.2022.12.003.

[3] José Luis Bosque et al. «A load index and load balancing algorithm for heterogeneous clusters». En: *J. Supercomput.* 65.3 (2013), págs. 1104-1113. DOI: 10.1007/s11227-013-0881-3. URL: https://doi.org/10.1007/s11227-013-0881-3.

[4] José Luis Bosque et al. «Parallel CBIR implementations with load balancing algorithms». En: *J. Parallel Distributed Comput.* 66.8 (2006), págs. 1062-1075. DOI: 10.1016/j.jpdc.2006.04.014. URL: https://doi.org/10.1016/j.jpdc.2006.04.014.

[5] P. Brucker. *Scheduling Algorithms*. Springer Berlin Heidelberg, 2013. ISBN: 9783662036129.

[6] Emilio Castillo et al. «Architectural Support for Task Dependence Management with Flexible Software Scheduling». En: *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 2018, págs. 283-295. DOI: 10.1109/HPCA.2018.00033. URL: https://doi.org/10.1109/HPCA.2018.00033.

[7] Alberto Cocaña-Fernández, Luciano Sánchez y José Ranilla. «Leveraging a predictive model of the workload for intelligent slot allocation schemes in energy-efficient HPC clusters». En: *Engineering Applications of Artificial Intelligence* 48 (feb. de 2016), págs. 95-105. DOI: 10.1016/j.engappai.2015.10.003.

[8] Alberto Cocaña-Fernández et al. «Eco-Efficient Resource Management in HPC Clusters through Computer Intelligence Techniques». En: *Energies* 12.11 (jun. de 2019), pág. 2129. DOI: 10.3390/en12112129.

[9] Omar Dakkak, Shahrudin Awang Nor y Suki Arif. «Scheduling through backfilling technique for HPC applications in grid computing environment». En: *2016 IEEE Conference on Open Systems (ICOS)*. 2016, págs. 30-35. DOI: 10.1109/ICOS.2016.7881984.

[10] M. Dorigo, V. Maniezzo y A. Colorni. «Ant system: optimization by a colony of cooperating agents». En: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 26.1 (1996), págs. 29-41. DOI: 10.1109/3477.484436.

[11] Briag Dupont, Nesryne Mejri y Georges Da Costa. «Energy-aware scheduling of malleable HPC applications using a Particle Swarm optimised greedy algorithm». En: *Sustainable Computing: Informatics and Systems* 28 (dic. de 2020), pág. 100447. DOI: 10.1016/j.suscom.2020.100447. URL: https://www.sciencedirect.com/science/article/pii/S2210537920301712.

[12] Eberhart y Yuhui Shi. «Particle swarm optimization: developments, applications and resources». En: *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*. Vol. 1. 2001, 81-86 vol. 1. DOI: 10.1109/CEC.2001.934374.

[13] Stijn Eyerman y Lieven Eeckhout. «Fine-Grained DVFS Using on-Chip Regulators». En: *ACM Trans. Archit. Code Optim.* 8.1 (feb. de 2011). ISSN: 1544-3566. DOI: 10.1145/1952998.1952999. URL: https://doi.org/10.1145/1952998.1952999.

[14] Yuping Fan. *Job Scheduling in High Performance Computing*. 2021. arXiv: 2109.09269 [cs.DC].

[15] Yuping Fan et al. «Deep Reinforcement Agent for Scheduling in HPC». En: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021, págs. 807-816. DOI: 10.1109/IPDPS49936.2021.00090.

[16] D. G. Feitelson y D. Tsafrir. *Logs of real parallel workloads from production systems*. Sep. de 2019. URL: https://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

[17] Jaime Fomperosa et al. «Task Scheduler for Heterogeneous Data Centres Based on Deep Reinforcement Learning». En: *Parallel Processing and Applied Mathematics*. Springer International Publishing, 2023, págs. 237-248. DOI: 10.1007/978-3-031-30442-2_18.

[18] Diego García-Saiz, Marta E. Zorrilla y José Luis Bosque. «A clustering-based knowledge discovery process for data centre infrastructure management». En: *J. Supercomput.* 73.1 (2017), págs. 215-226. DOI: 10.1007/s11227-016-1693-z. URL: https://doi.org/10.1007/s11227-016-1693-z.

[19] Adrian Herrera et al. «A Simulator for Intelligent Workload Managers in Heterogeneous Clusters». En: *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. Melbourne, Australia: IEEE, mayo de 2021, págs. 196-205. DOI: 10.1109/ccgrid51090.2021.00029. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9499578&isnumber=9499345.

[20] Grant Jenks. *Sorted Containers Documentation*. 2019. URL: https://grantjenks.com/docs/sortedcontainers/_modules/sortedcontainers/sortedlist.html.

[21] Bartłomiej Kocot, Paweł Czarnul y Jerzy Proficz. «Energy-Aware Scheduling for High-Performance Computing Systems: A Survey». En: *Energies* 16.2 (2023). ISSN: 1996-1073. DOI: 10.3390/en16020890. URL: https://www.mdpi.com/1996-1073/16/2/890.

[22] Baolin Li et al. «Sustainable HPC: Modeling, Characterization, and Implications of Carbon Footprint in Modern HPC Systems». En: *arXiv preprint arXiv:2306.13177* (2023).

[23] Antonio Llanes et al. «Soft computing techniques for the protein folding problem on high performance computing architectures». En: *Current drug targets* 17.14 (2016), págs. 1626-1648.

[24] Ioannis A. Moschakis y Helen D. Karatza. «Performance and cost evaluation of Gang Scheduling in a Cloud Computing system with job migrations and starvation handling». En: *2011 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, jun. de 2011. DOI: 10.1109/iscc.2011.5983873.

[25] Seyed Morteza Mirhoseini Nejad, Ghada Badawy y Douglas G. Down. «EAWA: Energy-Aware Workload Assignment in Data Centers». En: *2018 International Conference on High Performance Computing Simulation (HPCS)*. IEEE, jul. de 2018. DOI: 10.1109/hpcs.2018.00053.

[26] Raúl Nozal et al. «Load balancing in a heterogeneous world: CPU-Xeon Phi co-execution of data-parallel kernels». En: *J. Supercomput.* 75.3 (2019), págs. 1123-1136. DOI: 10.1007/s11227-018-2318-5. URL: https://doi.org/10.1007/s11227-018-2318-5.

[27] Michael Pinedo. *Scheduling*. Vol. 29. Springer, 2012.

[28] Yao Qin et al. «An energy-aware scheduling algorithm for budget-constrained scientific workflows based on multi-objective reinforcement learning». En: *The Journal of Supercomputing* 76.1 (oct. de 2019), págs. 455-480. DOI: 10.1007/s11227-019-03033-y.

[29] Valon Raca et al. «Runtime and energy constrained work scheduling for heterogeneous systems». En: *The Journal of Supercomputing* 78.15 (mayo de 2022), págs. 17150-17177. DOI: 10.1007/s11227-022-04556-7.

[30] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, 2010, págs. 125-129.

[31] C. H. Papadimitriou S. Dasgupta y U. V. Vazirani. *Algorithms*. 2006, págs. 89-96.

[32] C. H. Papadimitriou S. Dasgupta y U. V. Vazirani. *Algorithms*. 2006, págs. 133-160.

[33] Gilad Shainer et al. «Weather research and forecast (WRF) model performance and profiling analysis on advanced multi-core HPC clusters». En: *10th LCI ICHPCC* (2009).

[34] *Slurm Power Management Guide*. Ago. de 2023. URL: https://slurm.schedmd.com/power_mgmt.html.

[35] Esteban Stafford y José Luis Bosque. «Improving utilization of heterogeneous clusters». En: *J. Supercomput.* 76.11 (2020), págs. 8787-8800. DOI: 10.1007/s11227-020-03175-4. URL: https://doi.org/10.1007/s11227-020-03175-4.

[36] Esteban Stafford y José Luis Bosque. «Performance and energy task migration model for heterogeneous clusters». En: *J. Supercomput.* 77.9 (2021), págs. 10053-10064. DOI: 10.1007/s11227-021-03663-1. URL: https://doi.org/10.1007/s11227-021-03663-1.

[37] Andrew S Tanenbaum y Herbert Bos. *Modern operating systems*. Pearson, 2015.