UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería de Computadores

Trabajo Fin de Grado

Sistema Multiobjetivo para la Planificación de Flujos de Trabajo en Sistemas Multi-Nube

Autor: Aitor Del Álamo Martín

Fecha: 7-05-2019

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Grado en Ingeniería Informática en Ingeniería de Computadores

Trabajo Fin de Grado

Sistema Multiobjetivo para la Planificación de Flujos de Trabajo en Sistemas Multi-Nube

Autor: Aitor Del Álamo Martín

Tutor: José María Granado Criado

Tribunal Calificador

Presidente:

Secretario:

Vocal:

ÍNDICE DE CONTENIDOS

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

RESUMEN

En el siguiente TFG trataremos de resolver el problema de la asignación del flujo de trabajo de aplicaciones científicas en entornos multicloud usando optimización multiobjetivo. El entorno multicloud permite distribuir diferentes trabajos bajo demanda, permitiendo así un aumento de la velocidad y escalabilidad (Aslam & Islam, 15 December 2017). Gracias a ello se pueden ejecutar conjuntos de programas que buscan una solución común y obtener un resultado en un tiempo razonable.

Para resolverlo utilizaremos un algoritmo evolutivo multiobjetivo, más concretamente el algoritmo MOFA (algoritmo evolutivo multiobjetivo de las luciérnagas o *Multi-Objective Firefly Algorithm*). Tendremos un conjunto de luciérnagas que irán mejorando progresivamente para generar mejores soluciones. Posteriormente aplicaremos el algoritmo de ordenación no dominada *Fast Non-Dominated Sort* para obtener el frente óptimo, o lo que es lo mismo, el conjunto de soluciones no dominadas del algoritmo.

En sí, el enunciado presenta dos objetivos contrapuestos, el coste y el tiempo. Por tanto, es necesario obtener diferentes posibles soluciones al problema. Así nos encontraremos con soluciones con un coste aceptable pero un tiempo excesivo y viceversa.

El problema contiene diferentes tareas, que se tienen que repartir en varios clouds. El tiempo final de ejecución depende de dónde se han distribuido las tareas. Cada cloud tendrá tiempos de ejecución diferentes, variando además en función del tiempo de máquina usada en cada cloud. Además, cada tarea puede ser dependiente de otra, por lo que no es posible que se ejecute hasta que la predecesora termine. Para obtener el tiempo de ejecución de las tareas, tenemos que tener en cuenta diferentes factores, entre lo que podemos destacar el tiempo de transmisión de información entre tareas, si están en diferentes clouds o en el mismo, el tiempo que tarda la tarea en un cloud determinado, tiempo de espera de una tarea hasta que terminen sus sucesoras, etc.

Para evaluar el problema, partimos de 4 tipos de aplicaciones: Cybershark, Ligo, Montage y Sipht, que tienen diferente estructura interna y diferentes nodos terminales. El tiempo final se determina teniendo en cuenta aquellas tareas sin sucesoras, siendo éste el tiempo de finalización de la tarea sin sucesoras que más tarde en terminar.

1

INTRODUCCIÓN

El multicloud o multi-nube es un nuevo paradigma de utilización de recursos creado a partir de la computación en nube tradicional. Para utilizar este nuevo sistema es necesario tener varios proveedores del servicio, que otorgarán diferentes características y costes. Por suerte, los proveedores de multicloud tienen gran variedad de máquinas virtuales (VM) que podrán satisfacer los requerimientos de las aplicaciones (Sooezi, et al., 30 Nov.-3 Dec. 2015).

El flujo de trabajo es la mejor forma para representar aplicaciones y está extensamente utilizado en computación científica. Las tareas que son necesarias en los flujos de trabajo científico son de alta carga, es decir, es necesario un rendimiento alto para poder procesar las tareas y obtener el resultado en un tiempo aceptable.

Por consiguiente, es útil utilizar multicloud, aunque se puede generar un problema a la hora de adaptarlo a los diferentes clouds. En esto último es donde está el reto. El usuario tendrá un conjunto de tareas con dependencias, y mediante un algoritmo, determinará a qué cloud manda cada tarea para obtener el resultado con la mejor relación tiempo-coste.

En este TFG propondremos utilizar el algoritmo de las luciérnagas multiobjetivo (MOFA, *Multi-Objective Firefly Algorithm*) con el fin de optimizar la carga de trabajo y el coste con la restricción de la fiabilidad.

En relación al problema, consideramos un conjunto de tareas que tenemos que distribuir en diferentes clouds para que se ejecuten en el menor tiempo posible y con el menor coste de utilización de los clouds. Estamos ante un problema en el que calcular todas las posibles combinaciones requeriría un coste computacional demasiado alto, sobre todo al aumentar el número de tareas de las aplicaciones, por tanto, implementaremos un algoritmo que obtenga en un tiempo más reducido un resultado aceptable, sin llegar a calcular todas las soluciones. Esto vendría a decirnos que el problema es NP-Completo (Madni, 2016). Concretamente, el número de posibilidades de repartir *t* tareas entre *m* máquinas (estando estas máquinas repartidas entre varios clouds) sería de t^n. Por ejemplo, usando 700 tareas y 25 máquinas diferentes repartidas en tres clouds (uno de los experimentos que hemos realizado) se tendrían $1.34x10^{71}$ posibles soluciones.

Por otra parte, podemos decir que los objetivos del problema son contrapuestos, es decir, si mejora el tiempo de computo, podría subir el coste del problema, por tanto,

es necesario utilizar un algoritmo multiobjetivo para tener a ambos objetivos en consideración.

El algoritmo que utilizaremos (MOFA) es un algoritmo de inteligencia en enjambre. Estos algoritmos son capaces de empezar con un conjunto de soluciones válidas aleatorias e ir mejorándolas mediante unas directrices, hasta una condición de parada para obtener un resultado aceptable en un tiempo de computo aceptable.

El algoritmo de las luciérnagas multiobjetivo comenzará con un conjunto de soluciones, que se irán mejorando progresivamente en función de los dos objetivos. Como restricción para las soluciones del problema, tendremos la fiabilidad, que también dependerá de dónde se distribuyan las diferentes tareas.

Finalmente, una vez ejecutado el problema con el algoritmo multiobjetivo de las luciérnagas y un algoritmo de generación de soluciones aleatorias, llegamos a la conclusión de que obtenemos resultados mucho mejores con el algoritmo MOFA que con el aleatorio, aunque en algunos casos, la estructura interna de alguna aplicación hace que el algoritmo MOFA más igualado al aleatorio, sobre todo con pocas tareas. Además, cuanto mayor número de tareas utilizados para la ejecución, mejor es el resultado y mayor la diferencia entre las ejecuciones de aleatorio y MOFA.

Este TFG está ordenado de la siguiente manera: en la sección 2 explicaremos los objetivos del problema; seguidamente en la sección 3 explicaremos el problema junto con diferentes definiciones de algoritmos. En la sección 4 mostraremos la metodología utilizada para resolver el problema. Continuremos con los resultados obtenidos en la sección 5 y, por último, en la sección 6 presentaremos las conclusiones obtenidas de este trabajo.

2

OBJETIVOS

El objetivo principal de este trabajo es el de implementar un sistema de optimización multiobjetivo para la asignación de tareas de aplicaciones con grandes flujos de trabajo en sistemas multicloud. Este objetivo puede dividirse en los siguientes objetivos secundarios:

- Implementación del sistema de preprocesado de los datos: inicialmente los datos de las aplicaciones vienen dados como un conjunto de entradas en un fichero de texto, por lo que hay que adaptarlo a una estructura interna para poder conocer el orden de ejecución de las tareas, tiempos de ejecución, tamaños de las transferencias, …
- Implementación del algoritmo multiobjetivo de las luciérnagas (MOFA): Se realizará una adaptación multiobjetivo del algoritmo de las luciérnagas o FA (*Firefly Algorith)* ya que estamos tratando con un problema con dos objetivos.
- Ajuste paramétrico del algoritmo: debido a que el mecanismo de mejora de soluciones del algoritmo MOFA se basa en varios parámetros, hay que buscar la mejor combinación de los mismos, por lo que se requiere un estudio paramétrico.
- Experimentación usando diferentes tipos de aplicaciones, así como diferentes tamaños (en número de tareas) y valores de fiabilidad.

3

ANTECEDENTES / ESTADO DEL ARTE

Como se ha comentado anteriormente, en este trabajo vamos a resolver el problema de asignación de flujos de trabajo en sistemas multicloud. El problema se define mediante dos objetivos contrapuestos, el coste y el tiempo. Para poder resolverlo no podemos priorizar ninguno de los objetivos, ni obtener la mejor solución de cada uno de ellos ya que si disminuye el coste, aumenta el tiempo y viceversa. Es necesario tener en cuenta ambos objetivos para obtener soluciones válidas al problema.

Por tanto, para resolver este problema vamos a utilizar el algoritmo de las luciérnagas multiobjetivo (*MOFA*), donde los objetivos a tener en cuenta son el coste y el tiempo, ambos objetivos a minimizar, es decir, cuanto menor sea su valor, mejor será la solución. Aplicando este algoritmo, obtendremos un conjunto de soluciones Pareto óptimas, es decir, aquellas soluciones en las que no se puede mejorar un objetivo sin empeorar el otro.

A continuación, definiremos en detalle el problema y haremos una introducción a los algoritmos genéticos.

3.1 Definición del problema

El enunciado lo podríamos definir como problema de planificación de flujo de trabajo. En este problema trataremos el flujo de trabajo, modelo multicloud, el tiempo que tarda en terminar todas las tareas del problema, el coste computacional total, fiabilidad computacional y problemas de formulación.

El flujo de trabajo normalmente se representa mediante un grafo acíclico dirigido, en el cual cada nodo que compone el grafo sería una tarea y las aristas representan el orden de las mismas. Las primeras tareas que lo componen son tareas iniciales, y no tienen tareas predecesoras, mientras que las últimas tareas son las que no tienen sucesoras. Estas últimas determinarán el tiempo que tarda la aplicación en ejecutarse.

En nuestro caso, este conjunto de tareas tiene que ser repartido en varios clouds diferentes, con diferentes características en cuanto a la velocidad de procesamiento de sus componentes y el coste en función del tiempo que se está usando el cloud. Hay tareas, por ejemplo, que debido a su poca duración no es necesario ejecutarlas en un nodo del cloud con alto rendimiento, permitiendo a otras tareas ejecutarse en dichos nodos. Debido a que unas tareas son dependientes de otras, hasta que no

terminen todas sus tareas antecesoras no es posible ejecutar esas tareas. Además, es necesario que cada tarea tenga los datos de las tareas dependientes, por tanto, es necesario transferir dichos datos a la máquina donde se aloja la tarea, ya sea porque las tareas dependientes estuvieran en clouds distintos o en nodos distintos del mismo cloud. Además, no solo el tiempo de transferencia afecta al tiempo de ejecución, también el tiempo de espera requerido para enviar otros datos a otras tareas. Por ejemplo, si la tarea *x* debe enviar datos a las tareas *y* y *z*, el tiempo requerido para enviar los datos de *x* a *z* es igual al tiempo de envío de los datos (tiempo de transferencia) más el tiempo de envío de los datos de *x* a *y* (tiempo de espera), ya que *x* no puede enviar todos los datos a la vez.

Para realizar el cálculo del tiempo total que tarda el problema en terminar, es necesario llevar a cabo varias cuentas, que influyen directamente en el resultado final, al igual que dónde se ejecuten las tareas. Necesitamos calcular el tiempo de fin de una tarea, tiempo de espera de una tarea, tiempo de ejecución de una tarea, dependiendo de donde se ejecute, tiempo de recepción de los datos y tiempo de comienzo de la tarea. Una vez que se calculan todos los datos se puede obtener el *makespan*, que sería el tiempo total que tarda el problema en ejecutarse.

Antes de nada, mostraremos a continuación las definiciones de los elementos que se van a utilizar:

- T conjunto de todas las tareas de la aplicación
- t_i tarea i del flujo de tareas.
- $D(t_i, t_j)$ el tamaño de transmisión de datos de la tarea i a la tarea j.
- $W(t_i)$ carga de trabajo de la tarea t_i.
- $pre(t_i)$ tareas predecesoras de la tarea t_i.
- $succ(t_i)$ tareas sucesoras de la tarea t_i.
- n el número de tareas del flujo de trabajo.
- $VM(m)$ Conjunto de los tipos de máquinas que contiene la plataforma m.
- $VM(m,k)$ Tipo de la máquina k de la plataforma m.
- $P(m,k)$ la capacidad de procesamiento de la máquina *VM (m, k).*
- $C(m,k)$ coste por unidad de tiempo en la máquina *VM (m, k).*
- B_m ancho de banda de la plataforma m.
- $B_{mm'}$ ancho de banda entre las plataformas m y m'.

- $T_{trans}(t_i, t_j)$ tiempo de transferencia de la tarea t_i a la tarea t_j.
- $T_{send}(t_i)$ tiempo total de todas las transferencias que realiza la tarea t_i
- $T_{trans}(t_i)$ tiempo total de todas las transferencias que recibe la tarea t_i.
- $T_{exec}(t_i, VM(m,k))$ tiempo de ejecución de la tarea t_i en la máquina *VM (m, k)*.
- $T_{start}(t_i)$ tiempo de comienzo de la tarea t_i.
- $T_{end}(t_i)$ tiempo de finalización de la tarea t_i.
- $T_{rece}(t_i)$ tiempo de recepción de la tarea t_i.
- $T_{rent}(t_i, VM(m,k))$ tiempo de alquiler de la tarea t_i en la máquina *VM (m, k)*.
- $cost(t_i, VM(m,k))$ coste de alquiler de la tarea t_i en la máquina *VM (m, k)*.
- $rel(t_i)$ fiabilidad de la tarea t_i.
- $cost$ coste total de la carga de trabajo.
- $makespan$ tiempo total de trabajo.
- $reliability$ fiabilidad.
- rel_c restricción de fiabilidad de la carga de trabajo.

A continuación, vamos a ver la forma de calcular las dos funciones objetivo de nuestro algoritmo multiobjetivo.

3.1.1. **Cálculo de la función tiempo de ejecución**

Para calcular el tiempo de finalización de una tarea, utilizaremos la Ecuación (1), siendo la suma de los tiempos de comienzo, recepción y ejecución de dicha tarea.

$$T_{end}(t_i) = T_{start}(t_i) + T_{rece}(t_i) + T_{exec}\big(t_i, VM(m,k)\big) \quad (1)$$

Para calcular el tiempo de comienzo de una tarea, se utiliza la Ecuación (2). Como vemos, este valor depende de los tiempos de finalización de las tareas predecesoras, así como de los tiempos de espera (tiempo que transcurre desde que termina la tarea predecesora hasta que comienza la transferencia de datos).

$$T_{start}(t_i) = max_{t_j \in pre(t_i)}\{T_{end}\big(t_j\big) + T_{wait}(t_j, t_i)\} \quad (2)$$

Como hemos comentado anteriormente, el tiempo de espera de una tarea t_i frente a otra tarea predecesora t_j es aquel que transcurre desde que t_j comienza la transferencia de los datos a sus diferentes tareas sucesoras hasta que comienza la transferencia a la tarea t_i. Para obtener este tiempo, lo primero es obtener el conjunto

*A (*Ecuación *(3))*, que son todas las tareas sucesoras de las predecesoras de t_i. Este conjunto estará ordenado según su orden en el flujo de trabajo. A partir del conjunto *A*, obtenemos el subconjunto *B*, que son todas aquellas tareas de *A* cuyo orden está delante de la tarea t_i, es decir, el conjunto *B* contiene todas aquellas tareas a las que las predecesoras de t_i enviarán datos antes que a t_i. Por tanto, el tiempo de espera de la tarea t_i respecto a t_j es la suma de los tiempos de transferencia de la tarea t_j a todas sus sucesoras que van por delante de t_i, como se muestra en la Ecuación (4).

$$A = \bigcup_{t_j \in pre(t_i)} succ(t_j) \quad (3)$$

$$T_{wait}(t_j, t_i) = \sum_{t_z \in B} T_{trans}(t_j, t_z) \quad (4)$$

El tiempo de transmisión de los datos utilizados entre las tareas viene determinado por la Ecuación (5).

$$T_{trans}(t_j, t_i) = \begin{cases} \dfrac{D(t_j, t_i)}{B_m}, si\ t_i\ y\ t_j \text{ pertenecen al } mismo\ cloud \\ \dfrac{D(t_j, t_i)}{B_{mm'}}, si\ t_i y\ t_j \text{ pertenecen a } distintos\ clouds\ (m \neq m') \end{cases} \quad (5)$$

De esta forma (ecuaciones (2) a (5)) obtendríamos el tiempo de comienzo (T_{start}) de una tarea, primer elemento de la Ecuación (1). Pasemos a continuación a calcular el tiempo de recepción de una tarea (T_{rece}), segundo elemento de la Ecuación (1). Para obtener este valor usamos la Ecuación (6).

$$T_{rece}(t_i) = T_{trans}(t_i) - T_{start}(t_i), \quad (6)$$

donde $T_{trans}(t_i)$ es el tiempo total de transferencia de la tarea t_i y viene dado por la Ecuación (7).

$$T_{trans}(t_i) = max_{tj \in pre(ti)}\{T_{end}(tj) + T_{wait}(tj, t_i) + T_{trans}(tj, t_i)\} \quad (7)$$

Por último, calculamos el tercer elemento de la Ecuación (1), el tiempo de ejecución o T_{exec}. El tiempo de ejecución de cada tarea en los diferentes clouds varía

dependiendo de la velocidad de computo la máquina usada en dicho cloud. Para calcular este tiempo, tendremos en cuenta la carga de trabajo de la tarea y la capacidad de procesamiento de la máquina utilizada, tal y como se muestra en la Ecuación (8).

$$T_{exec}\left(t_i, VM(m,k)\right) = \frac{W(t_i)}{P(m,k)} \quad (8)$$

Finalmente, para calcular el tiempo total que tarda en ejecutarse la aplicación con una solución determinada, utilizaremos la Ecuación (9).

$$makespan = T_{end}(T_{exit}), \quad (9)$$

donde T_{exit} es la tarea final.

3.1.2. **Cálculo de la función coste**

Desde el punto de vista del coste, teniendo en cuenta el entorno multicloud, los diferentes clouds tienen diferentes sistemas de precios. Los algoritmos actuales solo toman el intervalo entre la hora de inicio y la hora de finalización de la tarea, pero la máquina virtual no se detiene hasta que se ha finalizado el envío de datos. La suma total final de los datos se obtiene del sumatorio de todas las tareas ejecutadas.

El tiempo de alquiler de cada tarea viene dado por la Ecuación (10).

$$T_{rent}\left(t_i, VM(m,k)\right) = T_{rece}(t_i) + T_{exec}\left(t_i, VM(m,k)\right) + T_{send}(t_i) \quad (10)$$

Una vez obtenido el tiempo de alquiler de una tarea, tenemos que calcular el coste que supone esa tarea, que dependerá del cloud utilizado. En nuestro caso, vamos a usar tres clouds diferentes que utilizan sistemas distintos para calcular el coste de utilización:

- Amazón EC2 (*m=1*). Utiliza un sistema de cobro por tramos de horas completas, con lo que cuesta lo mismo usar una máquina durante un minuto que durante 60. La Ecuación (11) muestra este sistema de cálculo del coste de alquiler.
- Microsoft Azure (*m=2*). Utiliza un sistema de cobro por tramos de minutos. La Ecuación (12) muestra este sistema de cálculo del coste de alquiler.

- Google Compute Engine (*m=3*). En este caso, se utiliza un sistema mixto en el que se usará un coste mínimo para cuando se use una máquina hasta 10 minutos y un coste proporcional al tiempo en minutos en caso contrario, tal y como puede apreciarse en la Ecuación (13).

$$cost\big(t_i, VM(1,k)\big) = \lceil T_{rent}(t_i, VM(1,k))/60 \rceil * C(1,k) \quad (11)$$

$$cost\big(t_i, VM(2,k)\big) = T_{rent}\big(t_i, VM(2,k)\big) * \frac{C(2,k)}{60} \quad (12)$$

$$cost\big(t_i, VM(3,k)\big) = \begin{cases} 10 * \frac{C(3,k)}{60}, si\ T_{rent}\big(t_i, VM(3,k)\big) \leq 10 \\ T_{rent}\big(t_i, VM(2,k)\big) * \frac{C(3,k)}{10}\ \ en\ caso\ contrario \end{cases} \quad (13)$$

A continuación, las Tablas 3.1-3.3 muestran los costes de utilización por hora de los tres clouds usados en este trabajo. Para cada cloud, las tablas muestran las diferentes máquinas disponibles (columna VM tipo) así como la capacidad de cómputo (columna Unidad de cómputo) y el coste por hora en dólares de dichas máquinas.

VM tipo	Unidad de computo	Coste por hora $
1	1,7	0,06
2	3,75	0,12
3	7,5	0,24
4	15	0,45
5	30	0,90

Tabla 3.1 Capacidad de cómputo y coste por hora de las diferentes máquinas del cloud Amazon EC2.

VM tipo	Unidad de computo	Coste por hora $
1	1	0,07
2	2	0,14
3	4	0,28
4	8	0,56
5	16	1,12

Tabla 3.2 Capacidad de cómputo y coste por hora de las diferentes máquinas del cloud Google Compute Engine.

VM tipo	Unidad de computo	Coste por hora $
1	1	0,05
2	2	0,099
3	4	0,2
4	8	0,401
5	16	0,802

Tabla 3.3 Capacidad de cómputo y coste por hora de las diferentes máquinas del cloud Microsoft Azure.

Por último y como hemos mencionado anteriormente, el cálculo del coste total viene dado por la Ecuación (14).

$$cost = \sum_{t_i \in T} cost(t_i, VM(m,k)) \quad (14)$$

3.1.3. **Restricción de fiabilidad**

El fallo es inevitable en un sistema distribuido de computación. Estos pueden provenir del propio sistema (averías del hardware, fallos del software o cortes de corriente) o del exterior (como, por ejemplo, ataques maliciosos), lo que puede provocar errores en la ejecución de los trabajos. La fiabilidad de un sistema viene dada por el porcentaje de que un trabajo sufra un fallo por unidad de tiempo, por lo que cuanto más tiempo de ejecución, mayor será la probabilidad de fallo.

Por tanto, para calcular la fiabilidad, es necesario tener en cuenta dónde se ejecuta la tarea, ya que los clouds tienen diferente coeficiente de fallo y sus máquinas diferentes capacidades de cómputo. Una vez que se calcula la fiabilidad, se determina si el resultado es o no válido. En el caso que nos ocupa, se ha utilizado la fiabilidad como una restricción, haciendo que todas aquellas soluciones que no cumplan el mínimo de fiabilidad establecido no son válidas y deben ser modificadas. La fiabilidad de una tarea se calcula utilizando la Ecuación (15), y viene dada en función de la fiabilidad de la máquina utilizada por la tarea y el tiempo de utilización de dicha máquina, mientras que la fiabilidad total de una solución se calcula multiplicando las fiabilidades de todas las tareas de la aplicación, tal como se muestra en la Ecuación (16).

$$rel(t_i) = \exp\left(-\lambda_m * T_{rent}\left(t_i, VM(m,k)\right)\right), \quad (15)$$

siendo $\lambda_m = \{0{,}001, 0{,}003, 0{,}002\}$ los valores de fiabilidad de los tres clouds usados en este trabajo.

$$reliability = \prod_{t_i \in T} rel(t_i) \quad (16)$$

3.2 Definiciones.

En este apartado explicaremos distintos métodos y algoritmos más profundamente para entender mejor el mecanismo utilizado para resolver el problema descrito en la sección anterior. Concretamente, vamos a utilizar un algoritmo multiobjetivo de inteligencia de enjambre, por lo que a continuación pasaremos a explicar estos términos.

3.2.1. **Algoritmos Genéticos.**

Un algoritmo genético está basado en la vista clásica de un cromosoma como una cadena de genes, siendo ésta una versión ejecutable por ordenador. Los algoritmos genéticos se inspiran en la evolución natural para solucionar un problema de optimización que de otra forma serían difíciles o computacionalmente costosos. La estructura de un algoritmo genético es un proceso iterativo que actúa sobre una población de individuos.

Un individuo o cromosoma sería una solución, y el algoritmo tendría un conjunto de soluciones (pull inicial) a elegir. Este individuo estaría compuesto por genes, que serían los valores de cada solución adaptadas al problema que tenemos. Cada gen se puede caracterizar de distinta forma, a saber: enteros, reales, binarios…

Cualquier algoritmo genético se podría dividir en 6 fases:

1. Inicialización: el pull inicial se crea de manera aleatoria, siendo suficientemente grande y diverso. De esta forma, cuando se calcule su función de coste o función de *fitness*, ésta será distinta en todos sus valores, en caso contrario el programa no reconocerá bien los resultados.
2. Función de *fitness*: la función de *fitness* evalúa cada uno de los individuos de la población, y les asigna un valor numérico que determina cómo de bueno es ese individuo. Dicha función puede ser a maximizar (cuanto más alto sea su

valor mejor será la solución) o a minimizar (cuanto más bajo sea su valor mejor será la solución). Esta función puede determinar la condición de parada del algoritmo genético, cuando el valor de un individuo supere un umbral determinado.

3. Selección: se utiliza para elegir a los individuos que se usarán para la reproducción. Existen diferentes formas de llevar a cabo la selección. Tres de ellas son: selección proporcional, cuanto mejor sea el fitness de un individuo más probabilidades de ser elegido; selección por torneo, se eligen varios individuos al azar de la población seleccionando al de mejor fitness; y por último selección por rango, los individuos se ordenan en función de su fitness y la probabilidad que tienen de reproducirse va en función de su posición. Hay ocasiones en las que interesa escoger soluciones que no sean tan buenas, ya que podrían favorecer más adelante a la función de reproducción.
4. Reproducción: para crear nuevos individuos a partir de los existentes, es necesario juntar dos o más y mezclarlos entre sí, de esta forma obtenemos nuevas soluciones. Uno de los métodos más utilizados es partir por la mitad a los individuos y cruzarlos entre sí, obteniendo dos nuevos con la mitad del material genético de los padres.
5. Mutación: puede darse el caso de caer en mínimos locales, impidiendo obtener una solución mejor porque los individuos que tenemos en la población nos impiden llegar a una mejor solución. Para ello utilizamos la mutación, que se da en un porcentaje muy bajo de individuos, entorno al 1-5% (este valor puede variar ya que dependerá del problema) en el cual se cambia aleatoriamente uno o varios genes, modificando la solución y reduciendo, por tanto, la probabilidad de caer en mínimos locales.
6. Eliminación: Tras la reproducción habrá más individuos en la población de los que había inicialmente (generalmente se duplica). Para evitar que la población crezca sin control, se eliminarán los peores individuos de la población, manteniéndose ésta siempre en un número fijo de individuos.

Cuando se termina el proceso, se repetirá tantas veces como sea necesario hasta que se cumpla la condición de parada del algoritmo, que bien puede ser un número

de iteraciones determinado, un número de individuos creados, un valor de fitness de un individuo que supere un umbral, etc.

Es buena idea que en la nueva población haya una copia exacta, sin modificar ni cruzar, de cada uno de los individuos elegidos durante la selección**.** De esta forma, si por alguna razón la nueva población de individuos resulta ser peor que la anterior, por lo menos no se pierden soluciones que hasta entonces eran buenas (Tr4nsduc7or, 2015).

3.2.2. **Algoritmos Multiobjetivo.**

En un problema de optimización se trata de encontrar un valor que represente el resultado óptimo para una función objetivo. Pero en ciertos casos se dan problemas en los que se requiere la optimización simultanea de más de un objetivo. En muchos de estos casos, dichos objetivos son contrapuestos, uno no puede mejorar sin empeorar el otro, es decir, entrarían en conflicto. Habría que hallar la mejor solución de ambos, en vez de empeorar uno y mejorar otro.

En los algoritmos de este caso, es necesario utilizar operadores que permitan comparar soluciones con varios objetivos, como la dominancia de una solución frente a otra. Podemos decir que una solución *x* domina a otra solución *y* si *x* es mejor que *y* en al menos un objetivo y mejor o igual en el resto. Ejemplos de algoritmos multiobjetivo son el *MOFA, NSGA II*, etc. Además, en este tipo de optimización, habitualmente se tendrá un conjunto de soluciones final, y no una única solución como ocurre con las aproximaciones monobjetivo. A este conjunto de soluciones se le conoce como frente óptimo de Pareto y estará compuesto por todas aquellas soluciones no dominadas de la población.

3.2.3. **Inteligencia de Enjambre.**

La inteligencia de enjambre (*IE*) es el comportamiento colectivo de sistemas descentralizados y autoorganizados.

Los sistemas se componen normalmente de una población de agentes que interactúan entre sí y con su medio ambiente. Los agentes siguen reglas muy simples, y aunque no hay una estructura de control centralizada, las interacciones entre estos agentes dan lugar a la aparición de la conducta global inteligente, sin que el

individuo sea inteligente. Ejemplos naturales de IE pueden ser las hormigas o las abejas.

La inteligencia en enjambre es un sistema multi-agente que tiene un comportamiento autoorganizado que muestra un comportamiento inteligente. Con esta definición, podemos ver que la inteligencia no solo se trata de una entidad única, individual y de enorme complejidad, sino que además comprende un sistema ordenado, estructurado, con funciones definidas y con cierta complejidad organizativa. (Beni, et al., s.f.). A continuación, mostraremos algunos ejemplos de algoritmos de inteligencia de enjambre.

3.3.2.1 Optimización por Enjambre de Partículas.

El algoritmo de Optimización por Enjambre de Partículas (*Particle Swarm Optimization*, *PSO*) (Caparrini, 2016) es un método computacional evolutivo. Es un algoritmo que se puede usar en espacios de búsqueda con muchas dimensiones. Este método se inspira en el comportamiento de los enjambres de insectos en la naturaleza.

En este algoritmo, tendremos una función a la que le podremos aplicar diferentes *x* e *y*, que serían coordenadas que aplicamos a nuestro problema y obtendremos diferentes valores, de los cuales buscaremos máximos o mínimos. Es el mismo caso que en los algoritmos genéticos, y la función se denomina función de fitness. Para encontrar los mejores valores, la idea es colocar partículas en el espacio de búsqueda al azar, pero dando la posibilidad de moverse considerando unas reglas que tienen en cuenta el conocimiento personal de cada partícula y el conocimiento global del enjambre.

El funcionamiento consiste en que cada partícula tiene una posición y una velocidad, en nuestro caso dada por los valores *x* e *y*. Además, las partículas en el mundo real físico tienen cierta inercia que las obliga a continuar hacia donde se dirigían, así como una aceleración, que dependerá de dos factores:

- Cada partícula es atraída hacia la mejor localización personal, la mejor localización que la partícula ha encontrado.
- Cada partícula es atraída hacia la mejor localización que ha sido encontrada por el conjunto de partículas en el espacio de búsqueda.

En definitiva, cada partícula es atraída por las demás dependiendo del mejor personal y el mejor global.

Una variante del PSO, denominada *MOS* (*Multi-Objetive Scheduling*), ha sido utilizado en (Hu, et al., 15 July 2018) para resolver el problema expuesto en este trabajo.

3.3.2.2 Algoritmo de la Colonia de Hormigas.

El algoritmo de optimización por colonia de hormigas (*Ant Colony Optimization, ACO*) (Caparrini, 2018) está en el comportamiento de las hormigas. ACO se utiliza cuando es necesario encontrar un camino hacia un objetivo o meta teniendo como posibilidad varios caminos. Las hormigas son insectos sociales, y se agrupan en comunidad formando colonias. La comunicación de las hormigas se realiza por la modificación del entorno, depositando por donde pasan una cantidad de sustancia química en el suelo, llamada feromona, la cual incrementa la posibilidad de que otras hormigas sigan el mismo camino.

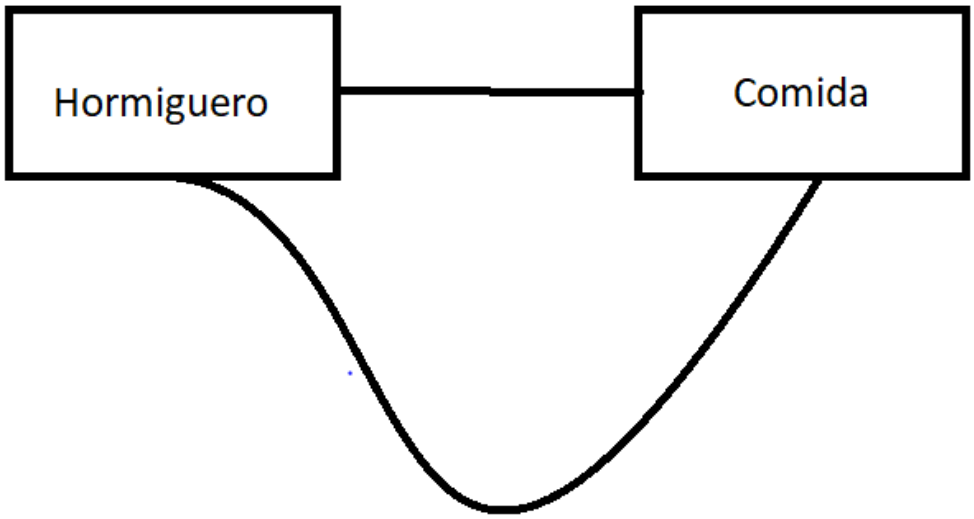

Figura 3.1 Camino más corto.

En el ejemplo de la figura 3.1, podemos ver el funcionamiento de las hormigas. Existen dos caminos desde el punto en el que está el hormiguero hasta la comida, uno mayor que otro. Comienzan a ir las hormigas desde un punto a otro, por los dos caminos ya que aún no hay feromonas en el suelo para indicar qué preferencia tienen las demás hormigas. Una vez que ha pasado cierto tiempo, como cada vez que pasa

una hormiga deja feromonas, por el camino más rápido han pasado una mayor cantidad de hormigas que por el más lento, por tanto, la cantidad de feromonas es mayor, y poco a poco las hormigas se decidirán por el camino más corto.

Este comportamiento se puede aplicar a varios objetivos para descubrir la distancia mínima entre dos puntos. Gracias al algoritmo de las hormigas, se pueden resolver problemas tan famosos como el viajante de comercio. Este problema trata de encontrar la distancia mínima recorriendo un conjunto de ciudades, empezando y terminando en la misma y sin pasar dos veces por un punto.

Cuando comenzamos el algoritmo, tenemos que determinar el número de hormigas que vamos a utilizar, es decir, la elección de camino se hará x veces, pero a medida que avanzamos, el camino que seguirá la hormiga siguiente tendrá en consideración el de las anteriores mediante las feromonas.

3.3.2.3 Colonia Artificial de Abejas

El algoritmo de colonia de las abejas (*Artificial Bee Colony, ABC*) (Karaboga, 2010) es un algoritmo metaheurístico y simula el comportamiento de un enjambre de abejas. El algoritmo tiene tres tipos de abejas: observadoras, obreras y exploradoras. Al comienzo del código tenemos que determinar el número de abejas de cada tipo. Este algoritmo también define dos características esenciales para su funcionamiento: exploración de un espacio de soluciones que devuelva un feedback positivo y abandono del espacio de soluciones que otorgue un feedback negativo.

Respecto al funcionamiento de las abejas, las obreras explorarán el espacio de soluciones cercano a su situación actual, tratando de alcanzar una solución mejor. Las observadoras se fijan en una obrera según un porcentaje, cuanto mejor sea la solución de la obrera, mayor posibilidad tiene la observadora de elegir esa obrera. Cuando se haya elegido una obrera, se copia la solución y se explora el espacio cercano tratando de mejorar. Por último, cuando una abeja lleve varios intentos de mejora sin conseguirlo, se transforma en exploradora o scout moviéndose a un punto aleatorio del espacio de soluciones para seguir la búsqueda en ese punto. Después de cada una de estas fases, es necesario realizar una evaluación para saber cómo de buenas son las diferentes abejas y posteriormente ordenar todas las soluciones. Las x primeras soluciones se clasifican como obreras. También podemos guardar la mejor

solución de forma permanente si llega el caso en la que la población empeora, aunque se podría impedir si no cambiamos las abejas por una exploración peor.

4

MATERIAL Y MÉTODO

En este capítulo vamos a explicar la metodología utilizada en este trabajo, concretamente definiendo el algoritmo utilizado, explicando el formato de los ficheros de datos y describiendo la metodología seguida en los experimentos realizados.

4.1 Algoritmo de la luciérnagas.

El algoritmo utilizado en el desarrollo de este proyecto ha sido el algoritmo de las luciérnagas (*Firefly Algorithm, FA*) (X.-S. Yang, 2009), el cual forma parte de los algoritmos de inteligencia en enjambre. Concretamente se ha implementado una versión multiobjetivo del mismo, cuya explicación será detallada posteriormente. Este algoritmo está basado en el comportamiento de las luciérnagas en la naturaleza, las cuales usan una luz para la comunicación o la reproducción. El algoritmo reproduce la atracción que sufren las luciérnagas dependiendo de la cantidad de luz que observan. Las luciérnagas más brillantes atraen a las menos brillantes. Todas las luciérnagas representan una solución al problema. El brillo que emite una luciérnaga está asociado con la calidad de la solución, a mayor brillo, mejor solución, es proporcional al valor de la función objetivo.

Tenemos que tener en cuenta tres reglas:

1. Todas las luciérnagas pueden ser atraídas por otras.
2. Una luciérnaga que emite menor brillo será atraída por otra que emita mayor brillo, moviéndose la que tiene peor solución.
3. La calidad de la solución que tiene una luciérnaga es proporcional al brillo que emite.

Si una luciérnaga no encuentra a otra cerca con mejor brillo que el suyo, se moverá de forma aleatoria.

Para explicar el algoritmo, cuyo pseudocódigo podemos ver en el Algoritmo 1, el FA realiza siempre los pasos de inicialización de la población. Primero determinamos el número de luciérnagas con las que empezaremos. Posteriormente inicializaremos aleatoriamente estas soluciones. Al comienzo, tendremos que especificar el número de generaciones, que será el número de iteraciones que el algoritmo se ejecutará (líneas 1-4). También tendremos en consideración la cantidad

de veces que se procesa una luciérnaga sin mejorar. Si excede el número establecido, se elimina y se crea otra aleatoria.

Algoritmo 1. Luciérnagas.

```
1  Definición función objetivo
2  Generación población inicial N
3  Calcular intensidad población
4  Mientras t < XEvaluaciones hacer
5      Para i desde 0 hasta  n-1 hacer:
6          Para j desde 0 hasta  n-1 hacer:
7              Si I_i > I_j hacer: // mover la luciérnaga i hacía la luciérnaga j
8                  Atracción de j sobre i
9                  Evaluar nueva solución
10                 Actualizar intensidad población
11             Fin si
12         Fin para
13     Fin para
14 Fin mientras
15 Ordenar luciérnagas y buscar la mejor
16 Procesos secundarios y visualización
```

Una vez que iniciamos el algoritmo, lo primero es acercar las luciérnagas menos brillantes a las más brillantes. En este paso todas las luciérnagas se comparan con las demás comprobando si son mejores (línea 7). Así obtenemos una nueva luciérnaga. Esta nueva remplaza a la anterior si si función de *fitness* es mejor que la original, si no, es descartada. Cuando ocurra esto, es necesario llevar una cuenta de las luciérnagas estancadas, para evitar que no avance la población (líneas 8-10). A continuación, se evalúa la nueva solución y se establece su intensidad de luz. Una de las posibles condiciones de parada es la cantidad de nuevas luciérnagas que el algoritmo ha evaluado. Cuando ha evaluado un número determinado de luciérnagas, deja de ejecutarse.

Posteriormente, después de cada iteración, las soluciones validas son guardadas para evitar que se pierdan mejores soluciones si la población tiende a empeorar, aunque no se daría el caso si comprobamos si la nueva luciérnaga es mejor que la que vamos a sustituir.

Cuando termina el bucle principal, se realiza un ranking de que solución del conjunto de población es mejor (línea 15).

En el algoritmo FA hay dos factores importantes, la variación de la intensidad de la luz y la fórmula de atracción que está determinada por el brillo. La intensidad se

puede dar en un rango de valores determinado, pero la atracción depende del espectador. Ambos factores disminuyen con la distancia.

La fórmula que determina la intensidad de la solución viene dada por la Ecuación (17).

$$\boldsymbol{I(r) = \frac{Io}{1+\gamma r^2}}, (17)$$

donde I_o es la intensidad original, r es la distancia y γ es un número comprendido entre [0,001, 100].

Como la atracción es proporcional a la intensidad de la solución, podemos definir la atracción como:

$$\beta(r) = \beta_0 e^{-\gamma r^2}, (18)$$

donde β_0 es la atracción con distancia r = 0.

Para saber la distancia entre dos soluciones se utiliza la distancia cartesiana, que es calculada mediante la Ecuación (19)

$$r_{ij} = |x_i - x_j| = \sqrt{\sum_{k=1}^{d} (x_{i,k} - x_{j,k})^2}, (19)$$

donde x_i y x_j son dos soluciones y d la cantidad de dimensiones u objetivos que tienen las soluciones.

En el caso de 2 dimensiones, que sería nuestro caso, la Ecuación (19) se transforma en la Ecuación (20).

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}, (20)$$

Por último, la atracción que realiza una solución más brillante x_j hacia otra solución x_i se determina siguiendo la Ecuación (21).

$$x_i = x_i + \beta_0 e^{-\gamma r_{ij}^2} (x_j - x_i) + \alpha \left(rand - \frac{1}{2} \right), (21)$$

donde *rand* es un valor aleatorio entre 0 y 1 y α es el peso que se le da al factor aleatorio y tiene un valor entre 0 y 1.

Algoritmo 2. Luciérnagas multiobjetivo

```
Creación población
Calcular Intensidad de las luciérnagas (coste, tiempo, fiabilidad)
Mientras x < evaluaciones hacer:
    Para i desde 0 hasta n-1 hacer:
        Para j desde 0 hasta n-1 hacer:
            Si I_j ≺ I_i hacer: // si la luciérnaga j domina a la i
                Atraer luciérnaga i hacia j
                Calcular fiabilidad
                Mientras fiabilidad < relc hacer:
                    Calcular nueva solución
                Fin mientras
                Calcular Intensidad luciérnaga
                Si nuevaLuciérnaga ≺ viejaLuciérnaga hacer:
                    Cambiar luciérnagas
                Fin si
            Fin si
        Fin para
    Fin para
    Si misma población hacer:
        Estancamiento
        Crear nueva población
        Guardar población actual
Fin mientras
Fast Non-Dominated Sort
Frente Óptimo
```

4.1.2. **Algoritmo de las Luciérnagas Multiobjetivo.**

En este problema, tenemos dos objetivos contrapuestos que queremos optimizar, por tanto, el algoritmo de las luciérnagas por sí solo no es apto. Para ello utilizaremos el algoritmo de las luciérnagas con multiobjetivo (Hidalgo-Paniagua, et al., 2017)

Como veremos, el algoritmo de las luciérnagas multiobjetivo (*MOFA*) que utilizaremos cambia a la hora de comprobar con respecto al original cómo de buena

es una solución al compararla con las demás. Podemos ver el comportamiento del mismo en el Algoritmo 2. Como en el algoritmo original, comenzamos creando una población con una cantidad de individuos que estableceremos, siendo en nuestro caso cincuenta individuos (soluciones). Posteriormente establecemos el número de individuos nuevos que crearemos, que será el número de evaluaciones, en nuestro caso cincuenta mil.

Creamos la población con cincuenta individuos y comenzamos el bucle principal hasta completar las evaluaciones establecidas (líneas 1-2). Para saber si un individuo es mejor que otro utilizaremos la dominancia, de la que hablaremos con más detalle en el próximo punto.

Si todos los individuos de una población no se dominan entre sí, se dice que la población ha sufrido un estancamiento (líneas 19-23), es decir, no puede mejorar más, por tanto, es necesario controlar este factor dentro del ámbito de la dominancia.

Recorremos toda la población comprobando todos los individuos entre sí, si uno es dominado por el otro, la solución peor es atraída por la mejor, por tanto, es necesario aplicar un operador de atracción para acercar la solución peor a la nueva solución (línea 7).

Cuando se ha aplicado el operador de atracción, se comprueba si el nuevo individuo es apto en cuanto a fiabilidad (líneas 8-9). Si no es apto, se cambia un número de tareas dentro de la solución entre uno y diez. Cuando se ha decidido el número de tareas que se van a modificar, se determina de forma aleatoria qué tarea va a ser cambiada, que será un número entre cero y la cantidad de tareas que estemos utilizando para la aplicación. Una vez decidido qué tarea modificar, podemos variar las unidades de computo, siempre hacia un número mayor para obtener una mejor solución (si no hemos llegado al límite) o variar el cloud en el que ejecutaremos las tareas hacia uno con mejor tasa de error, ya que dependiendo de en cual nos encontremos, el índice de error es menor o mayor. En este último caso, si no hemos llegado al límite reducimos en uno el número en la *VM*. Por último, se vuelve a calcular la fiabilidad de la solución y se comprueba con el límite impuesto. Si es mejor, continuamos la ejecución, si no, repetimos el proceso anteriormente descrito hasta que sea apta.

Posteriormente se calcula el coste y el tiempo de la solución (línea 12) y se comprueba la dominancia de la nueva solución respecto de la antigua. Si la nueva solución domina, la antigua se elimina, y viceversa (líneas 13-14).

Una vez que se ha terminado el proceso y se han creado las soluciones pertinentes, se almacenan todas y se aplica el algoritmo de ordenación no dominada *Fast Non-Dominated Sort*, para obtener el Frente de Pareto óptimo (líneas 24-25). De esto último hablaremos más adelante.

4.1.3. **Operador de atracción.**

En el algoritmo FA, para determinar cómo de buena es una solución, la luciérnaga emite una luz en función de la calidad de la solución, a mayor calidad, mejor solución. Esto se utiliza también para atraer a otras luciérnagas que tienen peor solución. Pero la atracción no depende solo de la intensidad sino de la posición de las dos luciérnagas. El operador de atracción se utiliza para mezclar las dos soluciones. En el MOFA aplicado a nuestro problema, es posible cambiar dos valores por cada tarea: el cloud y la máquina dentro del cloud. Para ello, hemos aplicado el operador de atracción (ecuación 22) dos veces, una para cada elemento, siguiendo la Figura 4.1 vista más adelante.

$$x_i = x_i + \beta_0 e^{-\gamma r_{ij}^2} \left(x_j - x_i\right) + \alpha \left(rand - \frac{1}{2}\right), (22)$$

donde *r* significa la distancia que hay entre la solución *i* y la solución *j*. A mayor distancia, menor atracción. La β_o significa la atracción de una luciérnaga sobre otra cuando la distancia *r* entre ellas es 0. x_i y x_j son las soluciones. α sería la importancia que se da al valor aleatorio, el cual está comprendido entre 0 y 1. Por último, γ sería un valor a parametrizar entre 0,01 y 100.

Debido a que el operador de atracción original del algoritmo trabaja con números reales y nuestros cromosomas son enteros, hubo que modificar dicho operador. La primera opción fue redondear el resultado de la ecuación (22). Después de hacer varios experimentos con el operador original y aplicando uno aleatorio, los resultados obtenidos eran peores que los aleatorios, por lo que fue descartado. La

segunda opción fue la de realizar el truncamiento en lugar de redondear en el resultado de la Ecuación (22), mejorando los resultados considerablemente. Esta opción fue originalmente definida en (Onwubolu & Davendra, 2009).

4.1.4. **Dominancia**

En el algoritmo MOFA, la dominancia de un individuo con dos objetivos respecto de otro individuo se describe siendo uno de los dos objetivos mejor que el otro individuo y el otro mejor o igual. En nuestro caso, los dos objetivos serán coste y tiempo. Para comprobarlo se utiliza el operador de dominancia. Si un individuo es mejor que otro en coste o tiempo y mejor o igual en el otro factor, se dice que el individuo A domina al individuo B, o bien que el individuo B es dominado por A. Si no se cumple esta regla, A no domina a B, pero B sí podría dominar a A.

4.1.5. **Frente de Pareto Óptimo.**

El frente de Pareto se da en situaciones donde se cumple que no es posible beneficiar un objetivo sin perjudicar a otro, es decir, busca un equilibrio para todos los objetivos. El óptimo de Pareto estaría compuesto por todas aquellas soluciones que no son dominadas por ninguna otra. En los algoritmos multiobjetivo, al no poder obtener una única solución, el resultado será un frente de Pareto óptimo, donde estarán las mejores soluciones encontradas durante la ejecución del algoritmo.

4.1.6. **Fast Non-Dominated Sort.**

Para obtener el frente de Pareto óptimo de una población vamos a utilizar el algoritmo de ordenación no dominada *Fast Non-Dominated Sort* (Deb, et al., 2002). Este algoritmo recorrerá la población, comparando cada solución con el resto para conocer si está dominada o no. El algoritmo identificará varios frentes que estarán compuestos por todas aquellas soluciones que solamente estén dominadas por el frente anterior. Si una solución no es dominada por ninguna, se considera perteneciente al frente óptimo, si es dominada por las soluciones del frente óptimo, formaría parte del segundo frente, y así se podrían crear varios frentes de varios niveles, dependiendo de la cantidad de soluciones que dominen a otras. En nuestro caso, como el algoritmo MOFA no necesita tener las soluciones agrupadas por frentes, no es necesario guardar todos los niveles de frentes, por lo que únicamente se ejecutará al final del mismo para quedarnos con las soluciones no dominadas.

Además, se ha modificado el algoritmo eliminando la segunda parte del mismo con el fin de que únicamente devuelva las soluciones que pertenecen al frente óptimo de Pareto (no necesitamos los otros frentes).

El funcionamiento del algoritmo modificado sería el siguiente:

1. Procesamos todas las soluciones comparándolas con las demás, anotando la cantidad de soluciones que dominan a cada solución.
2. Posteriormente se comprueba por cada solución la cantidad de soluciones que la dominan.
3. Las soluciones que no sean dominadas pertenecerán al frente óptimo de Pareto, que será el resultado del algoritmo.

4.1.7. **Representación del individuo.**

Como hemos utilizado un algoritmo de inteligencia en enjambre, es necesario establecer un conjunto de soluciones al comienzo del programa. Estas soluciones tienen que ir en función del problema, guardando todos los datos que sean necesarios para poder obtener la solución.

La población está formada por *n* individuos, que en nuestro caso son cincuenta. Cada individuo alberga diferentes características, a saber: el coste, el tiempo y la fiabilidad que tiene el problema al ejecutarlo con la configuración dada en el individuo. Todos estos valores son calculados una vez se tenga la configuración de las máquinas virtuales. Cada individuo contiene un array de tamaño el número de tareas que tiene la aplicación que se está configurando. Cada casilla del array contiene el cloud (*VM*) y la máquina (*CU*) que van a ejecutar la tarea, así como el tiempo final que tarda cada tarea en ejecutarse en conjunto con sus antecesoras, es decir, el tiempo que tarda en la línea temporal de ejecución del programa contando los retrasos. Este último dato no es parte propiamente del individuo, en realidad se utiliza porque permite aumentar el rendimiento del problema y reducir el tiempo de computo, ya que, de otra forma, el tiempo de cada tarea se calcularía varias veces y es siempre el mismo, y de esta forma lo calculamos una vez únicamente pudiendo usarlo tantas veces como sea necesario sin aumentar el tiempo de ejecución del algoritmo.

Coste

Tiempo

Fiabilidad.

CU VM TiempoFinal	CU VM TiempoFinal	CU VM TiempoFinal	CU VM TiempoFinal	CU VM TiempoFinal	CU VM TiempoFinal	CU VM TiempoFinal	CU VM TiempoFinal

CU = Unidad de computo

VM = máquina virtual.

Figura 4.1 Individuo.

En la Figura 4.1 podemos ver un individuo que forma la población. Nuestra población se constituye de cincuenta individuos. Tanto la CU como la VM se inicializan aleatoriamente al comenzar el bucle principal. En este ejemplo, el problema tendría ocho tareas.

4.1.8. **Otras Estructuras Necesarias**

Además de la representación del individuo, también necesitamos otras estructuras para guardar los datos necesarios para resolver el problema. Por una parte, tenemos un array, el cual contiene los ficheros que comparten las tareas de la aplicación. Cada casilla del vector representa un fichero y contiene el identificador del fichero junto con el peso del mismo.

Por otro lado, tenemos otro array con toda la información de las tareas. Cada casilla del array representa una tarea, y contiene el identificador de la tarea, su carga de trabajo, un array que contiene todas las tareas dependientes de ésta y otro array con las tareas sucesoras a ésta. Además, tenemos dos arrays con todos los ficheros de entrada y salida de la tarea, ya que es necesario tenerlos en cuenta a la hora de calcular el tiempo de transferencia de datos de una tarea a otra.

4.1.9. **Operadores Multiobjetivo Utilizados**

Para poder comparar y hacer una valoración de los resultados obtenidos, utilizaremos dos operadores multiobjetivo, el hipervolumen (Beume, et al., 2009) y el *Set Coverage* (Zitzler, et al., 2003).

Para la primera métrica, se recogen todos los resultados y se normalizan a valores de entre 0 y 1, utilizando los valores máximos y mínimos de los valores que se utilizarán, que serán del frente de Pareto. Una vez normalizado, como los dos objetivos son a minimizar, el hipervolumen se calcula utilizando el punto de referencia (1, 1).

El segundo operador (SC) es un operador de tipo binario. Mide el porcentaje de soluciones que un frente de Pareto A cubre (son dominadas o iguales) de un frente de Pareto B. Si todas las soluciones de B están cubiertas por, al menos, 1 solución de A, obtenemos un SC de 1, que sería el 100%. Por otro lado, si ninguna solución de B está cubierta por alguna solución de A, el SC sería de un 0%. Este operador no es recíproco, es decir, que A cubra al 87% de las soluciones de B no implica que B cubra el 13% de las soluciones de A

4.2 Estructura de los Ficheros de Entrada

Para llevar a cabo la ejecución del algoritmo, es necesario extraer todos los datos de los ficheros de datos de cada aplicación. Estos ficheros tienen información en bruto, por lo que hay que transformar esa información a las estructuras descritas anteriormente. Para ello, se tuvo que implementar un preprocesado que se encargara de realizar esta tarea. Por cada fichero con extensión *.dag* que hemos utilizado, teníamos diferentes datos que obtener, aunque no todos eran necesarios.

El fichero *.dag* comienza mencionando los archivos que se van a utilizar durante la ejecución de la aplicación. Cada línea perteneciente a esta sección va encabezada por *FILE*, separado por un espacio y el nombre de los archivos junto con el peso de cada uno de ellos. Todos estos datos son necesarios extraerlos para las dependencias de los ficheros de las tareas y para calcular el tiempo de transmisión de datos.

Seguidamente, nos encontramos con la sección *TASK*, en la que nos encontramos las diferentes tareas que tiene la aplicación, seguido del identificador que las reconoce y el tiempo que tardan en ejecutarse en una unidad de computo, es decir, su carga de trabajo.

Más adelante, nos encontramos con los archivos que necesita cada tarea para poder ejecutarse. Estos archivos pueden ser de entrada o de salida y vienen identificados por las etiquetas *INPUT* y *OUTPUT* respectivamente. Después de esta etiqueta, obtenemos el identificador de la tarea que lo usa y seguidamente los nombres de archivos. Puede haber varios archivos para una misma tarea, tanto *INPUT*, que son los que necesita la tarea, como *OUTPUT*, que son los que crea una vez ejecutada la tarea.

Por último, tendremos entradas etiquetadas como *EDGE*, que van a identificar las dependencias de las tareas. Esta parte es dependiente del tipo de aplicación que vamos a lanzar, ya que la estructura interna de las aplicaciones es distinta. Esto lo podemos ver en la Figura 4.2.

4.3 Experimentos.

A continuación, hablaremos sobre las ejecuciones, la obtención de información para elaborar un resultado en el problema, así como los parámetros que son necesarios en el programa, y como se ha llevado a cabo el ajuste paramétrico.

4.3.1. **Aplicaciones Utilizadas.**

Para el experimento, hemos utilizado 4 aplicaciones distintas[1]. Estas aplicaciones se diferencian en la estructura interna, presentando diferentes nodos finales y diferentes dependencias en los nodos. Cada nodo es una tarea. Las 4 aplicaciones utilizadas serán Cybershake, Ligo, Montage y Sipht. La Figura 4.2, muestra una representación de su estructura.

[1] Los ficheros utilizados han sido descargados de https://download.pegasus.isi.edu/misc/SyntheticWorkflows.tar.gz

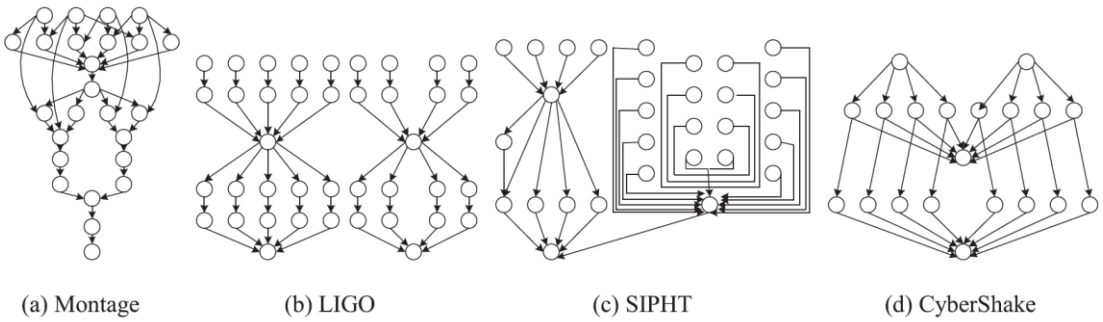

Figura 4.2 Estructura de las aplicaciones utilizadas en los experimentos.

Como podemos ver, cada estructura y por tanto cada ejecución de las aplicaciones son distintos.

En cada una, tenemos varios tamaños en número de tareas que hemos probado. En nuestros experimentos, hemos utilizado las aplicaciones con dimensiones de 100, 400 y 700 tareas para cada tipo de aplicación, probando por tanto nuestro algoritmo con tamaños pequeños, medianos y grandes.

4.3.2. **Parametrización.**

Como se comentó en la sección que describía el MOFA, este algoritmo requiere la parametrización de varios valores que van a influir en el comportamiento del sistema. Para realizar el ajuste paramétrico, hemos utilizado un tamaño de aplicación de 100 tareas en los 4 tipos de aplicaciones, probando diferentes combinaciones para obtener el mejor resultado.

Cada combinación de parámetros lo ejecutamos un total de 11 veces, cambiando α a los valores {0, 0,25 ,0,5 ,0,75 ,1}, y γ a los valores {0,0001, 0,01, 1, 100}. Ejecutando esto, obtenemos un total de 220 ficheros por aplicación, cada uno de los cuales tendrá el frente óptimo de Pareto obtenido con la correspondiente combinación de parámetros. Para tratar todos los resultados por igual, necesitamos normalizar los datos y realizamos el cálculo del hipervolumen. Así conoceremos cómo de buena es una solución.

Para normalizar los datos, primero tenemos que buscar los valores mayor y menor de cada objetivo del conjunto a normalizar, y posteriormente calcular el valor normalizado siguiendo la ecuación (23).

$$Valor\ normalizado = \frac{V - Min}{Max - Min}, (23)$$

siendo *V* el valor original y *Min* y *Max* los valores mínimo y máximo respectivamente de todo el conjunto de frentes a analizar.

Posteriormente, necesitamos clasificar los resultados juntando los ficheros de las 11 ejecuciones de cada aplicación que comparten la misma configuración y calculamos el promedio de los hipervolúmenes de dicho conjunto. Por último, nos quedaremos con la combinación de parámetros con el mejor hipervolumen medio. Como conclusión a este experimento, determinamos que la mejor combinación de parámetros es γ = 0,0001 y α = 0,75.

Por último, para asegurarnos de que la parametrización es correcta, lanzamos un último conjunto de experimentos con γ=0,001 (valor intermedio entre el obtenido y el superior; no se realizó un experimento con un valor inferior por ser ya muy bajo) y α en el rango de valores {0, 0,25, 0,5, 0,75, 1} obteniendo de nuevo 11 resultados para cada aplicación. Indicar que la configuración obtenida inicialmente seguía teniendo mejor resultado, por lo que se mantuvo dicha configuración.

4.3.3. **Ejecución de los experimentos.**

Una vez que hemos determinado los parámetros óptimos, lanzaremos los experimentos que utilizaremos para determinar el resultado. Para obtener la información, se han lanzado 31 ejecuciones de cada aplicación con la mejor configuración obtenida en el proceso de parametrización. Las ejecuciones de las 4 aplicaciones se han realizado con los tamaños de 100, 400 y 700 tareas.

Por último, se realizó un experimento teniendo en cuenta la fiabilidad, para lo cual hemos lanzado ejecuciones con 100 tareas de la aplicación Montage, 31 ejecuciones tanto del algoritmo MOFA como de la creación de población aleatoria. Estas ejecuciones se dividen según la restricción de fiabilidad, siendo esta de 0,2, 0,4, 0,6 y 0,8.

En el siguiente apartado comentaremos los resultados que hemos obtenido de estas ejecuciones, así como la conclusión a la que hemos llegado.

5

RESULTADOS Y DISCUSIÓN

A continuación, analizaremos los resultados que hemos obtenido de los experimentos. La idea inicial era la de hacer comparativas con (Hu, et al., 15 July 2018), pero debido a que en el artículo no aparece ningún detalle sobre las bases de datos utilizadas y no proporcionan su código fuente, fue imposible realizar tal comparativa. Por tanto, para poder hacernos una idea de la calidad de los resultados obtenidos, se diseñó un programa que generaba soluciones de forma aleatoria, y es con este programa con el que se han realizado las comparaciones de los experimentos. Comenzaremos analizando los resultados sin fiabilidad, que se compone de las 4 aplicaciones con tamaños de 100, 400 y 700 tareas, tanto del aleatorio como utilizando nuestro algoritmo MOFA. Posteriormente comentaremos los resultados con varios valores fiabilidad, que se compone únicamente de la aplicación Montage de tamaño 100 tareas.

Para obtener los resultados de los gráficos de los frentes de Pareto, hemos obtenido la mediana de todos los resultados y posteriormente, de cada experimento mediano, hemos obtenido el frente de Pareto para mostrarlo en el gráfico.

A continuación de los frentes, realizaremos una evaluación con el hipervolumen de ambas ejecuciones y una comparación con el Operador de Cobertura (*Set Coverage*). Este operador mide, dados dos frentes *A* y *B*, el porcentaje de soluciones de *B* que son dominadas o que existen por las soluciones de *A*.

5.1 Aleatorio VS MOFA Sin Fiabilidad.

Comenzamos con las comparativas entre los algoritmos aleatorio y MOFA sin fiabilidad, es decir, sin restricción de fiabilidad, para las cuatro aplicaciones con 100, 400 y 700 tareas.

5.1.1. **Cybershake**

En la Figura 5.1 observamos el experimento Cybershake 100, en color azul el algoritmo MOFA y en color rojo el experimento lanzado de forma aleatoria. Como podemos apreciar, hay una gran diferencia en los resultados, siendo los del aleatorio dominados todos por los del MOFA, aunque el rango de soluciones respecto de la variable “Coste” no es tan amplio como en el aleatorio.

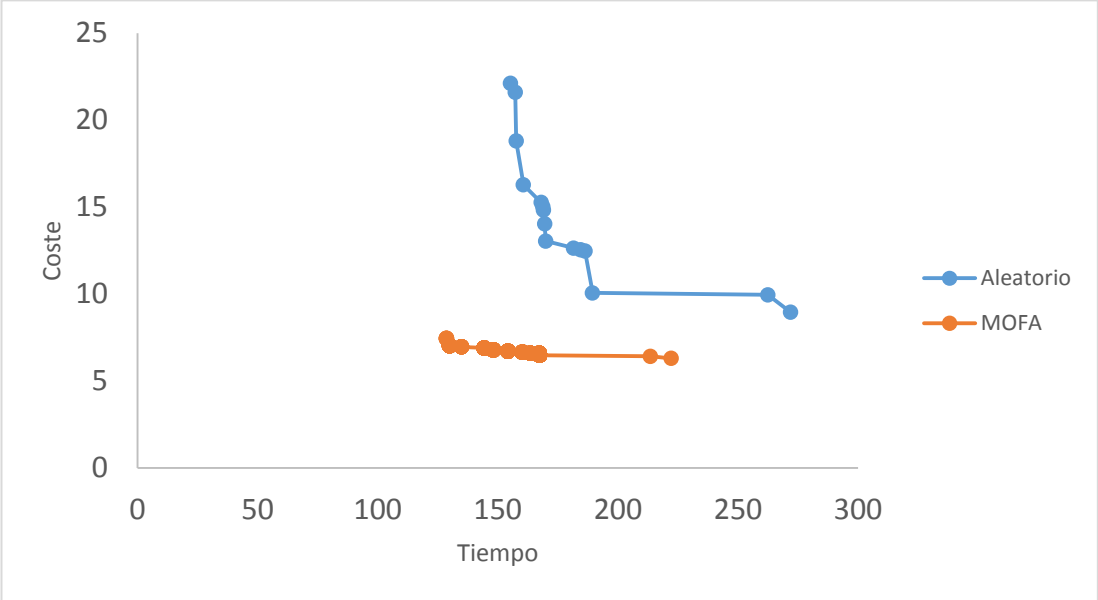

Figura 5.1 Frentes de Pareto mediano de la aplicación Cybershake con 100 tareas.

Respecto a los hipervolúmenes (Tabla 5.2), el aleatorio obtiene un valor de 67,72% respecto al valor **95,50%** del MOFA. Aquí podemos ver más claramente la diferencia que hay entre ambas ejecuciones.

Aplicando el Operador de Cobertura (Tabla 5.1), obtenemos un dominio del **100%** de la ejecución MOFA, frente a un 0% de la ejecución del aleatorio

En la Figura 5.2 observamos el experimento Cybershake 400, al igual que el caso anterior, todos los valores del MOFA dominan al aleatorio, aunque el rango de soluciones es más compacto.

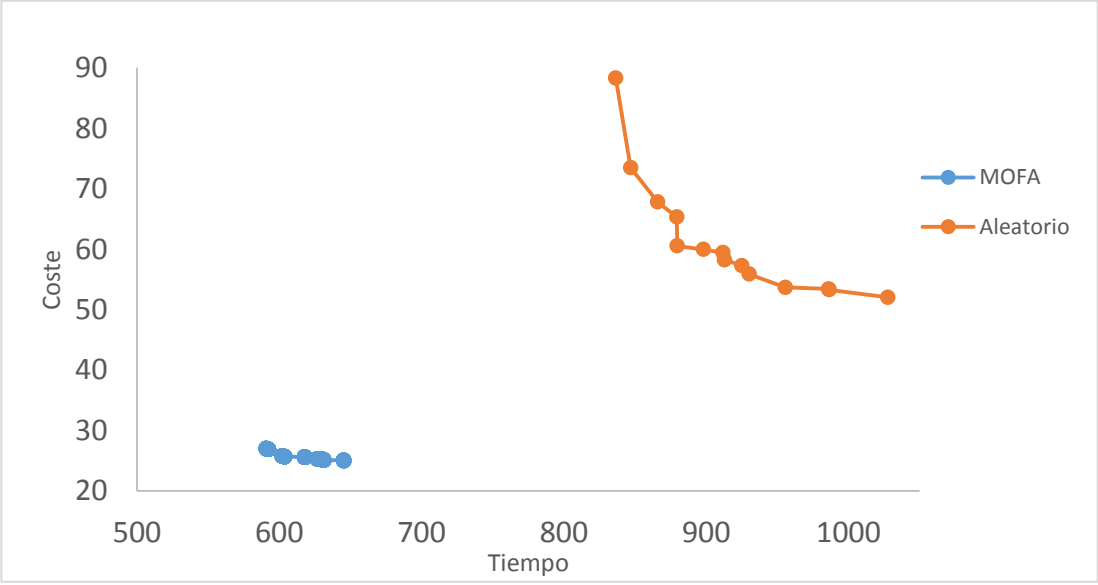

Figura 5.2 Frentes de Pareto mediano de la aplicación Cybershake con 400 tareas.

En lo referente al hipervolumen (Tabla 5.2), en la ejecución aleatoria hemos obtenido un valor de 28,68%, mientras que en el MOFA hemos obtenido un hipervolumen de **98,14%**. La diferencia es sustancial comprobando estos valores.

Podemos observar que todas las soluciones son dominadas por el algoritmo (Tabla 5.1), por tanto, MOFA obtiene un **100%** de cobertura frente al 0% del aleatorio aplicando el *Set Coverage* de nuevo.

En el caso del experimento con 700 tareas (Figura 5.3), hay gran diferencia y mejoría del MOFA respecto del aleatorio. Aunque el conjunto de soluciones del aleatorio se extiende por más valores, las soluciones del algoritmo MOFA son mejores.

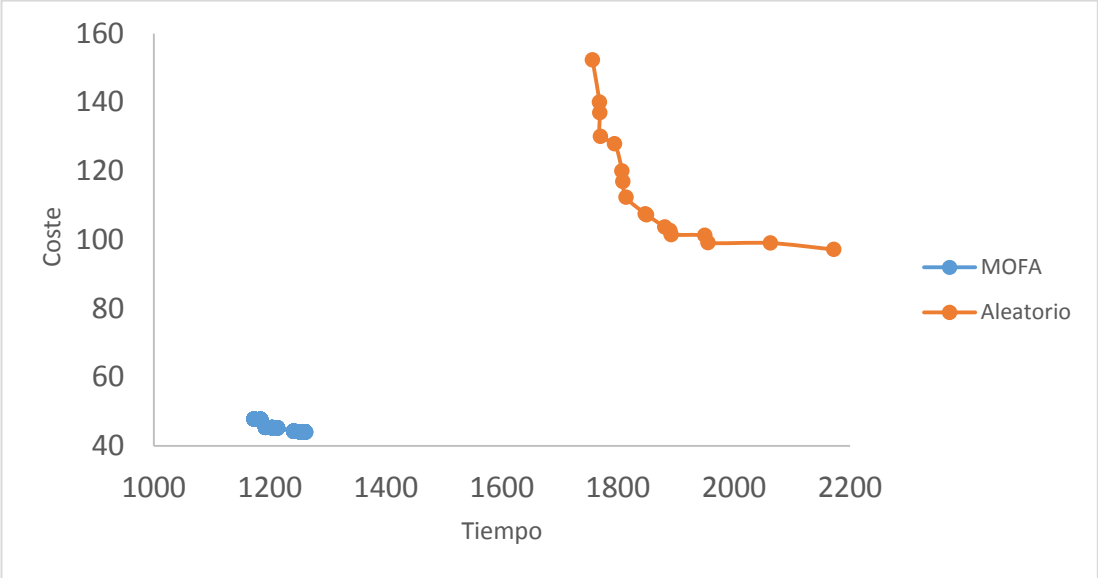

Figura 5.3 Frentes de Pareto mediano de la aplicación Cybershake con 700 tareas.

En lo que respecta a los hipervolúmenes (Tabla 5.2), la ejecución aleatoria obtiene un resultado de 21,93% mientras que la ejecución con el MOFA, obtenemos un hipervolumen de **99,60%**, siendo este muy superior.

Cerrando el apartado de la aplicación Cybershake, todas las ejecuciones, incluida esta, aplicando el Operador de Cobertura, tienen un **100%** de cobertura del algoritmo frente al 0% del aleatorio. Además, como podemos ver comparando los

hipervolúmenes, MOFA va ampliando su ventaja frente al aleatorio según va aumentando el tamaño del problema.

Nº de tareas	MOFA	Aleatorio
100	**100%** cobertura	0% cobertura
400	**100%** cobertura	0% cobertura
700	**100%** cobertura	0% cobertura

Tabla 5.1 *Set Coverage* Cybershake.

Nº de tareas	MOFA	Aleatorio
100	**95,50%**	67,72%
400	**98,14%**	28,68%
700	**99,60%**	21,93%

Tabla 5.2 Hipervolúmenes Cybershake.

5.1.2. **Ligo**

En este caso, el experimento es con la aplicación Ligo con 100 tareas, que recordemos, tiene distintas dependencias de las tareas que la aplicación Cybershake (distinta estructura interna). En estas soluciones (Figura 5.4), vemos que no todas las soluciones que nos proporciona el algoritmo MOFA son aceptables, incluso hay soluciones que son dominadas por la ejecución aleatoria. Pero el cómputo global de soluciones es mejor en el caso del MOFA.

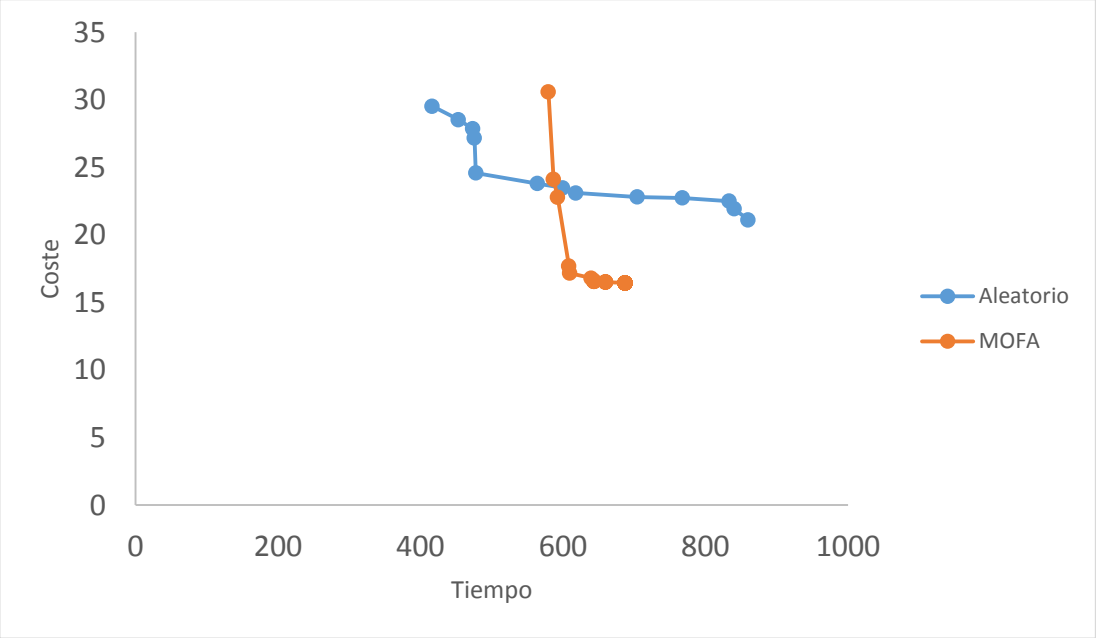

Figura 5.4 Frentes de Pareto mediano de la aplicación Ligo con 100 tareas.

En lo que respecta al hipervolumen (Tabla 5.4), nos proporciona una visión más clara de que conjunto de soluciones es mejor, teniendo un valor de 55,30% para el aleatorio y un valor de **72,95%** para el MOFA.

Obtenemos que el aleatorio tiene una cobertura del 22% de las soluciones de MOFA (Tabla 5.3), mientras que MOFA cubre un **53%** de las soluciones del aleatorio.

En la siguiente ejecución con 400 tareas podemos observar en la Figura 5.5 que, como ocurría con la aplicación Cybershake, también que hay gran diferencia en cuanto a resultados obteniéndose mejores los resultados del algoritmo MOFA.

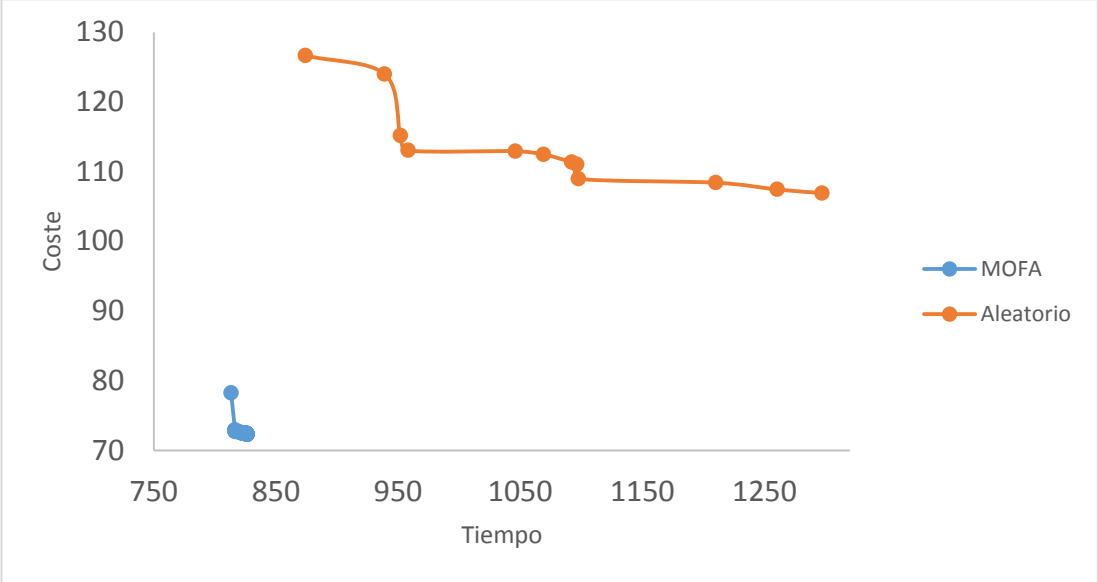

Figura 5.5 Frentes de Pareto mediano de la aplicación Ligo con 400 tareas.

Comparando también los resultados de los hipervolúmenes (Tabla 5.4), obtenemos un valor de 26,22% en el aleatorio y **94,06%** en el MOFA, donde podemos ver gran diferencia de resultados.

En este caso, el MOFA cubre el **100%** todas las soluciones frente al 0% de la ejecución aleatoria aplicando el Operador de Cobertura (Tabla 5.3).

Por último, al igual que en el experimento anterior, obtenemos mejores resultados aplicando el algoritmo MOFA, en este caso, con la aplicación Ligo de 700 tareas, como podemos ver en los frentes de la Figura 5.6.

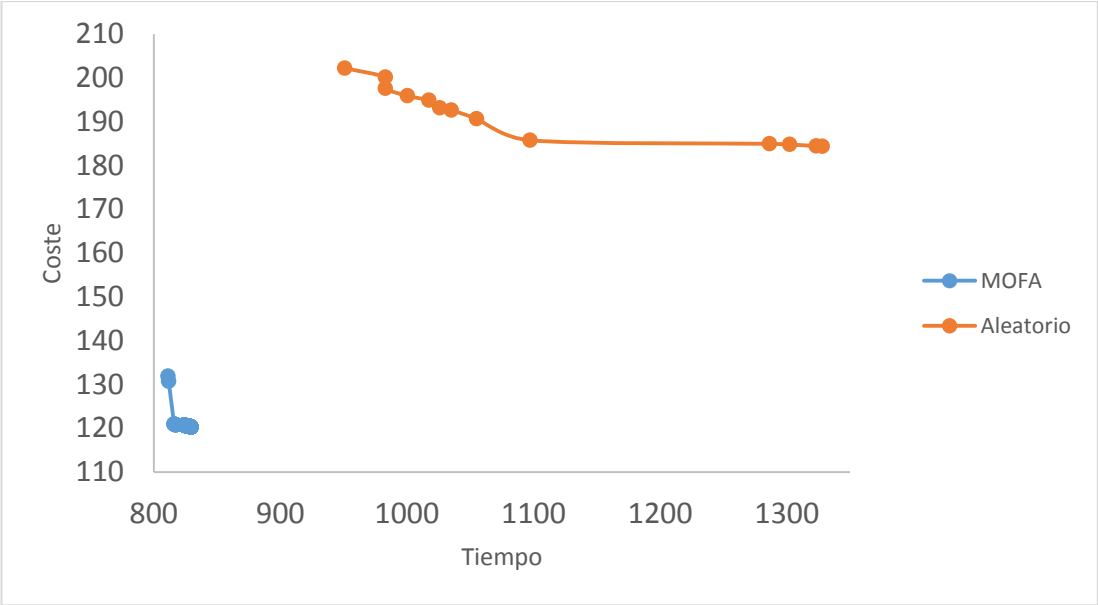

Figura 5.6 Frentes de Pareto mediano de la aplicación Ligo con 700 tareas.

Y si comparamos los hipervolúmenes de ambas ejecuciones (Tabla 5.4), llegamos a la misma conclusión. Obtenemos un hipervolumen de 21,22% con el aleatorio y **93,05%** de hipervolumen con el MOFA.

Al igual que en el anterior, el MOFA domina al **100%** todas las soluciones frente al 0% de la ejecución aleatoria aplicando el Operador de Cobertura (Tabla 5.3).

Podemos decir, al igual que en la aplicación Cybershake, que, al aumentar el tamaño del problema, MOFA se distancia claramente de la aproximación aleatoria, aunque para tamaños pequeños (100 tareas) están más próximos en este caso.

Nº Tareas	MOFA	Aleatorio
100	**53%** cobertura	22% cobertura
400	**100%** cobertura	0% cobertura
700	**100%** coberura	0% cobertura

Tabla 5.3 *Set Coverage* Ligo.

Nº Tareas	MOFA	Aleatorio
100	**72,95%**	55,30%
400	**94,06%**	26,22%
700	**93,05%**	21,22%

Tabla 5.4 Hipervolúmenes Ligo.

5.1.3. **Montage**

Para este experimento hemos utilizado una aplicación de configuración interna distinta, llamado Montage, con 100 tareas, en las que la mayor parte de las

soluciones del MOFA dominan a la ejecución aleatoria, como vemos en los frentes mostrados en la Figura 5.7. El "Coste" en las soluciones del algoritmo comprenden un rango más amplio de soluciones. Aunque la variable "Tiempo" tiene gran diversidad de valores.

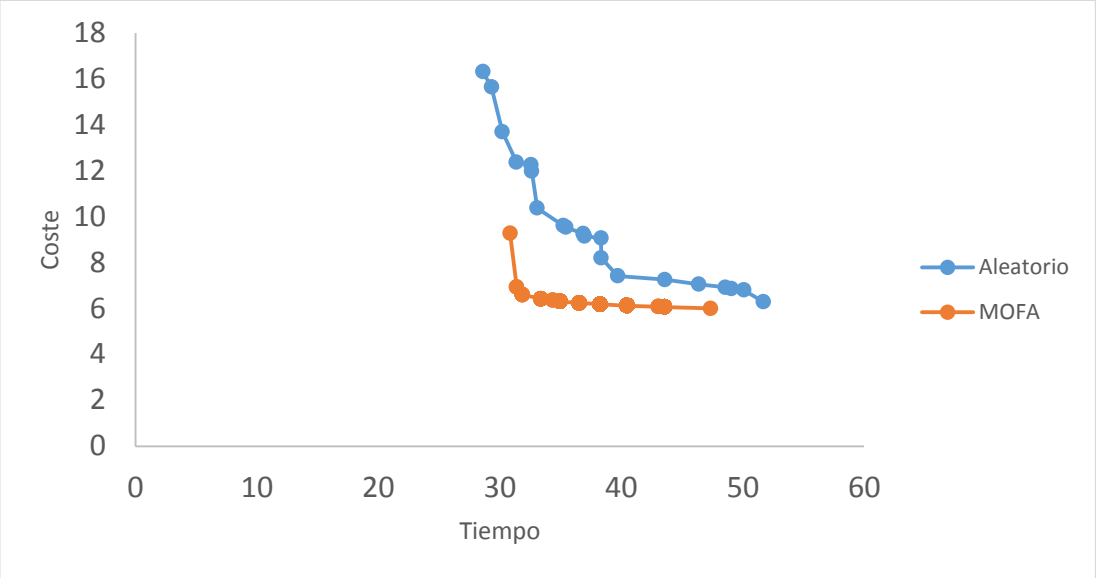

Figura 5.7 Frentes de Pareto mediano de la aplicación Montage con 100 tareas.

Con los hipervolúmenes podemos llegar a la misma conclusión (Tabla 5.6), siendo el MOFA mejor al aleatorio, llegando a un valor de **86,92%** el algoritmo frente al 81,44% del aleatorio.

Cuando aplicamos el operador de cobertura, cuyos valores podemos ver en la Tabla 5.5, vemos que el aleatorio no cubre a ninguna solución del MOFA mientras que éste cubre el **84%** de las soluciones del aleatorio.

A continuación, vemos la ejecución con 400 tareas (Figura 5.8). Como en las aplicaciones anteriores, vuelven a distanciarse los frentes al aumentar el tamaño del problema. En este caso, respecto de la variable "Tiempo" obtenemos valores muy parejos entre el algoritmo MOFA y el aleatorio. Podemos observar que, respecto al "Coste" los valores obtenidos apenas varían, no así en el tiempo, en el que obtenemos un rango más amplio de soluciones. Tras analizar las tablas de datos de los tres clouds utilizados (Tablas 3.1-3.3), nos dimos cuenta de que en todos los casos utilizar una máquina con el doble de capacidad de cómputo, implicaba el doble de coste por unidad de tiempo, pero como el tiempo de ejecución se reducía a la mitad, el coste de alquiler se mantiene igual. En el caso del resto de aplicaciones, este hecho parece que no tiene relevancia, pero en el caso de Montage con número de

tareas medio-alto, parece que sí le influye bastante, y por eso el coste prácticamente no varía.

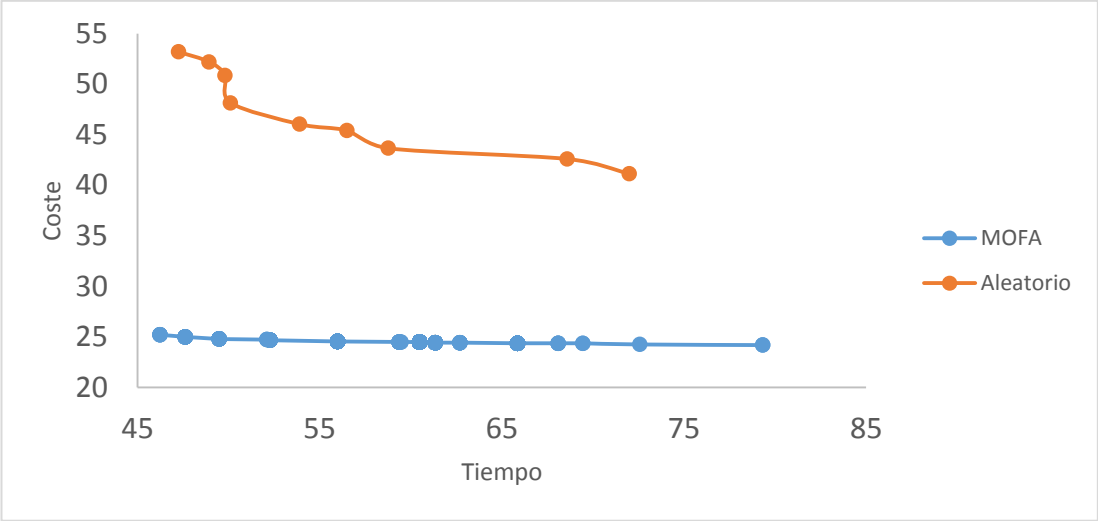

Figura 5.8 Frentes de Pareto mediano de la aplicación Montage con 400 tareas.

Comparando los hipervolúmenes (Tabla 5.6), podemos ver que el aleatorio obtiene un hipervolumen de 55,70% mientras que el MOFA obtiene un resultado de **96,07%**.

En este caso, el MOFA domina al **100%** todas las soluciones frente al 0% de la ejecución aleatoria aplicando el Operador de Cobertura.

A continuación, tratamos con la misma aplicación, pero con tamaño de 700 tareas. Obtenemos un conjunto de soluciones mejor con el algoritmo MOFA, pero con algún punto mejor en la variable "Tiempo" al que nuestro algoritmo no ha encontrado solución (Figura 5.9). Similar a la anterior, el rango de soluciones en la variable "Coste" es muy reducido, y podemos apreciar que el rango de resultados del eje X es mayor.

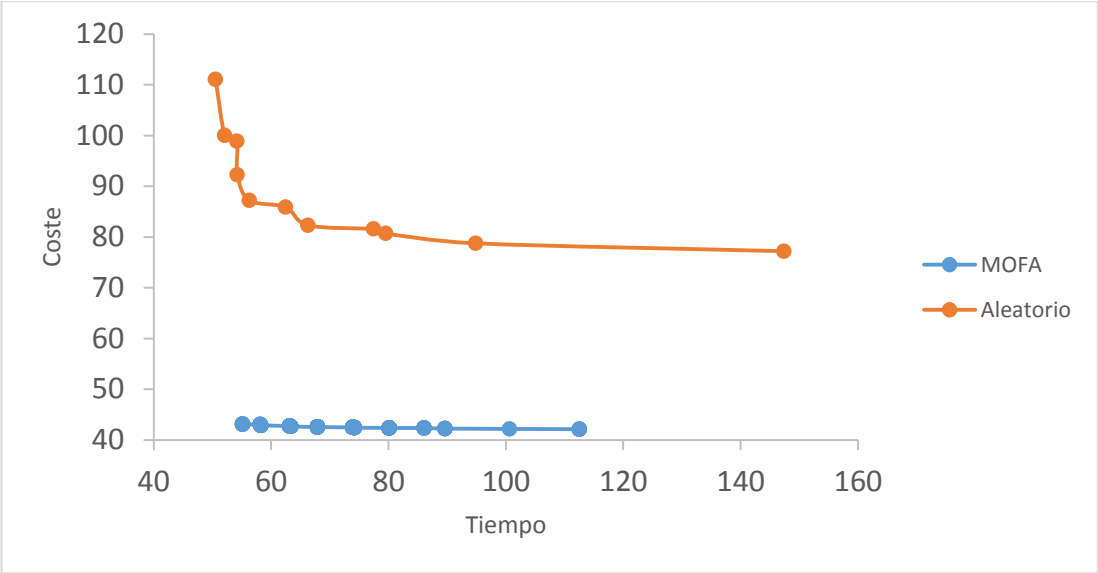

Figura 5.9 Frentes de Pareto mediano de la aplicación Montage con 700 tareas.

Nº tareas	MOFA	Aleatorio
100	**84%** cobertura	0% cobertura
400	**100%** cobertura	0% cobertura
700	**63%** cobertura	0% cobertura

Tabla 5.5 *Set Coverage* Montage.

Nº tareas	MOFA	Aleatorio
100	**86,92%**	81,44%
400	**96,07%**	55,70%
700	**94,48%**	48,06%

Tabla 5.6 Hipervolúmenes Montage.

Como hemos comentado en el gráfico anterior, los hipervolúmenes obtenidos no tendrán una diferencia tan grande como las primeras aplicaciones (Tabla 5.6). Aun así, el algoritmo sigue obteniendo mejores resultados. El hipervolumen del aleatorio es 48,06% mientras que el MOFA obtiene **94,48%**.

Aplicamos el Set Coverage y obtenemos un **63%** de cobertura del MOFA, respecto a un 0% de cobertura del aleatorio.

5.1.4. **Sipht**

En la siguiente ejecución con la aplicación Sipht, observando los frentes (Figura 5.10) podemos tener dudas acerca de qué conjunto de soluciones es mejor, ya que en ambos casos hay varias soluciones dominadas por ambas partes, tanto por el MOFA como por el aleatorio.

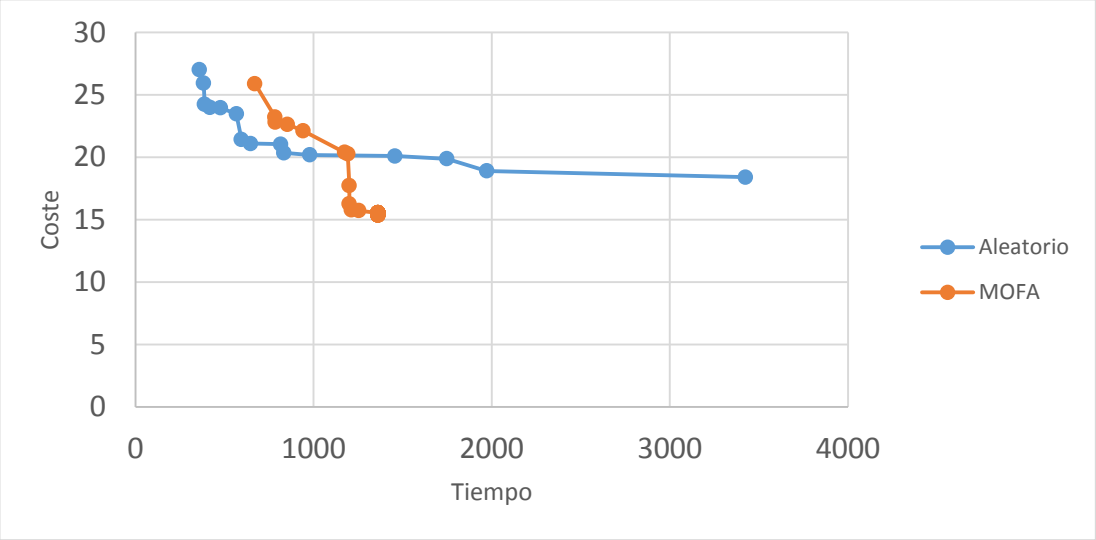

Figura 5.10 Frentes de Pareto mediano de la aplicación Sipht con 100 tareas.

Pero utilizando los hipervolúmenes (Tabla 5.8), llegamos a la conclusión de que el MOFA es mejor con un valor de **80,14%** respecto al valor de 73,16% del aleatorio, disipando así posibles dudas.

Cuando aplicamos el *Set coverage*, nos proporciona una cobertura del **54%** para el aleatorio, mientras que, para el MOFA, nos otorga un porcentaje del 26%. En este caso, la ejecución aleatoria saldría beneficiaría aplicando este operador. Éste es el primer caso en que el algoritmo MOFA no consigue mejorar completamente al aleatorio. Creemos que es debido a las peculiaridades de la aplicación, en la que una tarea es dependiente de muchas y a su vez la tarea final es dependiente de esa tarea (ver Figura 4.2). Parece que, para este tipo de aplicaciones, y teniendo pocas tareas, MOFA no es capaz de conseguir los buenos resultados obtenidos en las aplicaciones anteriores.

Para la aplicación Sipht con 400 tareas (Figura 5.11), obtenemos valores con el MOFA mejores que con el aleatorio, pero podemos comprobar que en la coordenada referente al "Tiempo", obtenemos mejores valores que en el MOFA, penalizando el coste, la otra variable. Nuestro algoritmo no ha llegado a ese conjunto de soluciones.

Aquí, el rango de soluciones tanto de la variable "Coste" como "Tiempo" en el MOFA es mayor que en la aplicación anterior.

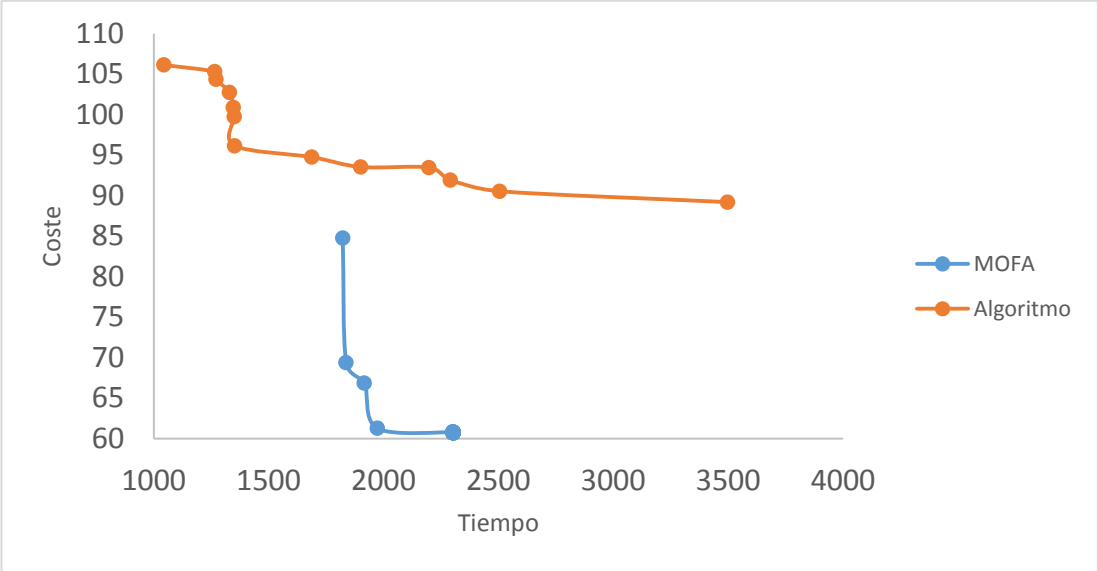

Figura 5.11 Frentes de Pareto mediano de la aplicación Sipht con 400 tareas.

Aunque obtenemos soluciones con el aleatorio que el MOFA no obtiene, los hipervolúmenes obtenidos nos resultan aclaratorios para saber que hay gran diferencia de resultados (Tabla 5.8). El aleatorio obtiene un hipervolumen de 44,17%, mientras que el MOFA obtiene un hipervolumen de **70,93%**, siendo este superior.

Respecto al Operador de Cobertura, obtenemos un 0% de cobertura para la ejecución aleatoria, mientras que un **38%** de cobertura del MOFA. En este caso, el MOFA sí gana al aleatorio.

Al igual que con 400 tareas, obtenemos resultados muy similares con 700 tareas (Figura 5.12), aunque obtenemos valores mejores con el MOFA, pero el aleatorio ha encontrado un rango de soluciones el cual el algoritmo no ha explorado e incluso podemos ver una solución en el MOFA que está dominada por el aleatorio.

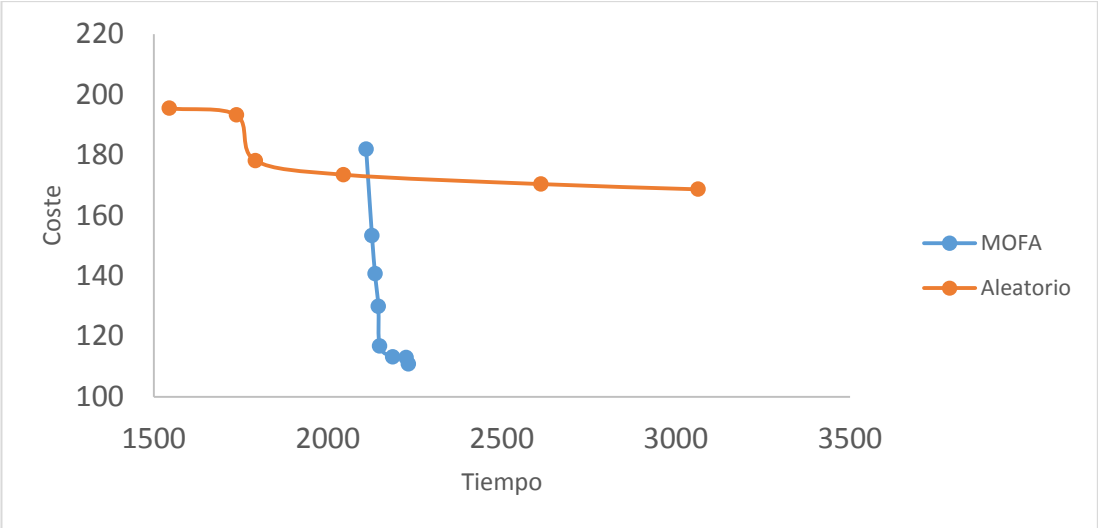

Figura 5.12 Frentes de Pareto mediano de la aplicación Sipht con 700 tareas.

Pero comprobando los hipervolúmenes (Tabla 5.8), observamos que el MOFA sigue siendo mejor respecto al aleatorio, el cual obtiene un resultado de 31,23%, frente al **76,53%** del MOFA.

Por último, aplicando el *Set Coverage* a última ejecución de la aplicación Sipht, obtenemos un 12,5% de cobertura del aleatorio, frente a un **33,3%** de cobertura del MOFA.

Nº Tareas	MOFA	Aleatorio
100	26% cobertura	**54%** cobertura
400	**38%** cobertura	0% cobertura
700	**33,3%** cobertura	12,5% cobertura

Tabla 5.7 *Set Coverage* Sipht.

Nº Tareas	MOFA	Aleatorio
100	80,14%	73,16%
400	70,93%	44,17%
700	76,53%	31,23%

Tabla 5.8 Hipervolúmen Sipht.

5.2 Aleatorio VS MOFA Con Fiabilidad

Para terminar con este capítulo, vamos a realizar un conjunto de experimentos usando diferentes valores de fiabilidad. Para estos resultados, hemos aplicado el

mismo algoritmo que en los resultados anteriores, pero añadiendo la restricción de fiabilidad, de la que hemos hablado anteriormente. Para todos los experimentos, hemos utilizado la aplicación, Montage con un tamaño de 100 tareas.

Fiabilidad	MOFA	Aleatorio
0,2	73% cobertura	12,5% cobertura
0,4	75% cobertura	0% cobertura
0,6	81% cobertura	0% cobertura
0,8	100% cobertura	0% cobertura

Tabla 5.9 *Set Coverage* Fiabilidad.

En la Tabla 5.9 podemos ver los resultados aplicando el Operador de cobertura para los resultados con distintas fiabilidades, y a simple vista, vemos que, en todas las ejecuciones, el resultado es siempre favorable para la ejecución del algoritmo MOFA.

Fiabilidad	MOFA	Aleatorio
0,2	80,81%	74,22%
0,4	84,54%	76,68%
0,6	90,42%	62,94%
0,8	92,34%	58,35%

Tabla 5.10 Hipervolúmenes Fiabilidad.

En la tabla anterior Tabla 5.10, observamos los resultados aplicando el operador multiobjetivo hipervolumen, donde MOFA obtiene los mejores resultados en todos los experimentos. Además, también podemos comprobar que, a mayor restricción, mejor resultado da MOFA en relación al aleatorio.

5.2.2. **Fiabilidad 0,2**

En la Figura 5.13 observamos que los resultados del algoritmo MOFA son mejores que el aleatorio, aunque con soluciones muy parecidas y en las que no hay gran diferencia. En ambos casos existe un conjunto de soluciones que tiene un amplio rango de valores, tanto para la variable “Coste” como para “Tiempo”.

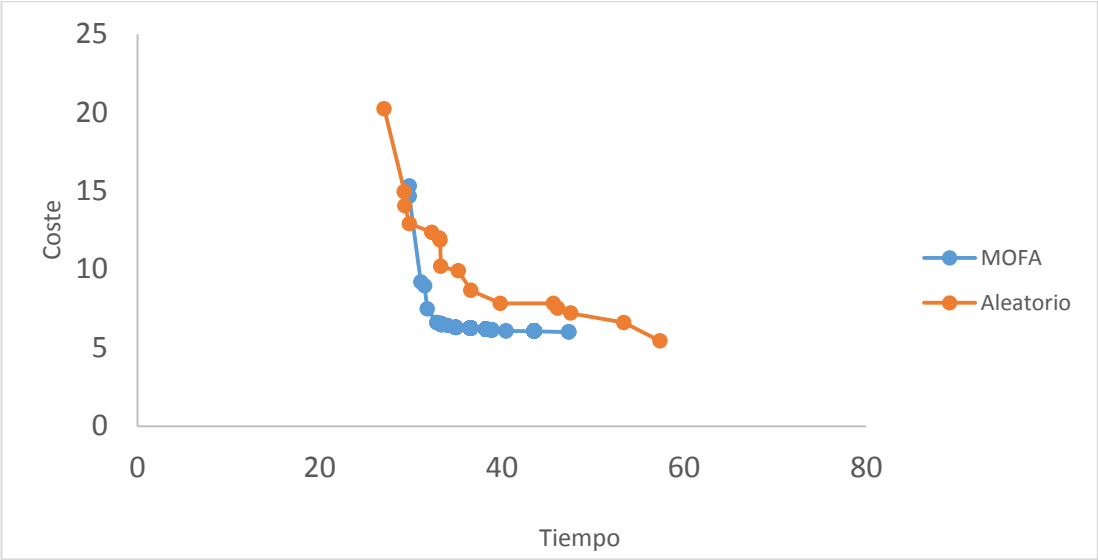

Figura 5.13 Frentes de Pareto mediano con fiabilidad 0,2.

A continuación, podemos observar en los hipervolúmenes (Tabla 5.10) que, efectivamente, los resultados son muy parecidos, donde el aleatorio obtiene un valor de 74,22%, respecto del **80,81%** que obtenemos con el MOFA.

Cuando aplicamos el *Set Coverage* a la siguiente ejecución, podemos ver que el MOFA otorga una cobertura de **73%** frente a un 12,5% del aleatorio.

5.2.3. **Fiabilidad 0,4**

La Figura 5.14 corresponde a la aplicación Montage de 100 tareas con 0,4 de fiabilidad, en la que podemos comprobar que la mayoría de los resultados del experimento aleatorio están dominados por el algoritmo MOFA. Este, a diferencia del caso anterior, no obtiene un rango de soluciones tan amplio en la variable "Coste". Aún así, el conjunto de soluciones es mayor.

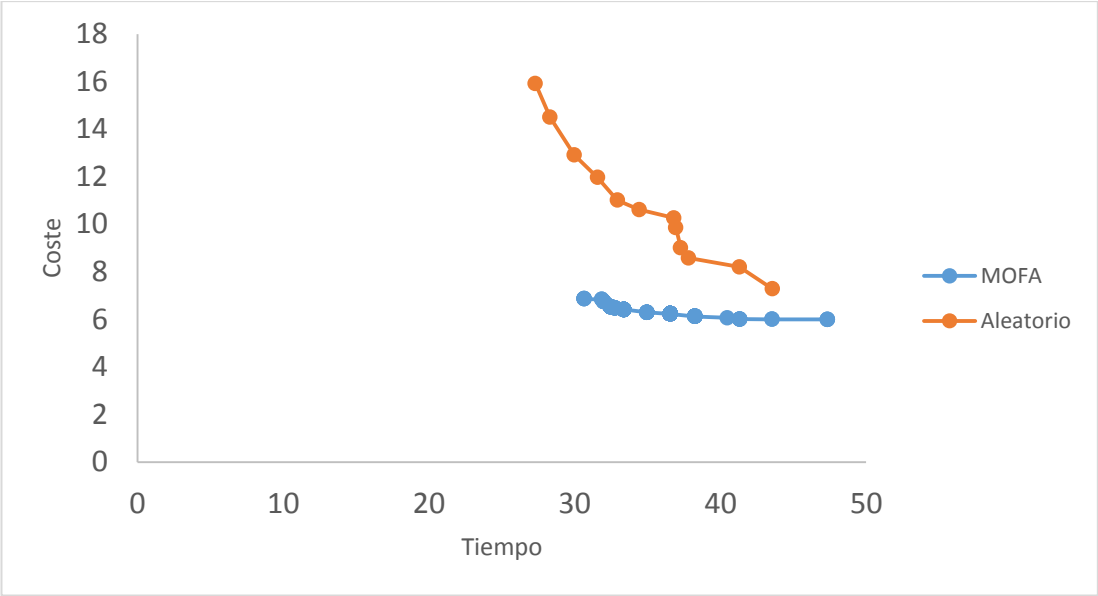

Figura 5.14 Frentes de Pareto mediano con fiabilidad 0,4.

Respecto al hipervolumen (Tabla 5.10), en este caso la diferencia entre ejecuciones es algo mayor, obteniendo el aleatorio un valor de 76,68% respecto de **84,54%** del MOFA.

En este caso, obtenemos que el aleatorio cubre el 0% de las soluciones del MOFA mientras que el MOFA cubre el **75%** de las soluciones del aleatorio.

5.2.4. **Fiabilidad 0,6**

En la ejecución con fiabilidad de 0,6, podemos observar en la Figura 5.15 que, en las soluciones del MOFA, el "Coste" apenas varía, no así el "Tiempo", como ya vimos anteriormente. Esto no ocurre en el aleatorio, que tiene valores más dispersos en ambas variables, aunque la mayoría de las soluciones en ambos casos son peores que las del algoritmo MOFA.

Comparando ambos hipervolúmenes (Tabla 5.10), llegamos a la conclusión de que el algoritmo MOFA es mejor, teniendo un valor de **90,42%**, respecto del 62,94% que tiene el aleatorio. Aquí la diferencia es mayor que en los casos anteriores.

Indicar que obtenemos una cobertura de 0% con la ejecución aleatoria frente al MOFA y una cobertura del **81%** del algoritmo MOFA frente al aleatorio.

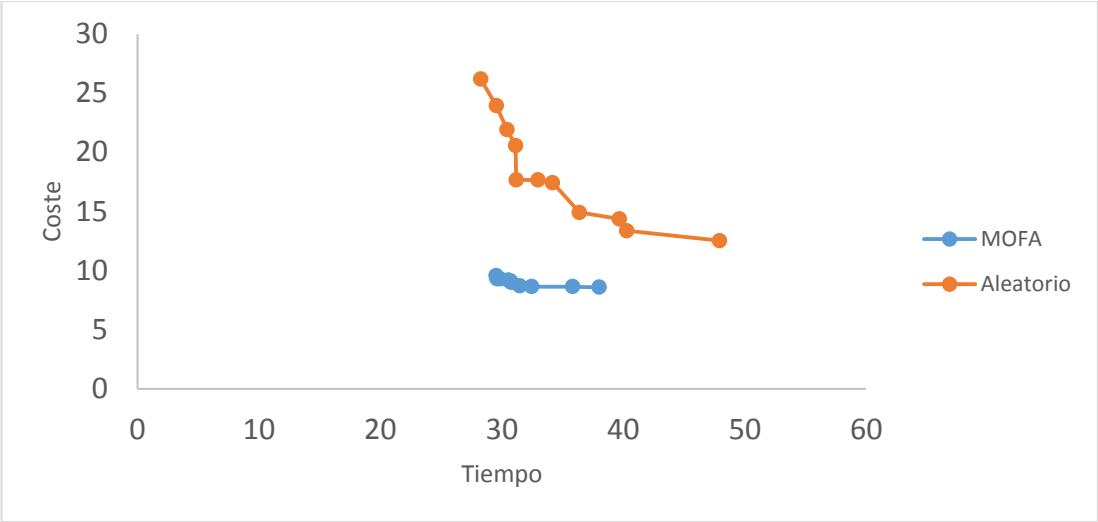

Figura 5.15 Frentes de Pareto mediano con fiabilidad 0,6.

5.2.5. **Fiabilidad 0,8**

Por último, observamos en el gráfico la ejecución con fiabilidad de 0,8, en la que se observa una mayor diferencia de soluciones (Figura 5.16). Todas las soluciones del aleatorio están dominadas por el algoritmo MOFA. Este último tiene las soluciones más cercanas que el aleatorio, que las tiene más dispersas. Hay que tener en cuenta que este valor de fiabilidad es muy restrictivo, de ahí la proximidad de las soluciones.

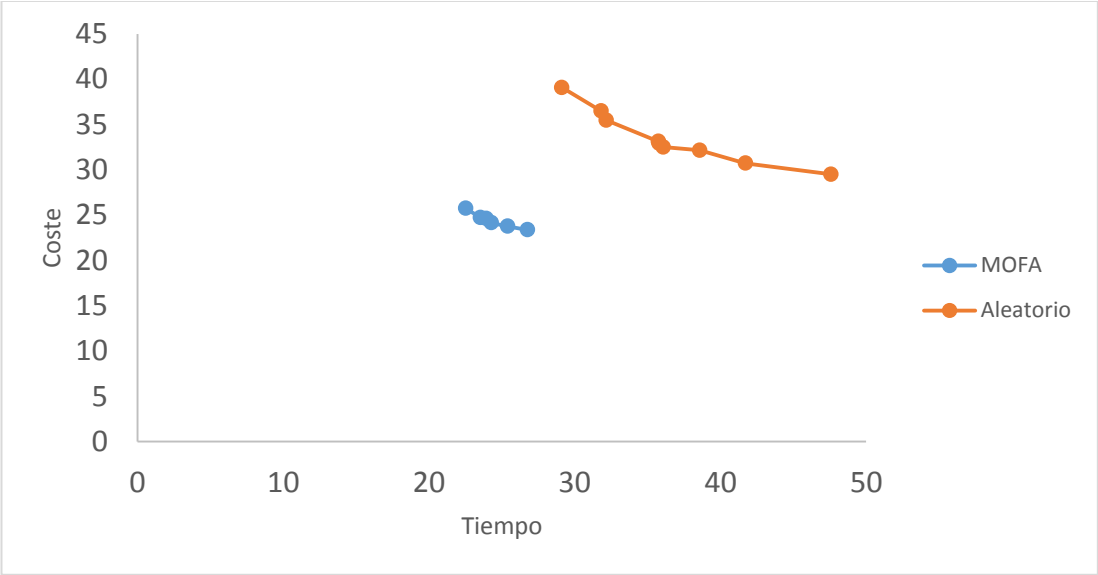

Figura 5.16 Frentes de Pareto mediano con fiabilidad 0,8.

Respecto al hipervolumen (Tabla 5.10), está ejecución es la más clarificadora contando a las anteriores, ya que se encuentra la mayo diferencia entre los hipervolúmenes. El aleatorio obtiene un valor de 58,35%, respecto de **92,34%** del MOFA.

Con el *Set Coverage*, obtenemos un 0% de cobertura del aleatorio frente al MOFA, mientras que el MOFA cubre el **100%** de las soluciones del aleatorio.

6

CONCLUSIONES

Una vez aplicada la solución utilizando el algoritmo multiobjetivo MOFA al problema mencionado anteriormente, podemos llegar a varias conclusiones.

El problema consta de encontrar una solución óptima para la distribución de tareas en un entorno multi-nube. Estas tareas son dependientes unas de otras, esto varía en función de la aplicación a la que pertenece, Cybershake, Ligo, Montage y Sipht.

Para obtener todos los resultados, primero hemos ejecutado las 4 aplicaciones con distinto número de tareas, a saber, 100, 400 y 700, obteniendo diferentes resultados. Esto únicamente para las ejecuciones sin la restricción de fiabilidad, para la que hemos ejecutado únicamente la aplicación Montage con tamaño de tareas de 100 y restricciones de fiabilidad de 0,2, 0,4, 0,6, 0,8. También por ambos casos, hemos lanzado las ejecuciones con el algoritmo, pero con soluciones obtenidas de forma aleatoria.

Como sistema de evaluación, de todas las ejecuciones antes mencionadas, hemos obtenido el valor mediano de las 31 ejecuciones para obtener los frentes de Pareto. Por cada ejecución tanto del aleatorio como del MOFA, sacamos los datos de la ejecución que corresponde con el valor mediano y lo mostramos en el gráfico. Utilizamos también los hipervolúmenes para comparar ambas ejecuciones. Por último, utilizamos el Operador de Cobertura, que aclara aún más la diferencia entre el MOFA y la ejecución aleatoria.

En todas las ejecuciones podemos ver que las ejecuciones del algoritmo MOFA son mejores que el aleatorio. Únicamente en la ejecución de la aplicación Sipht con 100 tareas los dos algoritmos están muy próximos, ganando el MOFA en hipervolumen y el aleatorio en cobertura. También podemos concluir que en todas las aplicaciones aumentar el tamaño del problema hace que las diferencias entre ambos algoritmos aumenten. Respecto a la fiabilidad, vemos como los frentes están más próximos, probablemente debido a que, al aplicar esta restricción y hacerla cada vez más fuerte, el espacio de soluciones se reduce considerablemente (una solución solo es válida si cumple la restricción), pero, aun así, MOFA consigue mejorar los resultados del aleatorio tanto en hipervolumen como en cobertura.

Por último, como hemos visto, no solamente el tamaño del problema afecta al resultado, también la estructura interna de las aplicaciones. Concretamente, para la aplicación Sipht con una estructura interna bastante peculiar, la diferencia entre

ambos algoritmos se acorta, llegando a no quedar claro qué algoritmo es mejor cuando dicha aplicación tiene pocas tareas.

Por tanto, tras los resultados obtenidos, podemos concluir que el algoritmo MOFA se comporta bastante bien en el problema de planificación de flujos de trabajo en sistemas multicloud.

Como posibles líneas futuras, podrían estudiarse modificaciones al operador de atracción con el fin de mejorar las soluciones obtenidas o implementar otros algoritmos genéticos.

6.1 Aporte Personal

Una vez finalizado este proyecto, son varios puntos que me han ayudado a comprender cómo se puede aplicar a un problema un algoritmo para obtener resultados aceptables en un tiempo determinado. También he tomado conciencia de la importancia del código óptimo y bueno, ya que, en algunos casos de ejecución, los resultados han tardado varios días, por lo que es necesario ser preciso a la hora construir una aplicación de calidad, para reducir lo máximo posible esto.

Además, gracias al trabajo, sé cómo comparar dos resultados válidos para conocer cuál es el más acertado, por el hipervolumen y el Operador de Cobertura.

En lo referente a la inteligencia en enjambre, he utilizado un algoritmo nuevo que no conocía, y he descubierto su potencial para obtener soluciones válidas en un plazo corto de tiempo. Ya había trabajado anteriormente con otros como el ACO o el ABC. En definitiva, he reforzado mis conocimientos en inteligencia de enjambre añadiendo un algoritmo más.

BIBLIOGRAFÍA

Aslam, S. & Islam, S. u., 15 December 2017. Information collection centric techniques for cloud resource management: Taxonomy, analysis and challenges. Volume 100, pp. 80-94.

Beni, G. et al., n.d. *www.ecured.cu.* [Online]
Available at: https://www.ecured.cu/Inteligencia_de_enjambre
[Accessed 4 Enero 2019].

Beume, N. et al., 2009. On the Complexity of Computing the Hypervolume Indicator.. *IEEE Transactions on Evolutionary Computation,* Volume 13, pp. 1075-1082.

Caparrini, F. S., 2016. *www.cs.us.es.* [Online]
Available at: http://www.cs.us.es/~fsancho/?e=70
[Accessed 5 Enero 2019].

Caparrini, F. S., 2018. *www.cs.us.es.* [Online]
Available at: http://www.cs.us.es/~fsancho/?e=71
[Accessed 10 Enero 2019].

Deb, K., Pratap, A., Agarwal, S. & Meyarivan, T., 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation,* 6(2), pp. 182-197.

Hidalgo-Paniagua, A., Vega-Rodríguez, M. A., Ferruz, J. & Pavón, N., 2017. Solving the multi-objective path planning problem in mobile robotics with a firefly-based approach. *Soft Computing,* 21(4), p. 949.

Hu, H. et al., 15 July 2018. Multi-objective scheduling for scientific workflow in multicloud environment. *Network and Computer Applications,* Volume 114, pp. 108-122.

Karaboga, D., 2010. *www.scholarpedia.org.* [Online]
Available at: http://www.scholarpedia.org/article/Artificial_bee_colony_algorithm
[Accessed 15 Enero 2019].

Madni, H. H., 2016. Resource scheduling for infrastructure as a service (IaaS) in cloud computing: Challenges and opportunities. Volume 68, pp. 173-200.

Onwubolu, G. & Davendra, D., 2009. *Differential Evolution: A Handbook for GlobalPermutation-Based Combinatorial Optimization.* primera ed. s.l.:Springer-Verlag Berlin Heidelberg.

Sooezi, N., Abrishami, S. & Lotfian, M., 30 Nov.-3 Dec. 2015. *Scheduling Data-Driven Workflows in Multi-cloud Environment.* Vancouver, BC, Canada, IEEE.

Tr4nsduc7or, 2015. *https://robologs.net/.* [Online]
Available at: https://robologs.net/2015/08/28/como-programar-un-algoritmo-genetico-parte-i-in-theory/
[Accessed 2 Enero 2019].

X.-S. Yang, 2009. Firefly Algorithms for Multimodal Optimization. *Stochastic Algorithms: Foundations and Applications,* Volume 5792, pp. 169-178.

Zitzler, E. et al., 2003. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation,* Volume 7, pp. 117-132.