INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO

Flujos de trabajo intensivos en datos para cómputo en la nube

TESIS
QUE PARA OBTENER EL TÍTULO DE
MAESTRO EN CIENCIAS EN COMPUTACIÓN

P R E S E N T A

FERNANDO AGUILAR REYES

ASESOR: **DR. JOSÉ OCTAVIO GUTIÉRREZ GARCÍA**

MÉXICO, D.F. **2016**

Con fundamento en los artículos 21 y 27 de la Ley Federal del Derecho de Autor y como titular de los derechos moral y patrimonial de la obra titulada "Flujos de trabajo intensivos en datos para cómputo en la nube", otorgo de manera gratuita y permanente al Instituto Tecnológico Autónomo de México y a la Biblioteca Raúl Bailléres Jr., autorización para que fijen la obra en cualquier medio, incluido el electrónico, y la divulguen entre sus usuarios, profesores, estudiantes o terceras personas, sin que pueda percibir por tal divulgación una contraprestación.

Fernando Aguilar Reyes

_____

Fecha

_____

Firma

Resumen

En este trabajo se desarrolló un algoritmo de planficación de flujos de trabajo con enfoque a entornos de cómputo en la nube, con la capacidad de calcular el numero de maquinas necesarias para reducir el tiempo de espera de tareas por recursos disponibles. Tambien, se implementó este algoritmo en un software simulador de flujos de trabajo. Los resultados experimentales muestran que el algoritmo puede optimizar los costos de ejecución de mejor manera que los algoritmos MinMin y Miope. Por ultimo, se implementó un sistema administrador de flujos de trabajo en la nube, llamado sweeper, con el objetivo de explorar cuáles son los retos a los que hay que enfrentarse para llevar este tipo de algoritmos de planifcación a entornos reales.

Palabras clave: Planificación de flujos de trabajo, Programación dinámica, Optimización de costos

Abstract

In this work we developed a workflow scheduling algorithm with a focus on cloud computing environments. This novel algorithm can computethe number of machines required to reduce task waiting time for available resources. Also, this algorithm was implemented in a workflow simulator software. The experimental results show that the algorithm can optimize execution costs better than the MinMin and Myopic algorithms. Finally, a cloud workflow manager system, called sweeper, was developed, with the goal of exploring what are the challenges we need to face in order to use those workflow scheduling algorithms on real life scenarios.

Keywords: Workflow scheduling, Dynamic programming, Cost optimizacion

Agradecimientos

A Marisela, por apoyarme en todo momento para hacer de la maestría todo un éxito.

Al profesor Octavio Gutiérrez, por sus sabios consejos.

A Fernando Esponda y a Víctor González, por otorgarme la opotunidad de estudiar la maestria.

A mi familia, Miguel y Florinda, Rodrigo y Mike.

A Propiedades.com, por facilitar los recursos tecnológicos para realizar este trabajo.

Índice general

Índice de figuras

Lista de Códigos

Capítulo 1

Introducción

El mundo digital dominado por datos requiere la utilización de algoritmos, tecnologías y mecanismos que permitan orquestar procesamientos de grandes volúmenes de datos, los cuales pueden tardar horas e, incluso, días en completarse. Aún así, el diseño de la aplicación junto con el flujo de trabajo que modela las tareas llevadas a cabo por la aplicación, requiere un esfuerzo considerable.

Ahora bien, sería deseable desarrollar una plataforma que permita distribuir estos flujos de trabajo computacionalmente intensivos en un sistema distribuído con el fin de disminuir el tiempo de ejecución del flujo. También, es deseable disminuir el presupuesto utilizado, ya que en la actualidad, es muy común utilizar servicios de cómputo en la nube, en el cual se paga por este servicio de acuerdo a un modelo económico en el que se paga sólo por el servicio utilizado.

En este trabajo se muestra el desarrollo de esta plataforma de ejecución de flujos de trabajo intensivo en datos con aplicación en cómputo en la nube. También se muestra el marco teórico necesario para el desarrollo de este proyecto.

Primero, empezaremos por describir el modelo de costos de cómputo en la nube, marcado por acuerdos de nivel de servicio, y se describirá un mecanismo para poder estimar el tiempo de ejecución del flujo de trabajo, necesario para poder estimar el prespuesto utilizado para ejecutar el flujo de trabajo.

1.1. Descripción del problema

Se tiene un flujo de trabajo definido por $W = (T, D)$, donde T es el conjunto de tareas del flujo de trabajo y D es el conjunto de dependencias que existen entre las tareas del flujo de trabajo. Se desea planificar las tareas en un conjunto de recursos R en la nube de diferente capacidad, los cuales están conectados entre sí por medio de una red. Se tiene una función $F : W \times R \mapsto (\mathcal{R}^+ \times R^+)$ la cual asocia un costo y un tiempo total de ejecución a una planificación de este flujo de trabajo con el conjunto de recursos dado. Se requiere minimizar la función F tanto en costo como

en tiempo. Cabe aclarar que, a diferencia del problema clásico de planificación de flujos de trabajo donde se tiene un conjunto *fijo* de recursos, en entornos de computo en la nube, se pueden solicitar los recursos necesarios para ejecutar el flujo de trabajo.

1.2. Lineamientos de diseño

Desarrollar un sistema de administración de flujos de trabajo conlleva varias decisiones de diseño, por lo que . Sin embargo, se pueden ver algunos patrones de uso o generalidades que quisiéramos observar en el sistema administrador de flujos de trabajo. A continuación, se listarán los lineamentos de diseño deseables en el sistema administrador de flujos de trabajo.

1.2.1. El flujo como centro de la aplicación

Los sistemas de cómputo distribuido tienen inherentes una complexidad que crece más conforme mayores capacidades de agregan a éstos. Los científicos y personas especialistas de algún dominio de conocimiento cada vez tienen menos tiempo para aprender los detalles del funcionamiento de estos sistemas distribuidos. Por otro lado, dibujar diagramas que describan la secuencia de ejecución de los programas es más intuitivo. Y un flujo de trabajo bien puede expresarse a través de estos diagramas. De esta forma, especificar el flujo de trabajo es el pilar que describe los procesos a ejecutar. Por otro lado, el flujo de trabajo aporta información que ayuda a la paralelización y ejecución distribuida del mismo.

1.2.2. Orientado a nubes

Aunque se pueden ejecutar Open Grid Scheduler, HTCondor, programas MPI, aplicaciones basadas en MapReduce y cómputo distribuido basado en grafos en la nube, éstos funcionan muy bien en entornos de una sola computadora con gran capacidad o en grids institucionales. Sin embargo, cuando se utilizan estos programas en enfoques como la nube, surgen algunas eventualidades, a saber: 1) es más difícil supervisar estos entornos por el hecho de la insfraestrucutra no es *on-premise* o, en el caso de grids comunitarios, se conoce a ciencia cierta el funcionamiento del sistema y, 2) hay un costo monetario por utilizar estos recursos que está explícitamente establecido, el cual es deseable minimizar. Además, otra característica importante de la nube, a saber, la utilización de máquinas virutales, hace que el rendimiento del sistema distribuido en general pueda escalarse prácticamente sin límites. Por ello, es deseable ejecutar el flujo de trabajo en el menor tiempo posible, respetando un presupuesto asignado. Esta flexibilidad sólo es posible encontrarla actualmente en las infraestructuras de servicios de cómputo en la nube. Por esta razón, el sistema administrador de flujos de trabajo tiene que estar fuertemente orientado a trabajar en la nube.

1.2.3. Intensivo en datos

Como se ha dicho en la introducción, las cargas de trabajo que administran estos sistemas usualmente generan enormes cantidades de datos, que van desde logs de operación, archivos intermedios de resultados hasta volcados de memoria en caso de que ocurra algún error. Así, manejar datos es primordial en nuestra solucíón. Este requerimiento se puede cumplir utilizando sistemas de archivos distribuidos con énfasis en la tasa de rendimiento de datos guardados.

1.2.4. Portable entre nubes

El hecho de tener la característica de ser portable entre nubes permite la flexibilidad de escoger la plataforma de cómputo en la nube conveniente a las necesidades de ejecución del flujo de trabajo. Así, es deseable poder escoger ejecutar el flujo de trabajo en Amazon Web Servies, Rackspace o Microsoft Azure (por citar algunos ejemplos), dependiendo de la infraestructura necesaria o prespuesto asignado.

1.2.5. Simple

Si bien, configurar un flujo de trabajo requiere conocimiento técnico del entorno de cómputo en donde se va a ejecutar, hay ciertos detalles que se pueden abstraer para reducir la carga cognitiva del usuario, de tal modo que éste se enfoque en modelar los pasos, las dependencias de los pasos y las restricciones. De esta forma, el sistema administrador de flujos de trabajo se encargará de organizar y planificar las tareas para cumplir con las restricciones impuestas por el usuarios.

1.2.6. Basado en estándares abiertos

Utilizar estándares hace que 1) la curva de aprendizaje para utilizar sistemas de administración de flujos de trabajo sea más suave y 2) se pueda adaptar a soluciones ya existentes. También, el uso de estándares abiertos promueve mantener a la vanguardia la infraestructura tecnológica para ejecutar el flujo de trabajo.

1.3. Objetivo

Construir un sistema administrador de flujos de trabajo que esté orientado a utilizar el enfoque de cómputo en la nube.

1.4. Organización del documento

En el capítulo 2, se revisa la teoría de planificación necesaria para construir un algoritmo acorde al objetivo planteado. Luego, en el capítulo 3 se describe el algoritmo para planficar tareas de un flujo de trabajo en entornos de cómputo en la nube, que

es la parte esencial para optimizar la ejecución de los flujos de trabajo. Además, en el capítulo 4 se describe la arquitectura y funcioxiento de Sweeper, el sistema de administrador de flujos de trabajo desarrollado en esta tesis. En el capítulo Finalmente, en el capítulo 6 presentamos conclusiones.

Capítulo 2

Teoría de planificación

2.1. Introducción

Cada día utilizamos aplicaciones de cómputo más complejas que resuelven problemas más complicados que involucran el procesamiento de grandes volúmenes de datos o contenido multimedia, demandando cada vez más poder de cómputo. Por ejemplo, hay instituciones financieras que necesitan procesar millones de transacciones diariamente; así, utilizan aplicaciones que durante el día guardan las operaciones bancarias y en la noche ejecutan estas operaciones en lotes para actualizar las cuentas bancarias. En la Bolsa Mexicana de Valores, el motor de negociación transaccional, MoNeT, puede procesar hasta 100,000 transacciones por segundo [5]. Un último ejemplo son los proyectos de cómputo científico: éstos requieren hacer numerosos cálculos para llegar a resultados pertinentes. Tal es el caso del descubrimiento del bosón de Higgs en el Gran Colisionador de Hadrones (LHC) de la Organización Europea para la Investigación Nuclear. Se estima que cada año, el detector principal del LHC genera 15 petabytes de datos que requieren ser analizados [18].

En cualquiera de los casos anteriores, es deseable distribuir la ejecución de estos flujos de trabajo entre varios recursos computacionales. Si bien es posible paralelizar algunos pasos de la ejecución de los flujos de trabajo, hay restricciones de orden que se deben respetar, por lo cual, es indispensable *planificar* la ejecución del flujo de trabajo entre los múltiples recursos computacionales.

Definimos la *planificación* como una función que asigna a cada tarea del flujo de trabajo un servicio que contiene los recursos para ejecutar dicha tarea, con el fin de completar la ejecución de todas las tareas de manera satisfactoria, cumpliendo ciertas restricciones [23], por ejemplo, restricciones de orden de ejecución. Con ello, se desea encontrar una forma óptima de hacer esta planificación para reducir el tiempo de ejecución total del flujo de trabajo. Sin embargo, con la aparición del cómputo en la nube, es posible ejecutar nuestro flujo de trabajo con otras restricciones, como minimizar el presupuesto necesario para la ejecución del flujo afectando el tiempo de

ejecución.

Sin embargo, aún no existe un consenso general sobre cuál es una definición completa de un flujo de trabajo [21], debido a que los sistemas que administran y ejecutan las tareas de un flujo de trabajo utilizan especificaciones diferentes para expresar el flujo de trabajo. También, esta falta de consenso da lugar a que un flujo de trabajo pueda ser interpretado desde varias perspectivas, es decir, un flujo de trabajo puede representar dependencias de datos o dependencias de orden entre las tareas. Así, es necesario establecer una base para representar los flujos de trabajo, y con ello, diseñar algoritmos de planificación de flujos de trabajo que puedan ser utilizados en ambientes de cómputo distribuidos. Finalmente, es deseable que estos algoritmos de planificación optimicen el tiempo de ejecución total del flujo de trabajo o que ajusten la ejecución del flujo a un limitado presupuesto de recursos.

2.2. Flujos de trabajo y el problema de planificación

La *planificación* es el proceso de asignación de recursos a tareas, de tal modo que se define un orden de ejecución de las tareas, teniendo lugar diferentes combinaciones de recursos y tareas. Primero se definirá de la forma más elemental el problema de la planificación para estudiar sus propiedades y con ello, explicar la relación de este problema fundamental con el problema de planificación de flujos de trabajo. Esto nos permitirá enunciar una propiedad importante del problema de planificación de flujos de trabajo, a saber: este problema pertenece a la clase de complejidad NP-completo.

2.2.1. El problema fundamental de la planificación

De acuerdo a Ullman et al. [20], el *problema fundamental de la planificación* está compuesto por:

1. Un conjunto de tareas $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$
2. Un ordenamiento parcial $\prec$ sobre $\mathcal{J}$
3. Una función de costo $U : \mathcal{J} \mapsto \mathbb{Z}^+$, la cual indica el tiempo que tarda en completarse cada una de las tareas de $\mathcal{J}$
4. Un número de computadoras (procesadores) k

En el trabajo de Ullman et al. [20], el objetivo del problema fundamental de la planificación es *minimizar* el tiempo total de ejecución, denotado por t_{max}, respetando el orden parcial definido por $\prec$, asignando tareas de $\mathcal{J}$ a los k procesadores. También, es importante notar que se asume que cuando una computadora ejecuta una tarea, ésta es ejecutada completamente, sin errores o interrupciones.

Como puede notarse, hay varias maneras de acomodar las k computadoras para que se ejecuten todas las tareas de $\mathcal{J}$. De hecho, Ullman et al. han demostrado que este problema pertenece a la clase de complejidad NP-completo [20]. Esto significa que no

se ha encontrado un algoritmo que pueda resolver el problema en tiempo polinomial. Entonces, la solución ingenua de probar ordenadamente todas las posibles asignaciones de tareas a computadoras resulta computacionalmente muy costosa.

2.2.2. Planificación de flujos de trabajo

Para plantear el problema de la planificación de flujos de trabajo, primero se mostrará que, bajo ciertas condiciones, un flujo de trabajo puede ser reducido a un grafo dirigido acíclico haciendo las transformaciones adecuadas. Luego, se utilizará la definición del problema básico de planificación con restricciones y su semejanza para flujos de trabajo. Finalmente, se hará una descripción de la complejidad de planificar flujos de trabajo.

Reducción de flujos de trabajo a grafos dirigidos acíclicos.

En el trabajo de Mair et al. [9] se propone un formato para una representación intermedia de flujos de trabajo basada en grafos dirigidos acíclicos, con el fin de transformar una especificación detallada de un flujo de trabajo, escrita en un lenguaje basado en XML llamado AGWL, a esta representación intermedia. De acuerdo a lo discutido anteriormente en la sección 2.1 de este documento, un flujo de trabajo puede ser interpretado desde varias perspectivas. Sin embargo, un grafo dirigido acíclico resume en pocos elementos (nodos y aristas) las tareas y las dependencias que deben ser cumplidas durante la planificación. Es por esta razón que es deseable reducir un flujo de trabajo, interpretado desde cualquier perspectiva, a un grafo dirigido acíclico.

Además, aún no existe un consenso sobre qué representa un flujo de trabajo [21], debido a las diferentes perspectivas de su interpretación. En el caso del lenguaje AGWL, éste cuenta con facilidades para poder expresar flujos de trabajo condicionales, a saber: construcciones `if`, `while`, `for` y `parallel`. Estas facilidades permiten expresar una gran variedad de flujos de trabajo. Sin embargo, para poder estudiar los flujos de trabajo con estas construcciones, se requieren de mecanismos de predicción o estimación de posibles rutas de ejecución del flujo de trabajo. Por esta razón, el estudio de estos mecanismos está fuera del alcance de este trabajo, restringiendo el estudio a flujos de trabajo con ejecución atómica (sin fallos) y sin estructuras condicionales.

Definición del problema de planificación de flujos de trabajo.

Una vez que se ha establecido a los grafos dirigidos acíclicos como nuestra representación básica de flujos de trabajo, se definirá el problema que conlleva asignar recursos a las tareas del flujo. Con el fin de no perder generalidad, tomaremos las definiciones de Wieczorek-Prodan [22] para definir el problema de la planificación de flujos de trabajo:

Definición 1. *Un **flujo de trabajo** es un grafo dirigido acíclico $w \in \mathcal{W}, w = (\mathcal{V}, \mathcal{E})$, compuesto de un conjunto de nodos $\mathcal{V}$ que representan tareas $\tau \in \mathcal{T}$ y un conjunto de aristas $\mathcal{E}$ que representan transferencias de datos $\rho \in \mathcal{D}$.*

Hay que notar en la definición anterior que la forma en que se relacionan las tareas y las transferencias de datos con los nodos y aristas del grafo está determinada por la representación del flujo de trabajo, es decir, las aristas representan tanto dependencias de datos como restricciones de orden de las tareas. Ahora se definirán los recursos de cómputo en donde se ejecuta el flujo.

Definición 2. *Un **servicio** es una entidad de cómputo que puede ejecutar una tarea $\tau \in \mathcal{T}$. El conjunto de todos los servicios disponibles para ejecutar el flujo de trabajo está denotado por $\mathcal{S}$.*

De este modo, se puede definir la planificación como una regla de asociación entre servicios y tareas del flujo de trabajo:

Definición 3. *La **planificación de un flujo de trabajo** w es una función $f : \mathcal{T} \mapsto \mathcal{S}$ que asigna servicios a las tareas del flujo. El conjunto que contiene todas las posibles planificaciones de w es denotado por $\mathcal{F}$.*

Cada posible planificación determina un costo de ejecución. A continuación, definiremos el modelo de costos determinado por la utilización de los servicios.

Definición 4. *Un **modelo de costos** $C = \{c_1, \ldots, c_n\}$ es un conjunto de criterios que determinan las restricciones en las que se debe ejecutar una planificación, por ejemplo, un límite en el tiempo de ejecución, en el costo monetario o la tolerancia a fallos, entre otros.*

Definición 5. *Para cada criterio $c_i \in C$, existe una **función de costo parcial** $\Theta_i : \mathcal{S} \mapsto \mathbb{R}$, en la que a cada servicio disponible para ejecutar el flujo de trabajo se le asocia un costo por ejecutar dicho servicio con las restricciones dictadas con el criterio c_i.*

Las funciones de costo parcial son útiles para determinar costos a nivel de tareas; es decir, con una función de costo parcial podemos definir el tiempo que tomará ejecutar una tarea en un recurso determinado o el costo monetario de ejecutar la tarea en dicho recurso.

Definición 6. *Para cada criterio $c_i \in C$, existe una **función de costo total** $\Delta_i : \mathcal{W} \times \mathcal{F} \mapsto \mathbb{R}$, que asigna un flujo de trabajo w planificado por f un costo basado en los costos parciales determinados por los servicios utlizados para ejecutar las tareas del flujo.*

Por lo tanto, el objetivo del problema de la planificación de flujos de trabajo es encontrar la planificación f que minimice las funciones de costo total Δ_i, $1 \leq i \leq n$.

Complejidad computacional de la planificación de flujos de trabajo.

Ahora, se hará una equivalencia con el problema fundamental de planificación visto anteriormente para estudiar su complejidad computacional. La equivalencia es la siguiente:

1. El conjunto de tareas $\mathcal{J}$ equivale a las tareas $\mathcal{T}$ descritas por el flujo de trabajo.

2. El ordenamiento parcial $\prec$ está representado tanto por las dependencias de datos $\mathcal{D}$ y las aristas $\mathcal{E}$ que representan el control de flujo que deben respetarse para ejecutar el flujo.

3. Las k computadoras equivalen al conjunto de servicios $\mathcal{S}$ disponibles para ejecutar el flujo.

4. La función de costo U tiene una equivalencia implícita. El problema fundamental asume que conocemos apriori el tiempo de ejecución de una tarea. Sin embargo, es muy frecuente que sólo tengamos una estimación del tiempo de ejecución. Por otro lado, podemos establecer una función auxiliar que relacione las tareas con los servicios. Dicha relación es la función de planificación f. En efecto, ejecutar una tarea en un servicio genera un costo, determinado por las funciones parciales y totales. Así, estableceremos una relación proporcional entre costo y tiempo, con el fin de mostrar que existe una equivalencia entre la función de costo W del problema fundamental y el modelo de costo de un flujo de trabajo.

Con lo anterior, se ha mostrado que el problema de la planificación de flujos de trabajo es equivalente al problema fundamental de planificación. Entonces, se puede inferir que el problema de la planificación de flujos de trabajo pertenece a la clase de complejidad NP-completo.

2.3. Taxonomía de los algoritmos de planificación de flujos de trabajo

Como han aparecido numerosos algoritmos para planificar flujos de trabajo, es natural que surjan clasificaciones [19] [27] para tener una mejor idea de los avances logrados en este tema y los puntos clave con los que trabajan los algoritmos.

De acuerdo a Yu et al. [27], se pueden clasificar los algoritmos de planificación de flujos de trabajo ejecutados en grids en dos grandes niveles: 1) los algoritmos de mejor esfuerzo y 2) los algoritmos de calidad en el servicio. Al primer grupo pertenecen aquellos algoritmos que tratan de minimizar el tiempo total de ejecución[1], haciendo uso de todos los recursos disponibles. En el segundo grupo, los algoritmos tratan de obtener una planificación que cumpla las restricciones especificadas como una medida de calidad, con la posibilidad de elegir soluciones que tomen un tiempo de ejecución subóptimo. A su vez, cada grupo tiene ramas de clasificación. En la figura 2.1 se puede visualizar la taxonomía.

[1] También conocido como *makespan*

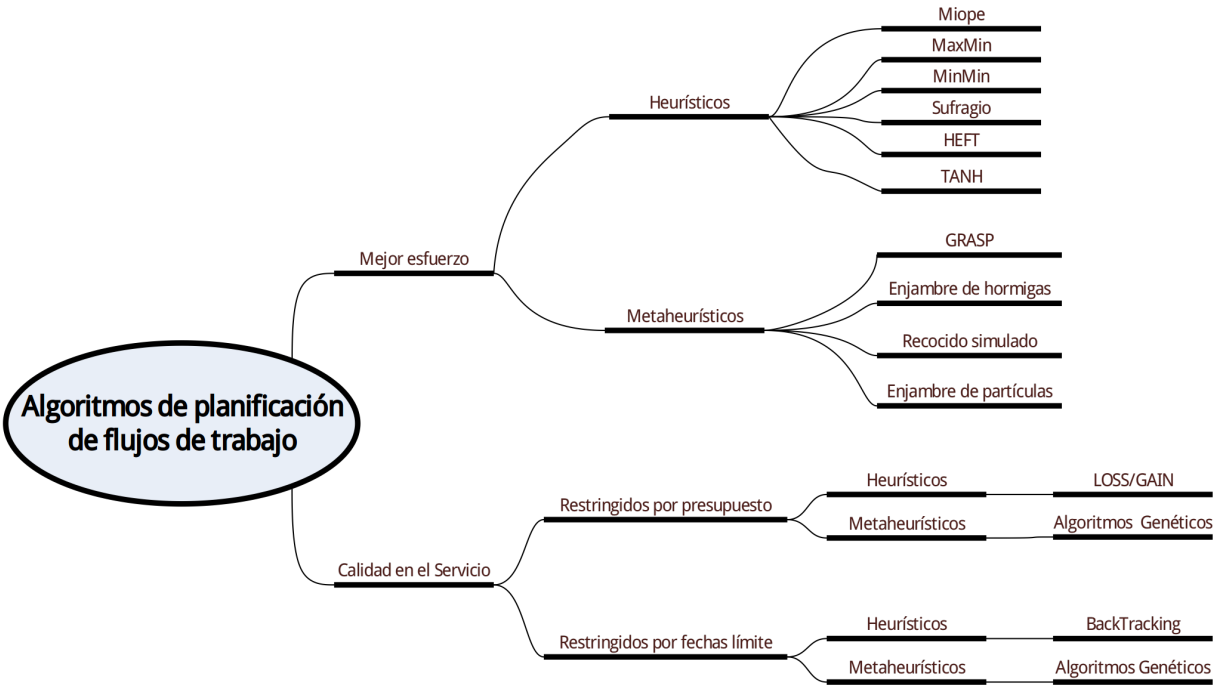

Figura 2.1: Diagrama de la taxonomía de los algoritmos de planificación de flujos de trabajo.

2.3.1. Algoritmos de mejor esfuerzo

Estos algoritmos tratan de minimizar algún criterio, que en muchos casos es el tiempo total de ejecución. Cabe aclarar que para los siguientes algoritmos, se asumirá que el grafo del flujo de trabajo tiene una correspondencia biyectiva entre los nodos $\mathcal{V}$ y las tareas $\mathcal{T}$, es decir, cada nodo del grafo representa una tarea del flujo de trabajo. Las aristas del grafo representan *dependencias entre tareas*.

También, los algoritmos de mejor esfuerzo se pueden dividir en algoritmos guiados por heurísticas y en algoritmos guiados por metaheurísticas. El primer subgrupo debe su nombre a que las soluciones están basadas en ideas que resuelven de manera aproximada el problema de la planificación bajo ciertas condiciones [27]. El segundo subgrupo contiene a los algoritmos que utilizan algún método para elegir o generar planificaciones que se acerquen al óptimo.

Algoritmos heurísticos de mejor esfuerzo.

Los algoritmos heurísticos se pueden dividir en cuatro grandes tipos: a) algoritmo inmediato, b) algoritmos basados en lista, c) agrupamiento de tareas y d) duplicación de tareas.

En el primer grupo se encuentra el algoritmo Miope [14], que sólo asigna tareas a recursos conforme se va necesitando. Los algoritmos basados en lista, como MaxMin,

MinMin y Sufragio [8], ordenan las tareas de acuerdo a algún criterio (fase de priorización) y seleccionan las tareas de acuerdo a cierto criterio (fase de selección). En el caso de los algoritmos de agrupamiento de tareas, tales como TANH [3], al inicio, crean un conjunto para cada tarea. Después, con un paso de mezcla se van uniendo conjuntos hasta que quede un número de conjuntos igual al número de recursos disponibles. Finalmente se ordenan las tareas de cada conjunto para ejecutarse en cada recurso. Por último, los algoritmos de duplicación de tareas (Híbrido [16], TANH [3]) planifican una tarea en varios recursos con el fin de reducir el costo por comunicación entre recursos; cada uno de estos algoritmos se caracteriza por la manera en que eligen las tareas a planificar.

Algoritmos metaheurísticos de mejor esfuerzo.

Estos algoritmos son clasificados de acuerdo a la metaheurística que utilizan. Para esta clasificación, vamos a describir cuatro tipos de metaheurísticas: a) algoritmos genéticos, b) búsqueda aleatoria adaptativa dirigida –GRASP–, c) recocido simulado y d) optimización por enjambre de partículas

Los algoritmos genéticos simulan el proceso de selección natural con posibles planificaciones e introducen mutaciones para evitar enfocarse en una solución local. Con la ayuda de una función de evaluación (*fitness*), se mide la calidad de la planificación y se eligen aquellas soluciones que resulten mejor evaluadas [26]. El algoritmo GRASP [4] genera aleatoriamente soluciones y con un procedimiento voraz[2] elige las soluciones óptimas locales [4]. El recocido simulado, como su nombre lo indica, emula el proceso de formación de cristales donde en cada iteración se va creando una mejor solución [25]. Finalmente, la optimización por enjambre de partículas representa una posible planificación como un punto en el espacio y el algoritmo hace que las mejores soluciones cambien su posición de acuerdo a una velocidad que es proporcional a la calidad de la solución [24].

2.3.2. Algoritmos de calidad en el servicio

En estos algoritmos, se definen un conjunto de restricciones que debe respetar la planificación. Principalmente, estas restricciones tienen que ver con un presupuesto global limitado y un límite en los tiempos de ejecución total del flujo de trabajo. También es posible definir límites de tiempo de ejecución o límites de presupuesto para cada tarea particular del flujo.

De acuerdo a la forma que trabajan estos algoritmos, pueden dividirse en: restringidos por presupuesto o restringidos por fecha límite. Los primeros son algoritmos que utilizan el presupuesto para cumplir con restricciones de calidad del servicio. Los algoritmos restringidos por fecha límite buscan planificaciones que cumplan con fechas proporcionadas. Ambos grupos se subdividen en heurísticos y metaheurísticos. A continuación, describiremos estas subdivisiones.

[2] Greedy

Algoritmos restringidos por presupuesto.

Los algoritmos restringidos por presupuesto se dividen en: a) heurísticos y b) metaheurísticos.

En la primera división se encuentran los algoritmos que trabajan con una idea base que genera soluciones que no están garantizadas que sean óptimas. En este grupo de algoritmos se encuentra el algoritmo LOSS/GAIN, propuesto por Tsiakkouri et al. [17]. Los algoritmos metaheurísticos generan soluciones aleatoriamente y utilizan la información de iteraciones pasadas para refinar las soluciones. En el trabajo de Yu et al. [26] se propone un algoritmo genético cuya función de evaluación valora mejor a las planificaciones que cumplan con el presupuesto con un tiempo de ejecución mínimo.

Algoritmos restringidos por fechas límites.

Los algoritmos restringidos por fechas límites también se dividen en: a) heurísticos y b) metaheurísticos.

En el primer grupo se encuentran el algoritmo BackTracking, propuesto por Menascé et al. [10], y el algoritmo de distribución de fechas límite, propuesto por Yu et al. [28]. En el segundo grupo se encuentra un algoritmo genético presentado en [26], cuya función *fitness* valora mejor a aquellas soluciones que cumplan con la fecha límite, con un presupuesto mínimo.

Estimación del tiempo.

En estos algoritmos se asume que se conoce el tiempo de ejecución de las tareas del flujo de trabajo. Sin embargo, en la etapa de diseño de un flujo de trabajo, es complicado conocer la duración de las tareas, debido a que son muchos los factores que influyen el tiempo total de ejecución del flujo, como el rendimiento de la plataforma de cómputo, la cantidad de datos a procesar, el tiempo de comunicación entre nodos de cómputo, entre otros factores. Para mitigar este problema, se han hecho algoritmos basados en series de tiempo que estiman el tiempo de ejecución de cada una de las tareas de un flujo de trabajo. Estos algoritmos utilizan información histórica de ejecuciones de flujos de trabajo para realizar las predicciones [7].

Capítulo 3

Flujos de trabajo sobre cómputo en la nube

En este capítulo, describiremos el algoritmo que es la pieza clave del sistema para agendar tareas de un flujo de trabajo en infraestructura de cómputo en la nube, tomando en cuenta los lineamientos de diseño propuestos anteriormente.

3.1. Antecedentes

También, un área de investigación activa es la planificación de tareas a nivel de procesador (CPU's), en donde se asume que se tiene un procesador cuyos núcleos son idénticos e intercambiables. Además, en los trabajos de investigación referentes a esta área se consideran a los grafos dirigidos acíclicos como los modelos de tareas más generales. Así, basados en el trabajo de Saifullah et al. [15], en donde primero se transforma un grafo dirigido acíclico en un conjunto de tareas secuenciales con secciones que se ejecutan en paralelo.

Es muy común que los servicios de cómputo en la nube se paguen la utilización de las máquinas virtuales por hora, dejando sin cobrar porciones de hora no utlizadas. Además, se pueden solicitar tantos recursos como se requieran, limitándose solamente al presupuesto disponible. Además, estas máquinas virtuales son recursos variados; pueden ser desde máquinas con un solo procesador, hasta VM's con grandes cantidades de memoria.

De acuerdo al primer capitulo del libro de Pinedo [12] hay esfuerzos para clasificar los problemas de planificación en general. Sin embargo, la clasificación presentada en este trabajo sólo contempla los problemas que optimizan una sola métrica. Aunque, se puede utilizar una combinación de funciones objetivo a optimizar y de esta forma se pueden utilizar estos algoritmos de planificación. Sin embargo, esta forma de atacar el problema de planificación multiobjetivo no garantiza evaluar todas las posibles opciones.

3.2. Los cuatro algoritmos

Si bien en la literatura hay múltiples algoritmos de planificación publicados por la comunidad científica, éste es sólo una parte (muy importante) de un sistema administrador de flujos de trabajo. En el caso de este trabajo, dividimos todo el proceso en cuatro fases:

1. Planificación
2. Asignación de recursos
3. Ejecución
4. Descarga de resultados

3.2.1. Planificación

La idea base del algoritmo es lidiar con las dependencias, ya que éstas generan restricciones sobre el problema. Otra característica a aprovechar es el hecho de que los recursos de cómputo en la nube pueden ser tratados como *efímeros*, es decir, pueden ser creados y destruidos conforme se vayan necesitando.

De esta forma, el primer paso del algoritmo de planificación es dividir el flujo de trabajo en segmentos consecutivos, con la propiedad de que un segmento sólo dependa de la ejecución completa del segmento anterior

Para generar los segmentos, se utiliza la definición de Saifullah et. al [15] basada en la profundidad de cada nodo del DAG que representa el flujo de trabajo.

Definición 7. *La **profundidad** h de una tarea t de un flujo de trabajo está definida como:*

$$h(x) = \begin{cases} \max_{u \in Padres(t)} h(u) + 1 & : |Padres(t)| \neq 0 \\ 1 & : |Padres(t)| = 0 \end{cases}$$

Donde Padres(t) es el conjunto de tareas que inmediatamente preceden a la tarea t, es decir, aquellas tareas u que tienen una dependencia de orden que va a la tarea t.

En la figura 3.1 se puede apreciar un ejemplo de este algoritmo, en donde se puede ver que a cada tarea del flujo de trabajo se le asigna un segmento, en cual se está garantizando que cada segmento sólo dependa del segmento anterior.

Otro punto a notar es que hay un límite en el número de tareas concurrentes que es posible ejecutar debido a las restricciones. Esto implica que a lo sumo hay un número finito de recursos concurrentes que se necesitan para correr el flujo de trabajo. Este límite equivale al máximo de tareas de los segmentos que conforman el flujo de trabajo.

En pseudocódigo 1, se calculan los segmentos de un flujo de trabajo.

Algorithm 1 Segmentación de un flujo de trabajo

Entrada: Un flujo de trabajo $workflow$
Salida: Un mapa con los segmentos de cada tarea del flujo de trabajo

```
1: procedure GET_TASK_SEGMENTS(workflow)
2:     visited ← Diccionario
3:     segment ← Diccionario
4:     for t ∈ workflow.tasks do
5:         GET_SEGMENT(t)
6:     end for
7:     return segment
8: end procedure
9: procedure GET_SEGMENT(task)
10:    visited[task] ← True
11:    max_seg ← 0
12:    if ¬(task ∈ segment) then
13:        segment[task] ← 0
14:        for p ∈ task.parents do
15:            v ← GET_SEGMENT(p)
16:            if v > max_seg then
17:                max_seg = v
18:            end if
19:        end for
20:        segment[task] ← max_seg + 1
21:    end if
22:    return segment[task]
23: end procedure
24: procedure ESTIMATE_RESOURCES(workflow)
25:    segments ← get_task_segments()
26:    segmentsHeight ← Diccionario
27:    max_segment ← 0
28:    for k, v in segments do
29:        segment ← v
30:        if segment ∈ segmentsHeight then
31:            val ← segmentsHeight[segment]
32:        else
33:            val ← 0
34:        end if
35:        val ← val + 1
36:        max_segment ← máx(val, max_segment)
37:    end for
38: end procedure
```

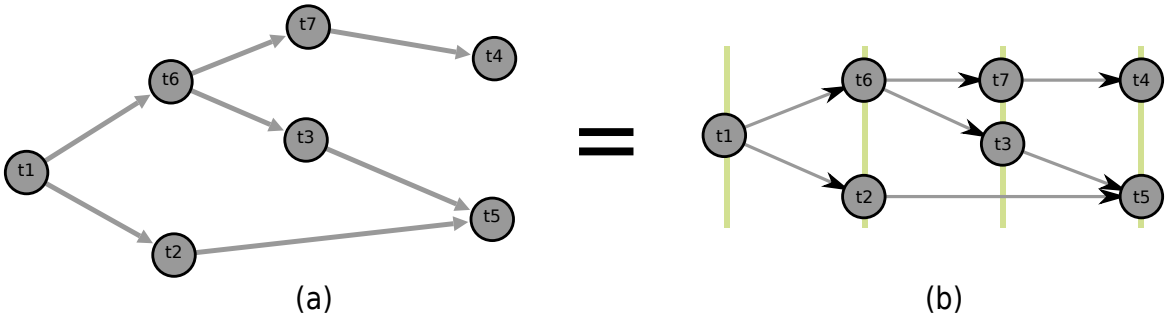

Figura 3.1: Transformación de un flujo de trabajo (a) a segmentos (b)

3.2.2. Asginación de recursos

Ahora, el cómputo en la nube abre otras posibilidades que no se pueden realizar en los entornos de cómputo distribuido tradicionales, como el hecho de poder solicitar máquinas virtuales de acuerdo a la demanda. De esta forma, con las tareas del flujo de trabajo segmentadas, se puede ver a cada segemento como un conjunto de tareas independientes. Así, se puede usar un algoritmo de programación dinámica que pueda encontrar la asignación óptima de máquinas virtuales y tareas entre cada segmento.

Así, este algoritmo se puede ver como el algoritmo de la mochila *inverso* con múltiples mochilas, en donde los las mochilas son las posibles configuraciones y las tareas son objetos que deben ser *embolsados*. El objetivo de este algoritmo es encontrar aquella asignación que pueda embolsar todas las tareas maximizando el uso del espacio de las mochilas, sujeto a una condición de optimalidad. Para este caso, la condición de optimalidad es encontrar el costo mínimo de ejecución.

Por el momento, para hacer el match entre configuraciones y tareas, se asume que cada tarea ocupa un núcleo de una máquina virtual y que todas las posibles configuraciones de máquinas virtuales del proveedor tienen memoria primaria suficiente para ejecutar las tareas.

El algoritmo de asignación de recursos a tareas de un segmento está descrito en el pseudocódigo 2. Como se puede notar en este pseudocódigo, hay dos funciones que son invocadas. La primera, llamada TAKE, calcula los costos óptimos de generar una asignación que utilice todos los cores de las configuraciones de la máquinas virtuales

Ahora, en los siguentes pseudocódigos, se mostrarán las funciones principales para generar la asignación de recursos:

La función anterior calcula el costo de la asignación óptima. La siguiente función construye a partir de la tabla de resultados generada por el procedimiento anterior.

Ahora bien, si el tiempo requerido para invocar máquinas virtuales fuera cero y las transferencias de datos fueran instantáneas, estos serían los únicos algoritmos que necesitariamos ejecutar para correr flujos de trabajo. Sin embargo, como las acciones anteriores no se ejecutan instantáneamente, hay que tomar en cuenta estos costos a

Algorithm 2 Asignación de configuraciones a segmento de flujo de trabajo

Entrada: Los siguientes parámetros:

- Una lista de tareas de un segmento *tasks*
- Una lista de configuraciones de recursos $resource\_configs$

Salida: Una lista de asignaciones de tarea – recurso *resource_mappings*

1: **procedure** BIN_PACKING($tasks$, $resource\_configs$)
2: $mem\_costs \leftarrow$ Matriz de $|tasks| \times |resource\_configs|$, inicializado con 0
3: $visited \leftarrow$ Matriz de $|tasks| \times |resource\_configs|$, inicializado con 0
4: $used \leftarrow$ Arreglo de tamaño $|resource\_configs|$, inicializado con `MAX_INT`
5: $resource\_mappings \leftarrow$ Lista vacía
6: TAKE(0, 0)
7: CHECK_TAKE(0, 0)
8: **return** $resource\_mappings$
9: **end procedure**

la hora de planificar.

3.2.3. Ejecución

Con las asignaciones ya resueltas, se procede a ejecutar el flujo de trabajo. (Todavía no está lista esta parte del algoritmo)

3.2.4. Descarga de resultados

Finalmente, si en las tareas hay una entrada que indique descargar los resultados, se transfieren los archivos especificados a la máquina principal que ejecuta el algoritmo.

Algorithm 3 Costos de ejecución de un segmento de un flujo de trabjo

Entrada: Los siguientes parámetros:

- Una lista de tareas de un segmento $tasks$
- Una lista de configuraciones de recursos $resource\_configs$

Salida: Una lista de asignaciones de tarea–recurso $resource\_mappings$

1: **procedure** TAKE(t_i, rc_i)
2: **if** $rc_i = |resource\_configs|$ **then**
3: **return** ALL_SCHEDULED
4: **end if**
5: **if** $visited[t_i, rc_i] \neq 0$ **then**
6: **return** $mem\_costs[t_i, rc_i]$
7: **end if**
8: **for** $y \in [rc_i, \ldots, |resource\_configs|]$ **do**
9: $rc \leftarrow resource\_configs[y]$
10: $t_{lim} \leftarrow \text{mín}(|tasks|, t_i + rc.cores)$ ▷ Permitimos subutilización
11: **for** $tt \in tasks[t_i, \ldots, t_{lim}]$ **do** ▷ Costo del core más usado
12: $taked\_cost \leftarrow \text{máx}(taked\_cost, \frac{tt.complexity\_factor}{rc.speed\_factor} rc.cost\_hour\_usd)$
13: **end for**
14: $used[y] \leftarrow used[y] + 1$
15: $taked\_yes \leftarrow$ TAKE($t_{lim}, 0$)
16: $used[y] \leftarrow used[y] - 1$
17: $taked\_not \leftarrow$ TAKE($t_i, y + 1$)
18: **if** $taked\_yes < taked\_not$ **then**
19: $mem\_costs[t_i, y] \leftarrow taked\_yes$
20: $visited[t_i, y] \leftarrow 1$
21: **else**
22: $mem\_costs[t_i, y] \leftarrow taked\_not$
23: $visited[t_i, y] \leftarrow -1$
24: **end if**
25: **end for**
26: **return** $mem\_costs[t_i, rc_i]$
27: **end procedure**

Algorithm 4 Asignación óptima de un segmento de un flujo de trabajo

Entrada: Tabla de resultados generada por TAKE
Salida: Dunno

```
1: procedure CHECK_TAKE(t_i, rc_i)
2:     if t_i < |tasks| ∧ rc_i < |resource_configs| then
3:         if visited[t_i, rc_i] = 1 then
4:             rc ← resource_configs[rc_i]
5:             t_lim ← mín(|tasks|, t_i + rc.cores)
6:             Añadir (tasks[t_i ... t_lim], resource_configs[rc_i]) a resource_mappings
7:             CHECK_TAKE(t_lim, 0)
8:         else if visited[t_i, rc_i] = −1 then
9:             CHECK_TAKE(t_i, rc_i + 1)
10:        end if
11:    end if
12: end procedure
```

Capítulo 4

Sweeper: ejecución de flujos de trabajo en cómputo en la nube

En este capítulo se describen los detalles del funcionamiento de Sweeper [1], el sistema de administración de flujos de trabajo orientado a cómputo en la nube. Utilizando este paquete, se pueden ejecutar tareas expresadas como comando de una terminal Linux, con dependencias de orden entre ellas entre recursos en la nube. Sweeper está desarrollado en el lenguaje de programación Python [13].

Para ejecutar flujos de trabajo con Sweeper, se especifican las tareas del flujo de trabajo y sus dependencias. Sweeper lee este archivo de descripción y enseguida estima los tiempos de ejecución de las tareas y elige la mejor asignación de máquinas virtuales y tareas del flujo de trabajo que optimicen los criterios dados.

Para crear un flujo de trabajo, se crea un archivo que cumpla con el formato YAML [6] llamado `workflow.yaml`. De esta forma, en la sección `workflow`, se definen, en forma de lista, las tareas que conforman el flujo de trabajo. En el código 4.1 se puede apreciar la estructura del archivo del flujo de trabajo.

En el listado de código anterior se muestra la descripcion de un flujo de trabajo que lista los archivos contenidos en las máquinas virtuales que ejecutan este flujo. También, este flujo trabajo contiene una tarea para recolectar la información de la versión del sistema operativo que ejecuta la máquina virtual. Por otro lado, en la figura 4.1 se encuentra la representación visual de este flujo de trabajo.

En el ejemplo anterior puede apreciar los elementos básicos que componen una tarea del flujo de trabajo, explicados a continuación:

- `name`: Nombre de la tarea, que es único entre los nombres de las demás tareas del flujo de trabajo.
- `command`: Comando en sintaxis de Bash que será ejecutado en esta tarea.

```
workflow:
  - name: check_files
    command: ls -alp

  - name: create_filelist
    command: ls -alp > sal.txt
    depends: [check_files]
    download_files: [sal.txt]

  - name: create_sysinfo
    command: lsb_release -a > info.txt
    depends: [check_files]

  - name: final_results
    command: ls -alp
    depends: [create_filelist, create_sysinfo]
```

Código 4.1: Flujo de trabajo de ejemplo.

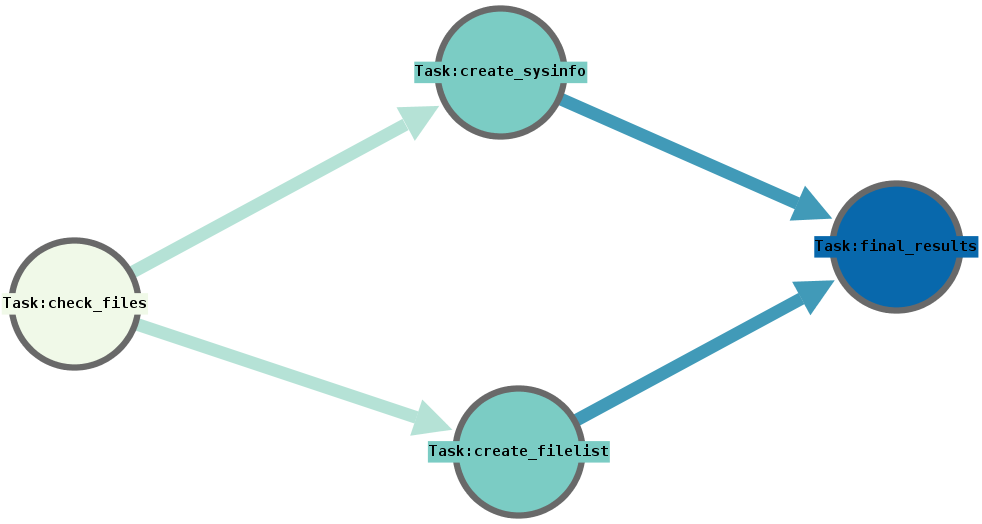

Figura 4.1: Visualización del flujo de trabajo *test*

- `depends`: Lista de los nombres de las tareas que necesitan ser ejecutadas antes de que esta tarea pueda ejecutarse.
- `download_files`: Lista de los archivos que deben ser descargados después de terminar la ejecución de la tarea.
- `include_files`: Lista de los archivos que deben ser subidos al cluster antes de que la tarea sea ejecutada.

Una característica importante de Sweeper que lo hace único a otros sistemas de administración de flujos de trabajo, es la expansión de parámetros. El usuario puede especificar una tarea que se requiera ejecutar con un conjunto predefinido de argumentos. Sweeper calcula todas las combinaciones posibles de parámetros y automáticamente crea una tarea a ejecutarse por cada posible conjunto de parámetros. Así, en el listado de código 4.3 se puede notar cómo se denotan las listas de parámetros de donde se generan las combinaciones de argumentos. En la figura 4.2 se encuentra la representación completa de este flujo de trabajo.

4.1. Arquitectura

En figura 4.3 se muestra la arquitectura de Sweeper, en donde se pueden notar cuatro grandes componentes: la ejecucíón del flujo de trabajo, el planificador, el perfilador de las tareas y el perfilador de los recursos. Sweeper utiliza las interfaces de programación de aplicaciones de cada proveedor de servicios de cómputo en la nube para reservar, ejecutar y administrar los recursos utilizados en la nube de cada proveedor. Actualmente, Sweeper puede trabajar con los servicios de cómputo en la nube de Microsoft Azure [11]. En las siguientes subsecciones, describiremos a detalle cada uno de los componentes de Sweeper.

4.1.1. Ejecución de flujos de trabajo

Este componente se encarga de ejecutar el flujo de trabajo en los recursos en la nube de acuerdo a la planificación generada por el componente de planificación. Por el momento, este componente implementa un sistema de colas concurrente en las que las tareas son insertadas y despachadas de acuerdo al tiempo de ejecución estimado.

4.1.2. Planifcador

El planificador busca la asignación de recursos que optimice la calidad en el servicio en la ejecución del flujo de trabajo, interpretado como restricciones de presupuesto o restricciones de fechas límite.

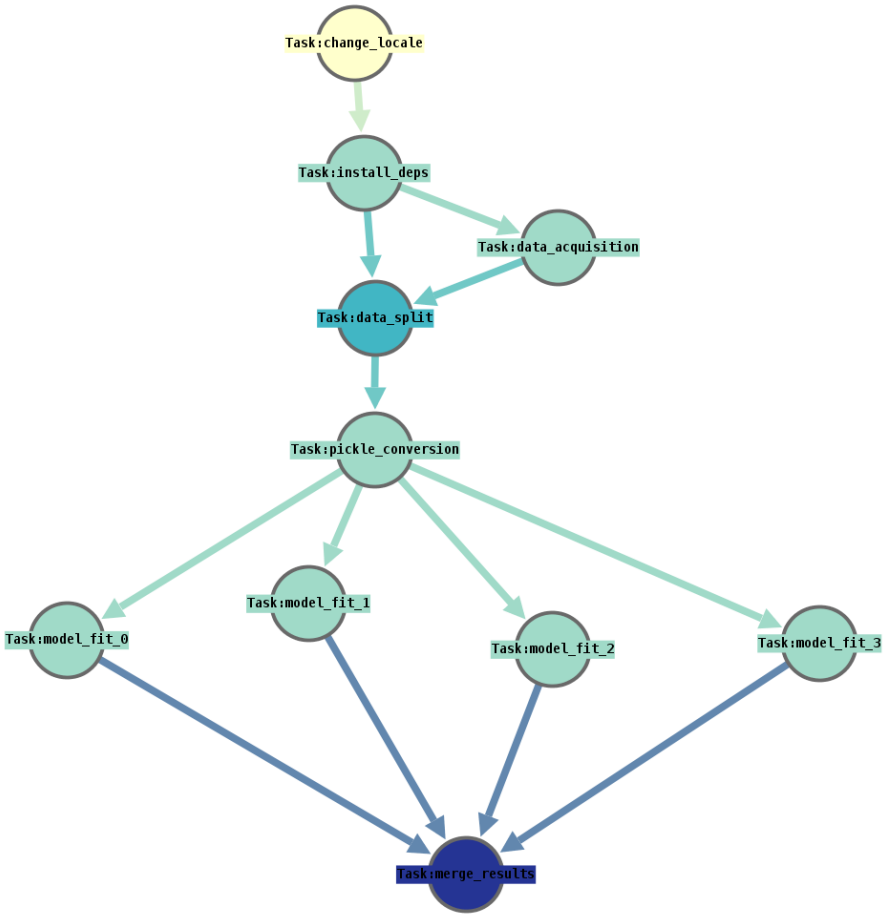

Figura 4.2: Visualización del flujo de trabajo *forests*

Figura 4.3: Arquitectura de Sweeper

```
workflow:
  - name: change_locale
    command: |
        echo "LC_ALL=en_US.UTF-8" | sudo tee -a /etc/environment;
        echo "LANG=en_US.UTF-8" | sudo tee -a /etc/environment;
        sudo update-locale LANG=en_US.UTF-8 LC_ALL=en_US.UTF-8;

  - name: install_deps
    command: |
        set -o xtrace;
        # scikit-learn
        export DEBIAN_FRONTEND=noninteractive;
        sudo apt-get update;
        sudo apt-get install unzip -y;
        sudo apt-get install build-essential gfortran gcc g++ \
                        python3-dev python3-pip python3-setuptools \
                        python3-numpy python3-scipy python3-matplotlib \
                        libatlas-dev libatlas-base-dev libatlas3gf-base -y;
        sudo update-alternatives --set libblas.so.3 /usr/lib/atlas-base/atlas
            /libblas.so.3;
        sudo update-alternatives --set liblapack.so.3 /usr/lib/atlas-base/
            atlas/liblapack.so.3;
        sudo pip3 install -U scikit-learn;
        # doge package
        sudo apt-get install git -y;
        git clone https://github.com/dominoFire/doger.git;
        cd doger;
        sudo pip3 install .;
        cd ..;
    depends: [change_locale]

  - name: data_acquisition
    command: |
        unzip train.csv.zip;
        mv train.csv labeled.csv;
    include_files: [train.csv.zip]
    depends: [install_deps]

  - name: data_split
    command: |
        doger split_traintest labeled.csv 0.70;
        doger split_xy labeled_train.csv Cover_Type;
        doger split_xy labeled_test.csv Cover_Type;
    depends: [data_acquisition, install_deps]

  - name: pickle_conversion
    command: |
        doger csv2pk labeled_train_predictors.csv labeled_train_response.csv;
        doger csv2pk labeled_test_predictors.csv labeled_test_response.csv;
    depends: [data_split]
```

Código 4.2: Flujo de trabajo para búsqueda de parámetros (parte 1).

```
1   - name: model_fit
2     command: |
3         doger gridsearch \
4             labeled_train_predictors.pk labeled_train_response.pk \
5             labeled_test_predictors.pk labeled_test_response.pk \
6             @config_file \
7             obj out;
8     depends: [pickle_conversion]
9     param_grid:
10        config_file: [gridsearch_xt_config.py, gridsearch_rf_config.py,
              gridsearch_gnb_config.py, gridsearch_bnb_config.py]
11    include_files: [gridsearch_xt_config.py, gridsearch_rf_config.py,
          gridsearch_gnb_config.py, gridsearch_bnb_config.py]
12
13  - name: merge_results
14    command: |
15        doger merge out;
16    depends: [model_fit]
17    download_files: [out/AllResults.csv]
```

Código 4.3: Flujo de trabajo para búsqueda de parámetros (parte 2).

4.1.3. Perfilador de tareas

El perfilador de tareas se utiliza para obtener un modelo de la estimación de tiempos de ejecución independiente de la máquina, el cual es útil para generar planificaciones que optimicen el tiempo total de ejecución o el presupuesto utilizado para la ejecución del flujo de trabajo. A grandes rasgos, el Perfilador de tareas mide el tiempo de ejecución de cada tarea de un flujo de trabajo, guardando un registro de estos tiempos de ejecución junto con los parámetros que se utilizaron para invocar cada tarea.

Para utilizar el perfilador, hay que invocar Sweeper con la opción `profile` en la carpeta en donde se encuentre el archivo de flujo de trabajo (`workflow.yaml`) que se quiera ejecutar, de la siguiente forma:

```
1 $ sweeper profile
```

4.1.4. Perfilador de recursos

También, para estimar la velocidad de las máquinas virtuales de un proveedor, sweeper se auxilia del PerfKitBenchmarker, un software diseñado para correr benchmarks, los cuales son pruebas de estrés que ejecutan las máquinas virtuales, cuyos resultados son analizados posteriormente. Para el caso de Sweeper, se utilzan las mediciones de FLOPS que provee el benchmark SPECfp.

Capítulo 5

Resultados

En este capitulo presentamos algunos resultados del algoritmo ciego y algunas comparaciones con otros algoritmos de planificación.

Para probar el algoritmo ciego, se generaron aleatoriamente flujos de trabajo utilizando el simulador de flujos de trabajo. El objetivo de estas simulaciones es comparar cuán óptimas son las planificaciones que genera el algoritmo de planificación propuesto en este trabajo respecto a los algoritmos MaxMin, MinMin y Miope. Cabe aclarar que para estas pruebas, el número y el tipo de recursos es determinado por el algoritmo ciego. Luego, utilizando estos recursos sugeridos por el algoritmo ciego, se generan otras planificaciones con los algoritmos MaxMin, MinMin y Miope. También, es importante notar que estos tres últimos algoritmos fueron programados para encontrar la planificación con el menor tiempo total de ejecución (makespan). La implementación del algoritmo ciego se encuentra en [2].

Debido a que el algoritmo ciego es configurable, se probó el algoritmo ciego sobre un conjunto de flujos de trabajo de prueba utlizando dos funciones de costo parcial diferentes. La primera función prueba el algoritmo ciego para que optimice el costo de ejecucion. La segunda función se utilizó para probar el algoritmo ciego configurado para optimizar el tiempo de ejecución.

Se generaron 1000 flujos de trabajo con un número variable de tareas, que van desde 1 hasta 50 tareas. Los factores de complejidad de las tareas varían entre 50 y 100. Cada flujo de trabajo fue generado con base en un generador de números aleatorios congruencial, utilizando una semilla diferente para cada flujo de trabajo.

Las configuraciones de recursos utlizados son los siguientes:

Para evaluar el desempeño de los algoritmos de planificación, se tomaron las planificaciones que arrojaba cada algortmo y con ellas, se calculó tanto el tiempo total de ejecución como el costo total de ejecución. El segundo es definido como el producto del costo por hora de mantener una máquina virtual por el periodo de tiempo por el que la máguina se encuentra prendida, es decir, la diferencia entre el tiempo final de la última tarea a ejecutar por la máquina y el tiempo inicial de la primera tarea a ejecutar.

Configuración	Cores	fv	Costo/Hr
Small	1	100	2.3
Medium	2	200	4.0
Large	6	400	7.0
ExtraLarge	8	800	10.0

Cuadro 5.1: Configuraciones de los recuros utilizados para las pruebas

En tablas 5.2 y 5.3, se pueden ver los resultados de las ejecuciones. Sorprendentemente, los resultados variando la función de costo parcial son muy parecidos. Esto puede explicarse a que ambas funciones comparten el componente $\frac{fc}{fv}$, donde fc es el factr de complejidad de la tarea y fv es el factor de velocidad del recurso. Además, La optimización de segmentos que realiza el algoritmo ciego, está basada en un algoritmo de programación dinámica, lo cual hace que ambas funciones similares obtengan resultados similares, ya que estas funciones de costo parcial obtienen costos proporcionales.

Algoritmo	Makespan	Varianza	Costo	Varianza
blind	7.75	3.93	77.74	51.78
maxmin	4.23	2.70	71.23	52.05
minmin	4.13	2.10	127.73	79.21
myopic	5.93	3.27	226.37	133.82

Cuadro 5.2: Resultados agredado de los algoritmos usando el algoritmo ciego para optimizar makespan

Algoritmo	Makespan	Varianza	Costo	Varianza
blind	7.75	3.93	77.74	51.78
maxmin	4.23	2.70	71.04	52.10
minmin	4.13	2.10	127.95	79.43
myopic	5.92	3.26	226.33	133.87

Cuadro 5.3: Resultados agredado de los algoritmos usando el algoritmo ciergo para optimizar costo

Si bien, analizando los resultados de las tablas anteriores, el algoritmo ciego tiene makespans promedios bastante altos comparados con los otros tres algoritmos. Sin embargo, el algoritmo ciego obtiene el segundo mejor costo promedio de los algoritmos comparados.

Capítulo 6

Conclusiones

En este trabajo, se propuso un nuevo algoritmo de planificacion de flujos de trabajo, tomando en cuenta su posible ejecución y despliegue en entornos de cómputo en la nube. Este algoritmo, a diferencia de los utlizados para comparar los resultados obtenidos, puede calcular el número mínimo de máquinas o recursos necesarios para correr el flujo de trabajo sin que alguna tarea del flujo de trabajo tenga que esperar por recursos. Otra ventaja de este algoritmo es que se puede utilizar una función de costo parcial, ya sea para optimizar costo, tiempo de ejecución u otros requisitos específicos.

También, en este trabajo se creó un prototipo de un sistema administrador de flujos de trabajo en computo en la nube, llamado sweeper, con el objetivo de vislumbrar cuáles son las complejidades de construir un sistema de estos. Sin duda, existen muchos retos técnicos a solventar, como la administración de las máquinas virtuales, el almacenamiento y la distribución de resultados, la configuración de los programas a ejecutar, entre otros.

6.1. Trabajo futuro

Existen múltiples áreas de oportunidad para mejorar este algoritmo, entre ellas se encuentran:

1. Una demostración formal de que el algoritmo ciego calcula el numero óptimo de máquinas necesarias para alcanzar el paralelismo no bloqueante

2. Nuevas funciones de costo parcial para evaluar el comportamiento del algoritmo utlizando otras fuentes de optimización

Por otro lado, el prototipo tiene posibles líneas de trabajo futuro, las cuales se mencionan a continuación:

1. Pooling de máquinas virtuales, debido a que es muy costoso en términos de tiempo el prender y apagar una máquina virutal.

2. Streaming de resultados. El prototipo actual descarga los resultados de la ejecución una vez que terminó el flujo de trabajo; sería mejor que entregue resultados conforme se vayan obteniendo.

3. Contenedores. Recientemente, la virtuallización a nivel sistema operativo se ha vuelto popular para desplegar servicios en la nube. Su principal característica es que pueden crearse más rápido que una máquina virtual y que pueden funcionar con un kernel compartido.

Bibliografía

[1] dominofire/sweeper – Bitbucket Git epository. `https://bitbucket.org/dominofire/sweeper/`. Consultado el 24 de abril de 2015.

[2] dominofire/workflow-simulator – Bitbucket Git epository. `https://bitbucket.org/dominofire/workflow-simulator/`. Consultado el 28 de julio de 2014.

[3] Rashmi Bajaj and D.P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, February 2004.

[4] James Blythe, Sonal Jain, Ewa Deelman, Yolanda Gil, Karan Vahi, Anirban Mandal, and Ken Kennedy. Task scheduling strategies for workflow-based applications in grids. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, volume 2, page 759–767. IEEE, 2005.

[5] Grupo BMV. Informe Anual 2012. Technical report, Bolsa Mexicana de Valores, 2012.

[6] Ingy döt Net, Clark Evans, and Oren Ben-Kiki. YAML Ain't Markup Language. `http://www.yaml.org/about.html`.

[7] Xiao Liu, Zhiwei Ni, Dong Yuan, Yuanchun Jiang, Zhangjun Wu, Jinjun Chen, and Yun Yang. A novel statistical time-series pattern based interval forecasting strategy for activity durations in workflow systems. *Journal of Systems and Software*, 84(3):354–376, 2011.

[8] Muthucumaru Maheswaran, Shoukat Ali, HJ Siegal, Debra Hensgen, and Richard F Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 30–44. IEEE, 1999.

[9] Michael Mair, Jun Qin, Marek Wieczorek, and Thomas Fahringer. Workflow conversion and processing in the ASKALON grid environment. In *2nd Austrian Grid Symposium*, pages 67–80. Citeseer, 2007.

[10] Daniel A Menasce and Emiliano Casalicchio. A Framework for Resource Allocation in Grid computing. In *MASCOTS*, pages 259–267, 2004.

[11] Microsoft Corporation. Microsoft Azure: Cloud computing platform & Services. `http://azure.microsoft.com/`. Consultado el 24 de abril de 2015.

[12] Michael L. Pinedo. *Scheduling: theory, algorithms, and systems*. Springer Science & Business Media, 2012.

[13] Python Software Foundation. Python 3.4.3 documentation. `https://docs.python.org/3/`. Consultado el 24 de abril de 2015.

[14] Krithi Ramamritham, John A. Stankovic, and P-F Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 1(2):184–194, 1990.

[15] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

[16] R. Sakellariou and Henan Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 111–, April 2004.

[17] Rizos Sakellariou, Henan Zhao, Eleni Tsiakkouri, and Marios D Dikaiakos. Scheduling workflows with budget constraints. In *Integrated Research in GRID Computing*, pages 189–202. Springer, 2007.

[18] Jamie Shiers. The worldwide LHC computing grid (worldwide LCG). *Computer physics communications*, 177(1):219–223, 2007.

[19] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[20] Jeffrey D. Ullman. NP–complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[21] Wil MP van Der Aalst, Arthur HM Ter Hofstede, Bartek Kiepuszewski, and Alistair P Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[22] Marek Wieczorek, Andreas Hoheisel, and Radu Prodan. Taxonomies of the multi-criteria grid workflow scheduling problem. In *Grid Middleware and Services*, pages 237–264. Springer, 2008.

[23] Marek Wieczorek, Andreas Hoheisel, and Radu Prodan. Towards a general model of the multi-criteria workflow scheduling on the grid. *Future Generation Computer Systems*, 25(3):237–256, 2009.

[24] Zhangjun Wu, Zhiwei Ni, Lichuan Gu, and Xiao Liu. A revised discrete particle swarm optimization for cloud workflow scheduling. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 184–188. IEEE, 2010.

[25] Laurie Young, Stephen McGough, Steven Newhouse, and John Darlington. Scheduling architecture and algorithms within the iceni grid middleware. In *UK e-Science All Hands Meeting*, pages 5–12. Citeseer, 2003.

[26] Jia Yu and Rajkumar Buyya. Scheduling scientific workflow applications with deadline and budget constraints using genetic algorithms. *Scientific Programming*, 14(3):217–230, 2006.

[27] Jia Yu, Rajkumar Buyya, and Kotagiri Ramamohanarao. Workflow scheduling algorithms for grid computing. In *Metaheuristics for scheduling in distributed computing environments*, pages 173–214. Springer, 2008.

[28] Jia Yu, Rajkumar Buyya, and Chen Khong Tham. Cost-based scheduling of scientific workflow applications on utility grids. In *e-Science and Grid Computing, 2005. First International Conference on*, pages 8–pp. IEEE, 2005.