INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO

Estudio de algoritmos de planificación para flujos de trabajo

TESIS
QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

P R E S E N T A

FERNANDO AGUILAR REYES

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

MÉXICO, D.F. **2014**

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 “Estudio de algoritmos de planificación de flujos de trabajo”, 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

ASDADSF ASDFA DFASA

Abstract

adfsafa asdf dfaF

Agradecimientos

Al profesor Octavio Gutiérrez, que sin su valiosa guía y ayuda, no habría sido posible realizar esta tesis.

A mis padres, Florinda y Miguel, por darme la vida, su tiempo y su ejemplo para motivarme a ser una persona de bien. También, agradezco a Rodrigo y Mike

Al Buró de Crédito, en especial a Eduardo Bettinger, María Eugenia Rico y Susana Pacheco, por depositar su confianza en mí y por darme la oportunidad de estudiar con la Beca Universitaria Red de Oportunidades.

Al personal del Departamento de Becas del ITAM, a los fundadores y donadores de la beca Nuestro ITAM.

A Martín Ibarra, por ser un mentor en la vida. A Gabriel Ibarra

A mis amigos de la Bátiz: Lira, Nieves, Félix, Mara, Karina.

A mis amigos del ITAM: Yorch, Alain, Leo R., Huipet, Andreu, Vic, Feliza, Mireya, Mich, Montse, Nacho, Julián y al Tocayo.

A mis amigos de la Maestría: Gaby, Salvador, Karen, Luis, Alfonso K., Alfonso A. y Jorge Acosta.

A mis asistentes Mario, Felipe, Marco y Sergio.

Índice general

Índice de figuras

Capítulo 1

Introducción

Cada día, utilizamos aplicaciones de cómputo más complejas que resuelven problemas más complicados, debido a que estos problemas involucran el procesamiento de grandes volúmenes de datos o también involucran la omnipresencia de contenido multimedia; así, éstas aplicaciones demandan 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 [10].

Otro ejemplo son las películas de animación: cuando los diseñadores han terminado de modelar a los personajes junto con el entorno y cuando también han especificado las animaciones de los personajes sobre el entorno, se requiere generar cada fotograma de la animación para después juntarlos y proyectarlos rápidamente para crear la ilusión de movimiento y, de esta forma, crear una película. Por ejemplo, en el 2005, investigadores de la Universidad de Innsbruck en Austria generaron una animación tridimensional, primero utilizando una sola computadora y luego utilizando varias computadoras interconectadas [27]. En el primer caso, tardaron aproximadamente 6 días en procesar la animación cuya duración es de un minuto. Por otro lado, utilizando varias computadoras, la animación fue procesada en poco menos de una hora.

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 (CERN). Se estima que cada año, el detector principal del LHC genera 15 petabytes (aproximadamente 15×10^{15} bytes) de datos que requieren ser analizados [32].

¿Qué tienen en común todas estas aplicaciones? Para empezar, toman mucho tiempo en ejecutarse. Entonces, una posible solución para disminuir el tiempo de ejecución de estas aplicaciones es distribuir el gran trabajo que requieren estos proyectos entre varias computadoras. Para ello, necesitamos dividir nuestra aplicación en partes

más pequeñas e independientes, algunas de ellas podrán ejecutarse de manera concurrente y paralela, otras no. De esta forma, tendríamos una solución escalable, es decir, si aumentamos el número de computadoras disponibles para nuestra aplicación, reduciremos el tiempo de ejecución.

Lograr esta paralelización requiere un esfuerzo por parte del desarrollador de la aplicación. Existen técnicas de paralelización que permiten al desarrollador expresar la aplicación en varias partes paralelas. A continuación, enlistaremos algunas de estas técnicas:

- **Programas multiproceso.** En esta técnica, la idea principal es utilizar varios procesos para repartir el cómputo. Por ejemplo, en el modelo fork/join, un programa se invoca recursivamente a sí mismo (fork), de tal modo que cada subprograma resuelve un problema más pequeño que el problema original. Después, los resultados parciales de cada subprograma son juntados en un sólo programa (join).

- **Threads.** Un proceso, en vez de invocarse a sí mismo varias veces, también puede invocar *threads* o hilos de ejecución que, a diferencia de un proceso, éstos no son controlados por el administrador de procesos del sistema operativo, sino por el programa mismo. Por lo tanto, ocupan menos recursos del sistema. De igual modo, se pueden implementar varios modelos de programación concurrente (por ejemplo: fork/join o productor/consumidor) como se hace con los programas multiproceso.

- **MapReduce.** Este paradigma de programación especializado en procesar grandes volúmenes de datos [13] funciona de la siguiente forma: primero se define una función $map() : (k_1, A) \rightarrow list(k_2, B)$, que se aplica a todos los elementos de tipo A para asociarles una llave y transformarlos al tipo B. Luego, la función $reduce() : (k_2, list(B)) \rightarrow list(B)$, hace una operación asociativa para sumarizar los resultados.

- **MPI.** La interfaz de paso de mensajes (MPI por sus siglas en inglés) es un conjunto de definiciones de bibliotecas [23], cuyas subrutinas son utilizadas por varios procesos paralelos para que puedan comunicarse entre si enviándose mensajes de manera asíncrona.

Aunque estas técnicas de paralelización son muy efectivas, éstas son aplicadas cuando el problema a resolver ha sido bien definido y cuando sólo hay una instancia de la aplicación que resuelve el problema planteado.

Ahora bien, hay aplicaciones que involucran varios pasos que están relacionados entre sí, por ejemplo, que el programa C requiera de la salida del programa A y del programa B para que pueda funcionar, tal y como se muestra en la figura 1.1. También, es conveniente definir los grandes bloques de la aplicación, porque puede suceder que los programas A, B y C estén hechos con diferentes técnicas de paralelización y/o construidos con diferentes plataformas y lenguajes, por lo que cada uno debe ejecutarse por separado.

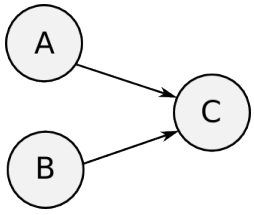

Figura 1.1: Flujo de trabajo con una tarea (C) que depende de dos tareas (A y B).

En los dos casos descritos anteriormente, hay una cierta secuencia que debemos seguir y, dentro de dicha secuencia, hay algunos pasos que podemos resolver de manera concurrente y otros pasos se resuelven de manera secuencial. De esta forma, los pasos de la aplicación representan un *modelo* de nuestro problema. Este modelo también es llamado *flujo de trabajo*. De manera muy abstracta, un flujo de trabajo es un conjunto de pasos que modelan la ejecución de un proceso. La utilidad de este sencillo concepto ha sido probado en varias áreas, de las cuales mencionaremos dos aplicaciones: en el ámbito de los negocios y en el ámbito científico.

- En el ámbito de los negocios, se puede definir un flujo de trabajo para modelar el proceso de fabricación de un producto. También, es posible expresar un flujo de trabajo con un diagrama dibujado con los bloques y reglas del *Lenguaje de Ejecución de Procesos de Negocio* para después simular su ejecución en una computadora.

- En el ámbito científico, los flujos de trabajo son utilizados para modelar aplicaciones que analizan resultados de experimentos diversos. Por ejemplo, la NASA utiliza una aplicación llamada Montage para generar una gran imagen de la bóveda celeste a partir de imágenes más pequeñas tomadas en diferentes observatorios.

En cualquiera de los dos casos anteriores, es deseable distribuir la ejecución de éstos flujos de trabajo entre varias computadoras. Si bien es posible paralelizar algunos pasos de la ejecución de nuestro flujo de trabajo utilizando las técnicas antes mencionadas, hay restricciones de orden que se deben respetar, por lo cual, es indispensable *planificar* la ejecución del flujo de trabajo entre las múltiples computadoras.

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 [39], 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 [36], 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 intepretado desde varias perpectivas , 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 sistemas de cómputo distribuidos. Finalmente, es desesable 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.

En este trabajo se hace un estudio de los principales algoritmos de planificación de flujos de trabajo, con énfasis en los algoritmos utilizados en cómputo distribuido, en especial en cómputo en la nube. En el capítulo 2 se provee un estudio detallado del concepto de flujos de trabajo y su aplicación en computación. El capítulo 3 trata los principales enfoques de cómputo distribuido para ejecutar estos flujos. En el capítulo 4 se hace un estudio de los principales algoritmos de planificación de los flujos de trabajo. Finalmente, en el capítulo 6 discutiremos algunas conclusiones sobre el análisis de estos algoritmos.

Capítulo 2

Flujos de trabajo

2.1. Definición y ejemplos

Un flujo de trabajo es un conjunto de pasos que modelan la ejecución de un proceso [17]. En particular, en esta tesis se estudian a los flujos de trabajo utilizados para vislumbrar la ejecución de un proceso de cómputo. A continuación, se muestran algunos ejemplos de estos flujos de trabajo.

2.1.1. Anotación de proteínas

En el proyecto *e-Protein*, realizado por la Escuela Imperial de Londres, se realizó un flujo de trabajo para la anotación de proteínas. El objetivo del proyecto [28] era la identificación y anotación de partes de proteínas que expliquen su estructura y su función. En la figura 2.1 se muestra el flujo de trabajo desarrollado, donde las cajas representan los programas que son ejecutados para cada paso del proceso de anotación, y las líneas que conectan a las cajas representan las dependencias de datos entre los programas, es decir, si una caja tiene una línea que apunta a ella, significa que dicho programa depende de otro porgrama determinado por el otro extremo de la flecha.

2.1.2. Curvas de amenaza sísmica

Otro notable ejemplo es el proceso para generar curvas de amenaza sísmica que describen las probabilidades de que ocurra un temblor en una determinada área. Para elaborar estas curvas, los científicos del Centro de Terremotos del Sur de California (SCEC por sus siglas en inglés) tienen que realizar una gran cantidad de simulaciones para que sus resultados puedan ser combinados y expresados en la curva de amenaza [14]. El flujo de trabajo para generar la curva de amenaza se muestra en la figura 2.2.

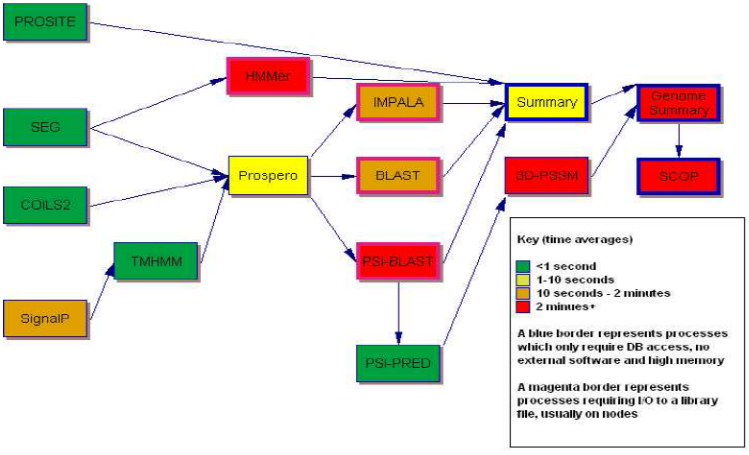

Figura 2.1: Flujo de trabajo para la anotación de proteínas.

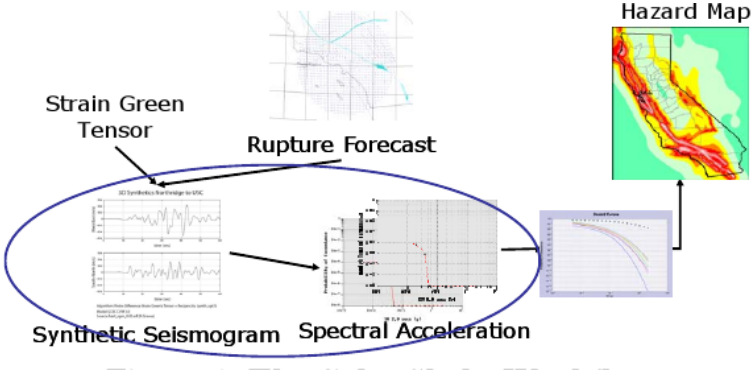

Figura 2.2: Flujo de trabajo para generar curvas de amenaza sísmica.

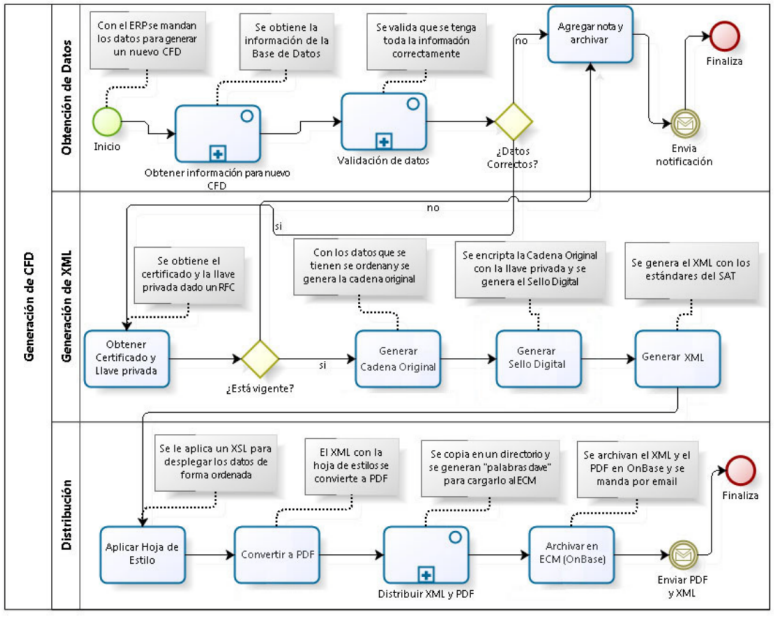

Figura 2.3: Flujo de trabajo para generar facturas electrónicas.

2.1.3. Generación de facturas

Recientemente en México, el Sistema de Administración Tributaria emitió los lineamientos para que las operaciones de compra-venta entre personas físicas y morales puedan ser registradas por medio de facturas electrónicas. Para generar estos Comprobantes Fiscales Digitales, las empresas tienen que hacer procesos de validaciones de RFC y encriptar el contenido de la factura con un mecanismo de llave privada. Naturalmente, este proceso de generación de factura requiere de varias actividades. En la figura 2.3 se muestra el flujo de trabajo que utiliza una empresa para generar sus facturas. Este flujo se encuentra documentado en una tesina presentada por Alcerreca [7].

2.1.4. Procesamiento de imágenes astronómicas

La NASA elaboró un software que permite crear una gran imagen del espacio exterior a partir de varias imágenes mosaico tomadas desde distintos telescopios. Esta aplicación, llamada Montage, utiliza un flujo de trabajo para generar la gran imagen. En la fase inicial de procesamiento, cada una de las imágenes mosaico puede ser procesada de manera independiente. En la figura 2.4 se puede observar que el flujo de trabajo del proyecto Montage inicia con varias actividades en paralelo, donde el

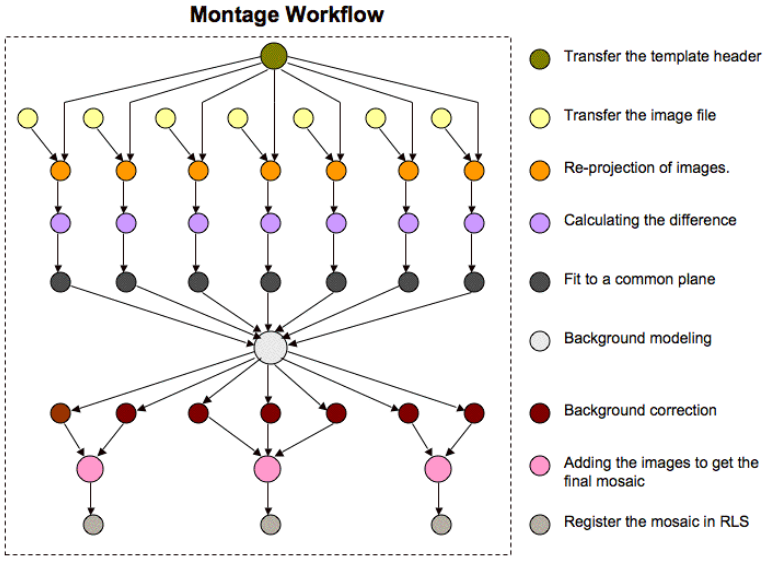

Figura 2.4: Flujo de trabajo para procesamiento de imágenes del proyecto Montage.

color de las actividades representan la fase de procesamiento de la imagen. Luego, en la acividad de color gris, todas las imágenes son juntadas para generar la imagen de fondo común para todas las imágenes mosaico.

2.2. Estructura de los flujos de trabajo

Como hemos visto en los tres ejemplos anteriores, los flujos de trabajo presentados describen los *pasos* que se necesitan para calcular la solución de un problema. En éstos, no se ha hablado sobre los detalles para hacer los cómputos necesarios para cada flujo. En los flujos de trabajo tampoco se han definido las plataformas de cómputo en que se ejecutan estos flujos. Tan sólo se han definido los grandes pasos para solucionar el problema y las dependencias que tienen estos pasos.

Ahora, es muy común que estas dependencias estén dictadas por los datos que requiere cada paso para funcionar. Sin embargo, hay situaciones en los que los pasos del flujo no requieren los datos del paso anterior para funcionar, sino que las dependencias están marcadas por el orden temporal que deben seguir estos pasos. Así, se puede notar una ambigüedad en la definición de un flujo de trabajo.

2.3. Perspectivas de los flujos de trabajo

En la sección 2.2 se argumentó que un flujo de trabajo puede representar varias cosas, a saber: el orden temporal de las tareas o las dependencias de datos entre cada tarea, produciendo una ambigüedad en la interpretación de un flujo de trabajo. Debido a esta ambigüedad del significado de un flujo de trabajo, es posible interpretar un flujo de trabajo desde varias perspectivas. En el trabajo de Van der Aalst et al. [36] se identifican una serie de perspectivas identificadas en las definiciones de las especificaciones de un flujo de trabajo utilizadas en los sistemas que administran la ejecución de flujos de trabajo. Las cuatro perspectivas se enuncian a continuación:

- **Perspectiva de control de flujo.** Describe las relaciones de las actividades (pasos) con estructuras de control, tales como: secuencia, decisión, ejecución de actividades en paralelo y punto de sincronización conjunta. El flujo de trabajo para la generación de facturas es un buen ejemplo de un flujo de trabajo visto desde la perspectiva de control de flujo porque, al observar la figura 2.3, se pueden notar que hay actividades, simbolizadas con un rombo, que determinan si se ejecutan o no ciertas tareas.

- **Perspectiva de datos.** En ella, los flujos de trabajo describen las entradas y salidas de datos, tanto de ejecución como de control, que se tienen en cada actividad del flujo. También se toman en cuenta los datos locales a cada actividad; es decir, que sólo son necesarios dentro del contexto de ésta. En el ejemplo del flujo de trabajo para la anotación de proteínas, mostrado en la figura 2.1, podemos ver que esta perspectiva describe adecuadamente la estructura del flujo, ya que el orden de ejecución de las actividades están definidas por los datos que requiren cada una de las actividades.

- **Perspectiva de recursos.** Muestra cuáles son los recursos con los que se cuentan para ejecutar el flujo de trabajo y la forma en que estos recursos se encuentran organizados. Estos recursos pueden ser desde entidades de cómputo hasta roles con responsabilidades específicas cumplidas por actores humanos. El flujo de trabajo para generar curvas de amenaza sísmica de la figura 2.2 es un ejemplo de esta perspectiva, porque los elemenos necesarios para generar las curvas de amenaza son provistos por geólogos.

- **Perspectiva operacional.** Aquí se detallan las operaciones elementales necesarias en cada actividad para ejecutar el flujo de trabajo. Estos detalles incluyen las transferencias de datos entre las operaciones y su correspondencia en programas. El flujo de trabajo del proyecto Montage, mostrado en la figura 2.4 ejemplifica esta perspectiva, ya que los colores utilizados en el flujo de trabajo son las actividades específicas que se aplican a cada imagen a procesar.

Cabe aclarar que estas perspectivas están relacionadas entre sí de modo que el control de flujo es la base en la que descansan las demás perspectivas. Esto es porque

la perspectiva de datos requiere que el control de flujo tenga los datos de entrada y salida como prueba de que se cumplieron las pre y post-condiciones de cada actividad, respectivamente; la perspectiva de recursos define con qué se ejecutarán y almacenarán los datos del flujo de trabajo; mientras que la perspectiva operacional trata los detalles sobre cómo se utilizan físicamente los recursos, los datos y los programas a lo largo del flujo de trabajo.

2.4. Planificación como control de flujo

El hecho de que la perspectiva de control de flujo sea la base de las demás perspectivas indica que la forma en que controla la ejecución del flujo determina de manera fundamental el rendimiento de la ejecución total de una instancia del flujo de trabajo. Por lo tanto, es de vital importancia encontrar métodos de planificación que permitan encontrar correspondencias entre recursos y actividades que cumplan con los requisitos dictados en las especificaciones examinadas en la perspectiva del control del flujo y que también maximicen el rendimiento de la ejecución de todo el flujo en general.

2.5. Alcance

En este trabajo, se establecerán las siguientes consideraciones para definir el alcance de este estudio de los algoritmos de planificación, a saber: la utilización de grafos dirigidos acíclicos y la suposición de que no habrá errores en la ejecución atómica de las tareas de los flujos de trabajo. En las siguientes se explicarán la importancia de estas suposiciones.

2.5.1. Grafos dirigidos acíclicos

Los flujos de trabajo están representados como *grafos dirigidos acíclicos*, con el objetivo de simplificar el estudio. Se utilizan estos grafos porque son una estructura de datos que representa intuitivamente las dependencias entre tareas. Se requiere que estos grafos sean dirigidos porque esta característica representa el orden de ejecución de las tareas. Además, el hecho de que no existan ciclos en los grafos implica que no se necesita ningún mecanismo de control de flujo condicional; es decir, si la ejecución de las tareas de un flujo de trabajo dependiera de las salidas de otras tareas del flujo, no se podría saber apriori cuál es el orden de ejecución de las tareas, y los algoritmos de planificacíón tendrían que especular cuál sería el posible orden de ejecución.

2.5.2. Ejecución atómica de tareas sin fallos

Las tareas (o actividades) de los flujos de trabajo son consideradas *atómicas*, i.e., una tarea no puede ejecutarse incompletamente ni incorrectamente. Algunos sistemas de administración de ejecución de flujos de trabajo tienen mecanismos para lidiar con estos errores, pero el estudio de éstos queda fuera del alcance de este trabajo,

debido a la complejidad que representa diseñar e implementar sistemas distribuidos tolerantes a fallos. Un ejemplo de un sistema de administración de flujos de trabajo se encuentra en el trabajo de Kandaswamy et al. [18], en el cual describe que utilizan sobreprovisionamiento de recursos y migración de recursos para mitigar las fallas.

Capítulo 3

Cómputo distribuido

Los flujos de trabajo requieren de recursos de cómputo para su ejecución. En los flujos de trabajo de ejemplo del capítulo 2 se mencionaron aplicaciones científicas que, por su naturaleza, requieren un alto poder de cómputo para llevarse a cabo. Por otro lado, también existen aplicaciones de negocio que, si bien son computacionalmente sencillas, se requiere ejecutar una gran cantidad de instancias de estos flujos de trabajo. Por ello, ejecutar cualquiera de estos flujos de trabajo en una sola computadora resulta prohibitivo. Por lo tanto, comúnmente se hace uso de varias computadoras para distribuir la carga computacional necesaria para correr estos procesos.

Dependiendo de cómo estén organizados estas computadoras, se definen los enfoques para correr los flujos de trabajo con cómputo distribuido.

3.1. Enfoques de cómputo para flujos de trabajo

Diversos enfoques se han aplicado para distribuir la ejecución de un flujo de trabajo entre varias computadoras. De acuerdo a Buyya et al., los enfoques de cómputo más importantes para los flujos de trabajo son los *clusters*, los *grids* y las *nubes* [11]. A continuación, explicaremos cada uno de los enfoques.

3.1.1. Clusters

Los *clusters* son sistemas distribuidos, paralelos, compuestos de varias computadoras conectadas entre si que funcionan como un único recurso de cómputo [11].

Un ejemplo de un cluster se puede encontrar en la Universidad Autónoma Metropolitana, campus Iztapalapa. El cluster, llamado *Aitzaloa*, está compuesto por 270 nodos de cómputo, cada uno equipado con dos procesadores Intel Xeon Quad-Core y 16GB en RAM; los nodos están conectados entre sí por medio de switches Ethernet e Infiniband. El cluster también cuenta con un sistema de archivos distribuido basado en Lustre. La capacidad real de cómputo del cluster Aitzaloa es de 18.4 teraFLOPS [4].

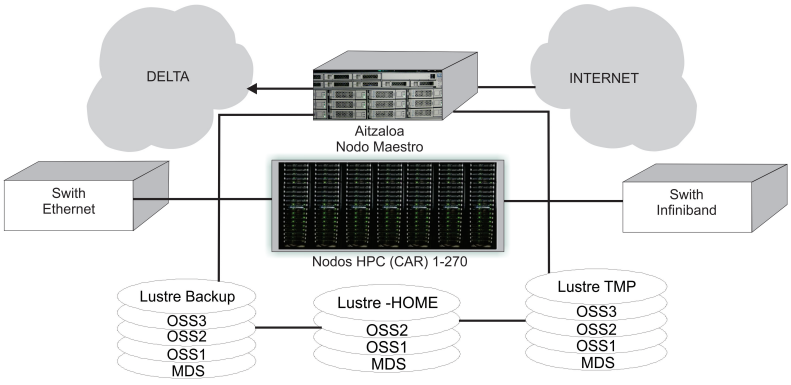

Figura 3.1: Topología del cluster Aitzaloa

El detalle de la topología del cluster se puede apreciar en la figura 3.1, en donde podemos apreciar que los switches son los puntos de conexión entre el nodo maestro, el sistema de almacenamiento distribuido y los nodos de cómputo.

3.1.2. Grids

Los *grids* son sistemas distribuidos, paralelos, compuestos de computadoras autónomas y geográficamente distribuidas que pueden trabajar en conjunto o de manera independiente de acuerdo a los objetivos, políticas y mecanismos de uno o varios administradores del sistema, es decir, un grid puede ser compartido entre varias instituciones [11].

El proyecto *LANCAD*[1] es un buen ejemplo, pues une el cluster *Aitzaloa* de la UAM, el cluster de la UNAM *KamBalam*, y el cluster *Xiuhcoatl* del CINVESTAV por medio de una red de fibra óptica instalada en las estaciones del Sistema de Transporte Colectivo Metro. La suma de la potencias reales de cada *nodo robusto* del grid es de 48.55 teraFLOPS [2].

En la figura 3.2 se muestra los tramos de línea de Metro que cuentan con fibra óptica para conectar cada uno de las supercomputadoras (clusters) de las tres instituciones.

3.1.3. Nubes

Las *nubes* (clouds) son sistemas distribuidos, paralelos, compuestos de computadoras o máquinas virtuales interconectadas que son aprovisionadas para usarse como uno o varios recursos de cómputo, de acuerdo a un contrato de nivel de servicio acordado entre el proveedor de la nube y el cliente [11].

[1] Laboratorio Nacional de Cómputo de Alto Rendimiento

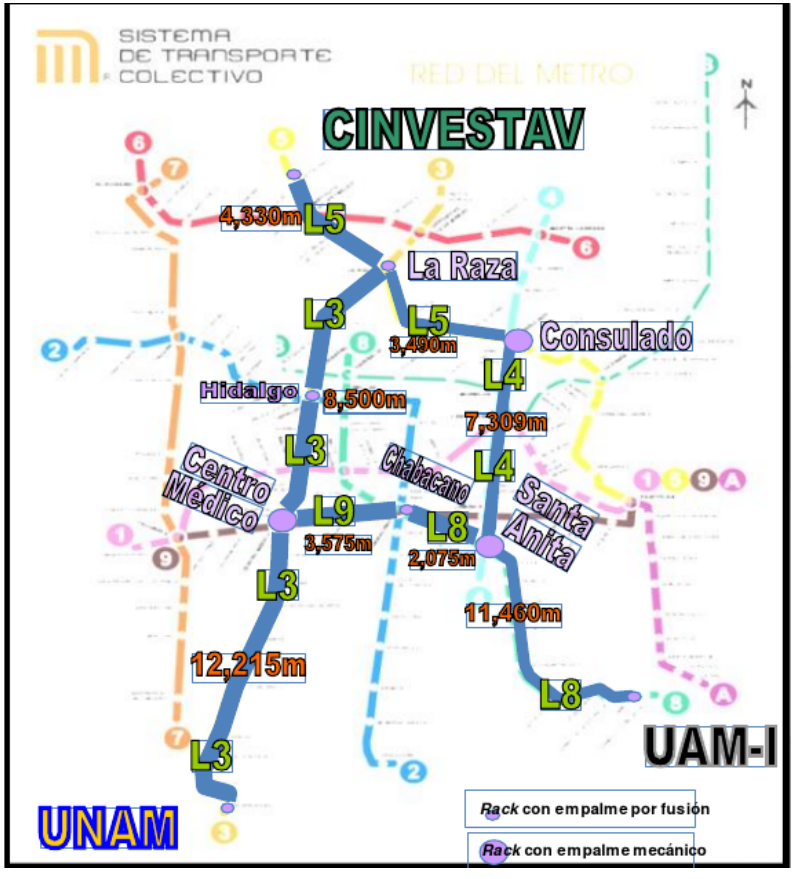

Figura 3.2: Red de fibra óptica del grid LANCAD.

Empresas nuevas y existentes proveen servicios de cómputo en la nube, tales como GoGrid, Rackspace, Amazon, Microsoft, IBM, Oracle, entre otras. La forma en que operan es la siguiente: se paga cierta cantidad por utilizar servicios de cómputo o almacenamiento durante determinado tiempo. Así, los clientes no tienen que invertir grandes cantidades de dinero para contar con una gran infraestructura como en el caso de los los clusters y los grids.

3.2. Comparación de los enfoques

Si bien estos enfoques difieren, principalmente, en la forma en que están organizadas las unidades de cómputo, podemos enumerar algunas observaciones.

Primero, es natural ver que los grids están compuestos de clusters, pero también podría verse a un grid como un gran cluster. Hasta cierto punto este razonamiento parece correcto. Sin embargo, la diferencia importante entre clusters y grids radica en el hecho de que el grid es comúnmente administrado por varios técnicos de varias instituciones que, pueden tener objetivos y necesidades diversas, mientras que un cluster requiere de menos administradores y se asume una mayor flexibilidad para enfrentar fallos. [11]

Segundo, tanto los grids como las nubes son físicamente muy similares: ambos consisten de varios clusteres interconectados y distribuidos geográficamente. Entonces, ¿cuál es la diferencia? Lo que distingue a las nubes de los grids son dos características: (1) en una nube, se utilizan *máquinas virtuales* para distribuir el recurso. El usuario tiene la vista de una parte del grid como si fuera una computadora dedicada exclusivamente para éste. Por otro lado, en los grids, es común que el usuario acceda a él con una cuenta asignada por un administrador, de tal suerte que se tiene una vista de una gran máquina que es compartida entre varios usuarios. Muchos de los administradores de grids en el mundo administran los trabajos encargados por los usuario de tal modo que se ejecuten lo más rápido posible con los recursos disponibles en un tiempo dado. Los usuarios, en algunos casos obtienen acceso al grid y trabajan sus proyectos de manera gratuita o, en caso contrario, pagan una cuota por un tiempo fijo de cómputo. Pero, en el caso de las nubes, se agrega un modelo económico para acceder a los recursos llamado *pay as you go*, en donde el usuario paga una cuota por los recursos utilizados, sin alguna limitación mas que el presupuesto. Asi, el usuario podría pagar una gran cantidad para que su flujo de trabajo se ejecute en el menor tiempo posible o también podría pagar una menor cantidad sacrificando el tiempo total de ejecución del flujo.

Capítulo 4

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. En este capítulo se define de la forma más elemental el problema de la planificación para estudiar sus propiedades y también se explica la relación de este problema fundamental con los flujos de trabajo.

4.1. Definición del problema

De acuerdo a Ullman et al. [35], el problema de la planificación se define de la siguiente manera:

Definición 1. *El **problema de la planificación** está compuesto por:*

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

Para Ullman et al. [35], el objetivo del problema de la planificación es *minimizar* el tiempo total de ejecución, denotado por $t_{\max}$, respetando el orden parcial definido por $\prec$, asignando tareas de S 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.

4.2. La complejidad de planificar

Como puede notarse, hay varias maneras de acomodar las k computadoras para que se ejecuten todas las tareas de S. De hecho, Ullman et al. han demostrado que este problema pertenece a la categoría NP-completo [35]. Esto signifca 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 costoso.

Así, la forma de atacar estos problemas NP-completos es utilizar métodos de aproximación [19] que obtengan soluciones subóptimas o utilizar heurísticas que resuelvan este problema asumiendo ciertas restricciones.

4.3. Planificación de flujos de trabajo

Hasta ahora, se ha mencionado el problema básico de planificación con restricciones. Con ello, se pretende plantear el problema de la planificación de flujos de trabajo y apoyarse en la definición del problema básico de planificación para enumerar propiedades sobre este problema.

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.

4.3.1. Reducción de flujos de trabajo a grafos dirigidos acíclicos

En el trabajo de Mair et al. [25] 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. Como se vio en el capítulo 2, 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.

Cabe aclarar que, como se vio en el primer capítulo de este trabajo, aún no existe un conseso sobre qué representa un flujo de trabajo [36], debido a las diferentes perpectivas de interpretación de un flujo. En el caso del lenguaje AGWL, éste cuenta con facilidades para poder expresar flujos de trabajo condicionales, a saber: constucciones 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. En algunos sistemas de administración de ejecución de flujos de trabajo, como Pegasus [15], aplican desenroscado de bucles como técnica de estimación de la ruta de ejecución del flujo, para transformar el flujo de trabajo condicional en un grafo dirigido acíclico. Sin embargo, el estudio de estos mecanismos está fuera del alcance de este trabajo.

De esta forma, es posible asumir, tanto para el estudio del problema de planificación de flujos de trabajo como para los algoritmos de planificación, que se tendrá como entrada la representación del flujo de trabajo en forma de un grafo dirigido acícliclo.

4.3.2. 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 la definiciones de Wieczorek-Prodan [38] para definir el problema de la planificación de flujos de trabajo:

Definición 2. *Un **flujo de trabajo** es un grafo dirigido $w \in \mathcal{W}, w = (\mathcal{V}, \mathcal{E})$, 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 que 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.

Ahora, se definirán los recursos de cómputo en donde se ejecuta el flujo.

Definición 3. *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, la planificación es definida como una función:

Definición 4. *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 calendarizaciones de w es denotado por $\mathcal{F}$.*

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

Definición 5. *Un **modelo de costos** es un conjunto de criterios $C = \{c_1, c_2, \ldots, c_n\}$ 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 6. *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.*

Definición 7. *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 calendarizado por f un costo basado en los costos parciales determinados por los servicios utlizados para ejecutar las tareas del flujo.*

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, $1 \leq i \leq n$.

4.3.3. Complejidad computacional de la planificación de flujos de trabajo

Ahora, se hará una analogía con el problema básico de planificación visto en la sección 4.1 para estudiar su complejidad computacional. La analogía es la siguiente:

1. El conjunto de tareas S 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 debe 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 básico 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. Por lo pronto, 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 básico 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 básico de planificación. Entonces, se puede inferir que el problema de la planificación de flujos de trabajo pertenece a la categoría de problemas NP-completo.

Capítulo 5

Algoritmos de planificación de flujos de trabajo

En el capítulo anterior definimos el problema de planificar flujos de trabajo. También se vio que dicho problema pertenece a la categoría de los problemas NP-completos, lo cual significa que la complejidad –o tiempo de ejecución– para resolver este problema no está acotada por una función polinomial. Por ello, diversos algoritmos se han propuesto para atacar, ya sea de manera total o parcial, el objetivo principal de la planificación: minimizar los costos, tales como el tiempo total de ejecución o un presupuesto.

Ahora, con los enfoques de cómputo propuestos en el capítulo 2, se crean diversos algoritmos para diferentes necesidades. Mientras que en los clusters y los grids se asumen que estos recursos son compartidos, los algoritmos de planificación para estos recursos asumen que los flujos deben ser ejecutados lo más pronto posible, con el fin de hacer que el recurso esté ocupado la mayor parte del tiempo. Por otro lado, el enfoque de nubes permite al diseñador de flujos de trabajo elegir entre ejecutar el flujo con todos los recursos a costa de un elevado presupuesto o, reducir dicho presupuesto a costa de un tiempo de ejecución más elevado.

5.1. Clasificación de los algoritmos de planificación

Como han aparecido numerosos algoritmos para planificar flujos de trabajo, es natural que surjan clasificaciones [34] [42] 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. [42], se pueden clasificar los algoritmos de planificación de flujos de trabajo ejecutados en grids en dos grandes niveles: los algoritmos de Mejor Eesfuerzo y los algoritmos de Calidad en el Servicio[1]. Al primer grupo pertenecen

[1]En la literatura especializada se conoce a este término como *Quality of Service*

aquellos algoritmos que tratan de minimizar el tiempo total de ejecución[2], 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. A continuación, mostraremos los algoritmos descritos en la clasificación de Yu et al.:

5.2. Algoritmos de mejor esfuerzo

Este tipo de 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 [42]. El segundo subgrupo contiene a los algoritmos que utilizan algún método para elegir o generar planificaciones que se acerquen al óptimo.

5.2.1. Algoritmos heurísticos de mejor esfuerzo

Los algoritmos heurísticos se pueden dividir en cuatro grandes tipos:

1. algoritmo inmediato,
2. algoritmos basados en lista,
3. agrupamiento de tareas y
4. duplicación de tareas.

El primer grupo corresponde al algoritmo miope, que sólo asigna tareas a recursos conforme se va necesitando. Los algoritmos basados en lista 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, 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. Finalmente, los algoritmos de duplicación de tareas 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.

[2]También conocido como *makespan*

5.2.2. Algoritmos metaheurísticos de mejor esfuerzo

Los algoritmos metaheurísticos son clasificados de acuerdo a la metaheurística que utilizan. Para esta clasificación, vamos a describir tres tipos de heurísticas:

1. algoritmos genéticos,
2. búsqueda aleatoria adaptativa dirigida –GRASP– y
3. recocido simulado.

Los algoritmos genéticos simulan el proceso de selección natural con posibles calendarizaciones 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. El algoritmo GRASP[3] genera aleatoriamente soluciones y con un prodecimiento voraz[4] elige las soluciones óptimas locales. El recocido simulado, como su nombre lo indica, emula el proceso de formación de cristales donde en cada iteración se va creando una solución más óptima.

5.3. 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 prespuesto para cada tarea particular del flujo.

De acuerdo a la forma que trabajan estos algoritmos, pueden dividirse en: restringidos por presupuesto o restringidos for 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 calendarizaciones que cumplan con fechas proporcionadas. Ambos grupos se subdividen en heurísticos y metaheurísticos. A continuación, hablaremos de estas subdivisiones.

5.3.1. Algoritmos restringidos por presupuesto

Los algoritmos restringidos por presupuesto se dividen en:

1. heurísticos y
2. metaheurísticos.

[3]GRASP son las siglas en inglés de *Greedy Randomized Adaptive Search Procedure*
[4]Greedy

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. [31]. Los algoritmos metaheurísticos generan soluciones aleatoriamente y utilizan la información de iteraciones pasadas para refinar las soluciones. En el trabajo de Yu et al. [41] proponen un algoritmo genético cuya función de evaluación valora mejor a las calendarizaciones que cumplan con el presupuesto con un tiempo de ejecución mínimo.

5.3.2. Algoritmos restringidos por fechas límites

Los algoritmos restringidos por fechas límites también se dividen en:

1. heurísticos y
2. metaheurísticos.

En el primer grupo se encuentran el algoritmo BackTracking, propuesto por Menascé et al. [26], y el algoritmo de distribución de fechas límite, propuesto por Yu et al. [43]. En el segundo grupo se encuentra un algoritmo genético basado en [41], cuya función fitness valora mejor a aquellas soluciones que cumplan con la fecha límite, con un presupuesto mínimo.

5.3.3. Estimación del tiempo

En los algoritmos vistos en esta sección, se asume que se conoce el tiempo de ejecucion de una tarea. 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. 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 [21].

Capítulo 6

Software para la administración y ejecución de flujos de trabajo

En el capitulo anterior se extendió la clasificación de algoritmos de planificación elaborada por Yu et al. [42]. Sin duda, estos algoritmos han sido probados en simulaciones e implementados en sistemas que administran la ejecucion de los flujos de trabajo. De acuerdo a Yu el al. [42], un sistema de administración de flujos de trabajo[1] se encarga de definir, coordinar y ejecutar los flujos de trabajo en los recursos de cómputo.

Para este trabajo, clasificaremos los sistemas de administración de flujos de trabajo de acuerdo a su enfoque de cómputo, enumerando las siguientes características: año de aparición, proyecto, utilización, autores y los algoritmos de planificación utilizados en estos sistemas.

6.1. Software orientado a clusters

6.1.1. HTCondor

HTCondor [33] es un sistema para administrar flujos de trabajo que necesitan ser ejecutados en entornos de cómputo distribuido. Desde 1983 [3], HTCondor es desarrollado y mantenido por la Universidad de Wisconsin en Madison. Éste tiene un modulo llamado DAGman que se encarga de planificar tareas de un flujo de trabajo, expresadas como grafos dirigidos acíclicos. El algoritmo de planificación utilizado por DAGMan es el algoritmo miope.

[1] En la literatura académica, estos sistemas son conocidos como *workflow management sofware systems*

HTCondor presenta al usuario un único recurso de cómputo, formado de varias computadoras interconectadas entre si. Esta es la razón por la que este sistema de administración de flujos de trabajo cae dentro de la clasificación de software orientado a clusters.

6.2. Software orientado a grids

6.2.1. SwinDew-G

SwinDew-G [40] es un sistema de administración de flujos de trabajo cuya característica especial es que trabaja con redes de computadoras P2P. En la primera version de SwinDew, el proceso de planificación es estático [40], i.e., en la fase de preparación de la ejecución se crea el plan de ejecución del flujo y se aplica sin ninguna modificación a lo largo del tiempo. Varios esfuerzos se han hecho para mejorar el mecanismo de planificación, como el algoritmo CTC [20] diseñado especialmente para flujos de trabajo que son intensivos en instancias pero que contienen pocas tareas simples, como los flujos de trabajo utilizados en los bancos y empresas [21].

Finalmente, SwinDew y sus variantes fueron desarrollados en la Universidad de Swinburne, en Australia.

6.2.2. Pegasus

Pegasus [15] fue desarrollado en la Universidad del Sur de California. Al igual que SwinDew-G, se puede utilizar Pegasus para trabajar con grids y con nubes. Pegassus implementa max-min, min-min y Sufragio como algoritmos de planificación. Las principales aplicaciones de Pegasus son los flujos de trabajo científicos. Los primeros trabajos publicados que reportan el uso de Pegasus datan del 2003 [6]. Dentro de la arquitectura de Pegasus, el módulo encargado de planificar el flujo de trabajo es el Mapper.

6.3. Software orientado a nubes

6.3.1. SwinDew-C

SwinDew-C [22] es un sistema de administración de flujos de trabajo basado en SwinDew-G, cuya principal característica es que agrega nubes externas como recursos disponibles para la ejecución de flujos de trabajo. La arquitectura de SwinDew-C es muy similar a la arquitectura de SwinDew-G, con la diferencia de que agrega nuevos componentes para manejar restricciones de calidad de servicio y administración de fallos. En el caso de una falla en la ejecución de las tareas, SwinDew-C replanifica la ejecución de la tarea fallida con un algoritmo basado en optimización por colonia de hormigas.

6.3.2. Aneka

Aneka [12] es una implementación de una plataforma como servicio que, además de poder especificar y ejecutar flujos de trabajo, también puede planificar y ejecutar aplicaciones distribuidas que hayan sido construidas con distintos modelos de programación distribuida: basado en tareas, basado en threads y basado en MapReduce. El modelo de programación que se utiliza en Aneka para trabajar con flujos de trabajo es el modelo basado en tareas.

El diseño de Aneka es basado en servicios. Cada servicio se encarga de una funcionalidad específica para la administración del grid o nube. Así, existe un módulo específico para la planificación de la ejecución de las aplicaciones que varía de acuerdo al modelo de programación elegido para desarrollar la aplicación.

Técnicamente, es posible construir una nube con Aneka [37] utilizando computadoras de escritorio convencionales. De esta forma, Aneka provee servicios para administrar el esquema de precios de la nube construida con este software. Del mismo modo, Aneka utiliza un mecanismo de negociación mediante agentes que buscan *cerrar* los mejores tratos con los proveedores de los servicios de la nube, que mejor satisfagan los requisitos de calidad en el servicio.

Es importante mencionar que la plataforma Aneka fue desarrollada inicialmente por el laboratorio GRIDS de la Universidad de Swinburne, en Austraila. Actualmente, Aneka es desarrollado por la empresa Manjrasoft [5].

6.3.3. Askalon

Askalon [16] es un entorno que facilita el desarrollo y la ejecución de flujos de trabajo. Askalon utiliza un lenguaje llamado AGWL para especificar flujos de trabajo. Al igual que Aneka, el diseño de Askalon está basado en servicios, tales como: el motor de promulgación, el administrador de recursos, la base de datos de checkpoints y el planificador.

En el servicio planificador, Askalon utiliza varios algoritmos de planificación: HEFT, miope y un algoritmo genético. Esto es porque antes de ejecutar el flujo de trabajo, se hace una predicción del tiempo estimado de ejecución de cada tarea. Entonces, hay un módulo encargado de actualizar la información de las predicciones y, en base al resultado de los tres algoritmos, se elige la planificación que mejor cumpla con los requisitos de calidad en el servicio dictados por el usuario.

Actualmente, el proyecto Askalon es desarrollado por la Universidad de Innsbruck, en Austria [1].

Capítulo 7

Implementación

Con el fin de experimentar con los algoritmos de planificación de flujos de trabajo, se implementaron algunos algoritmos descritos en este trabajo. Los algoritmos fueron implementados en Java.

Se implemetaron los algoritmos Miope, MinMin y MaxMin. El algoritmo Miope es un algoritmo voraz que sólo busca el recurso que pueda ejecutar la tarea lo más pronto posible. Los algoritmos MinMin y MaxMin buscan recursos que puedan minimizar tanto la duración de la tarea como el tiempo en el que inician las tareas. Luego, en el caso de MaxMin, se busca la tareas que sean más tardada de ejecutar ó, en el caso de MinMin, se busca la tarea que tarde menos en ejecutarse.

Ahora, se hicieron dos supocisiones para simplificar la implementación:

1. Los recursos pueden ejecutar todas las tareas.
2. Para el caso de los algoritmos MaxMin y MinMin, el tiempo de disponibilidad de transferencia de archivos es el tiempo en que las tareas inmediatamente precedentes han sido ejecutadas, asumiendo que la transferencia de datos entre nodos es instantánea.
3. Una tarea es considerada lista para planificarse si todos sus predecesores inmediatos han sido planificados.
4. Los recursos ejecutan una sola tarea a la vez.

7.1. Descripción del código

El diseño de las clases y objetos se hizo de tal modo que los elementos más comunes de los algoritmos fueran representados en una clase. Así, describiremos primero las clases `Task`, `Resource` y `Schedule`, para luego describir la clase `Workflow` y las clases que implementan los algoritmos: `Myopic`, `MinMin` y `MaxMin`. En la figura 7.1

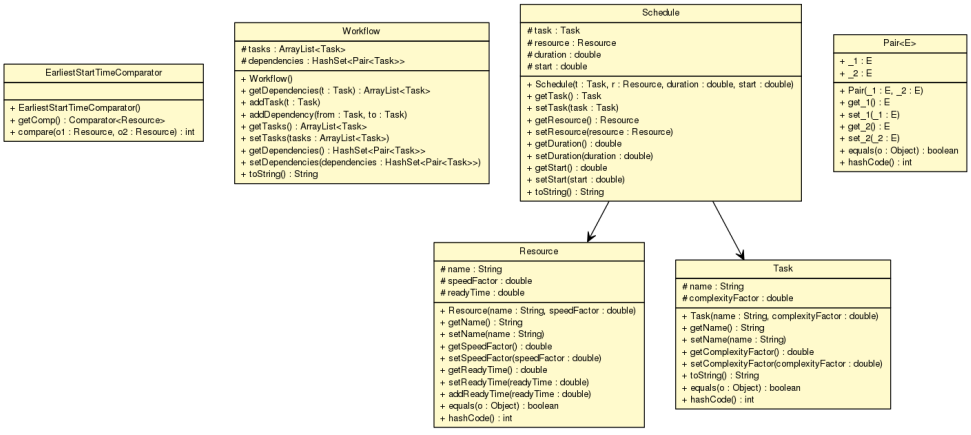

Figura 7.1: Diagrama de clases UML de las clases principales.

se puede apreciar el diagrama de clases UML de las clases que modelan los elementos del problema de planificación de flujos de trabajo.

7.1.1. Clase `Task`

La clase `Task` representa una tarea que forma parte de un flujo de trabajo. Cada tarea tiene un nombre que debe ser único en el flujo de trabajo del que forma parte. También, la tarea contiene un factor de complejidad, que es un número que representa qué tan complicado es ejecutar esta tarea. La idea básica es que a mayor factor de complejidad, mayor es el tiempo requerido para ejecutar la tarea o un recurso más rápido es necesario para ejecutar la tarea.

7.1.2. Clase `Resource`

En esta clase se representa a un recurso que puede ejecutar tareas de un flujo de trabajo. De forma similar a la clase `Task`, un recurso tiene un nombre –que no es necesario que sea único– y un factor de velocidad que indica la rapidez con la que dicho recurso puede ejecutar una tarea.

7.1.3. Clase `Schedule`

Representa una planificación de una tarea en un recurso. Cada objeto de esta clase guarda la tarea y el recurso que fueron asignados, el tiempo en que se empezará a ejecutar la tarea y el tiempo estimado de ejecución de la tarea en el recurso.

7.1.4. Clase `Workflow`

La clase `Workflow` representa un flujo de trabajo, compuesto de tareas y las relaciones de precedencia entre tareas. Cabe aclarar que no se permiten que existan tareas con el mismo nombre y también se verifican que las relaciones de precedencia entre tareas formen un grafo dirigido acícilico.

7.2. Código de los algoritmos

En esta sección, detallaremos las decisiones de diseño que fueron necesarias para implementar los algoritmos.

7.2.1. Algoritmo Miope

Como se vió en el capítulo 5 y en la sección A.1.1 del Apéndice, el algoritmo Miope asigna tareas a recursos de tal modo que se empiecen a ejecutar lo más pronto posible. Aunque la idea básica resulte fácil de entender, hay algunas cuestiones que deben ser resueltas para implementar este algoritmo.

Primero, hay que tomar en cuenta cómo obtener tareas que estén listas para planificarse. Para ello, se planifican sólo aquellas tareas cuyos predecesores inmediatos o padres han sido planificados. Cabe aclarar que se entiende que una tarea está planificada si ésta ya está asignada a un recurso, por lo que no se permite que una tarea sea asignada a varios recursos. De esta forma para generar la lista de tareas a planificar, se seleccionan aquellas tareas que no han sido planificadas y cuyos padres ya han sido planificados. Este último requerimiento asegura que se respeten las dependencias entre tareas a la hora de planificar. está implementado en el método estático `Utils.checkParents()`. Por otro lado, para verificar si una tarea está planificada, se excluyen de la lista de tareas a planificar aquellas tareas ya planificadas.

En la figura 7.2.1 está el código en Java del algoritmo miope, utilizando las clases antes descritas. Cabe notar que para conocer cuál es el recurso más disponible, se mantiene una cola de prioridad de los recursos con el tiempo de disponibilidad más pronto.

7.2.2. Algoritmo MaxMin y MinMin

Para los algoritmos MaxMin y MinMin se utilizaron varias métodos desarrollados para el algoritmo miope. Por ejemplo, el método `Utils.parentsReadyTime()` utilizado para obtener cuándo terminaran de ejecutarse todas las tareas que son predecesores inmediados de una tarea en flujo de trabajo dada una calendarización. Como se vio en la sección A.1.2 del Apéndice, hay algunas definiciones que son comunes para ambos algoritmos. En la figura 7.2.2 están codificadas dichas definiciones.

Ahora, como ambos algoritmos son muy similares, se hizo un sólo método para ambos algoritmos, con un parámetro para elegir si se ejecuta la porción correspon-

```
1  List<Schedule> schedule(Workflow w, List<Resource> resourceList) {
2    Utils.checkScheduleParams(w, resourceList);
3    //lista de tareas no calendarizadas
4    ArrayList<Task> readyTasks = new ArrayList<Task>(w.getTasks());
5    ArrayList<Task> schedTasks = new ArrayList<Task>();
6    ArrayList<Schedule> schedules = new ArrayList<Schedule>();
7    //heap para mantener a los recursos mas pronto disponibles
8    PriorityQueue<Resource> R = new PriorityQueue<Resource>(resourceList.size()
         , EarliestStartTimeComparator.getComp());
9    R.addAll(resourceList);
10   Task t; Resource r;
11   while(!readyTasks.isEmpty()) {
12     Iterator<Task> it = readyTasks.iterator();
13     if(it.hasNext()) {
14       t = it.next();
15       if(Utils.checkParents(t, schedTasks, w)) {
16         r = R.peek();
17         double d = t.getComplexityFactor() / r.getSpeedFactor();
18         double st = Math.max(r.getReadyTime(), Utils.parentsReadyTime(t,
               schedules, w));
19         Schedule s = new Schedule(t, r, d, st);
20         readyTasks.remove(t);
21         schedTasks.add(t);
22         r.addReadyTime(d);
23         //Actualizamos heap
24         R = new PriorityQueue<Resource>(resourceList.size(),
               EarliestStartTimeComparator.getComp());
25         R.addAll(resourceList);
26         schedules.add(s);
27       }
28     }
29   }
30   return schedules;
31 }
```

Figura 7.2: Código en Java del algoritmo Miope

```
1  /** Estimated Execution Time */
2  private static double EET(Task t, Resource r) {
3    return t.getComplexityFactor() / r.getSpeedFactor();
4  }
5
6  /** Estimated Availability Time */
7  private static double EAT(Task t, Resource r) {
8    return r.getReadyTime();
9  }
10
11 /** File Available Time */
12 private static double FAT(Task t, Resource r, Workflow w, List<Schedule>
       partialSchedule) {
13   //le vamos a dar una "semantica" diferente
14   //tomamos el tiempo de los padres como el tiempo de archivos listo
15   return Math.max(r.getReadyTime(), Utils.parentsReadyTime(t, partialSchedule
         , w));
16 }
17
18 /** Estimated Completion Time */
19 private static double ECT(Task t, Resource r, Workflow w, List<Schedule>
       partialSchedule) {
20   return EET(t, r) + Math.max(EAT(t,r), FAT(t,r, w, partialSchedule));
21 }
```

Figura 7.3: Código de las definiciones comunes para MaxMin y MinMin.

diente a MaxMin o MinMin, el cual es de tipo `XMinAlgorithm`. En la figura 7.2.2 se muestra el punto de entrada común de los algoritmos MaxMin y MinMin.

Luego, en la figuras 7.2.2, 7.2.2 y 7.2.2 está codificada la rutina que planifica las tareas. Note que en donde está la sentencia `switch` se ejecuta el código que es correspondiente a MaxMin o a MinMin.

```
1  List<Schedule> schedule(Workflow w, List<Resource> resourceList,
       XMinAlgorithm algorithm) {
2    Utils.checkScheduleParams(w, resourceList);
3
4    ArrayList<Task> schedTasks = new ArrayList<Task>();
5    ArrayList<Task> allTasks = new ArrayList<Task>(w.getTasks());
6    ArrayList<Task> readyTasks = null;
7    List<Schedule> scheduleList = new ArrayList<Schedule>();
8
9    while(!allTasks.isEmpty()) {
10   //puedes sacar subconjuntos de todas las tareas
11   readyTasks = new ArrayList<Task>();
12   for(Task t: allTasks)
13     if(!schedTasks.contains(t) && Utils.checkParents(t, schedTasks, w))
14     readyTasks.add(t);
15   scheduleList = scheduleXMin(w, resourceList, readyTasks, scheduleList,
         algorithm);
16   for(Schedule s: scheduleList) {
17     Task t = s.getTask();
18     if(!schedTasks.contains(t))
19     schedTasks.add(t);
20     if(allTasks.contains(t))
21     allTasks.remove(t);
22   }
23   }
24
25   return scheduleList;
26 }
```

Figura 7.4: Método principal en Java para los algoritmos MaxMin y MinMin

```
1  List<Schedule> scheduleXMin(Workflow w, List<Resource> resourceList,
2      List<Task> availTasks, List<Schedule> partialSched,
3      XMinAlgorithm algorithm) {
4    int n_tasks = availTasks.size(),n_resources = resourceList.size();
5    double ECT[][] = new double[n_tasks][n_resources];
6    double MCT[] = new double[n_tasks];
7    int min_mct_r_idx = -1, x_t_idx = -1; double min_mct, x_mct;
8    while(!availTasks.isEmpty()) {
9      int R_t_idx[] = new int[availTasks.size()];
10     for(int i_t=0; i_t<availTasks.size(); i_t++) {
11       //todos los recursos pueden ejecutar todas las tareas
12       Task t = availTasks.get(i_t);
13       min_mct_r_idx = -1; min_mct = Double.MAX_VALUE;
14       for(int i_r=0; i_r<resourceList.size(); i_r++) {
15         Resource r = resourceList.get(i_r);
16         ECT[i_t][i_r] = ECT(t, r, w, partialSched);
17         if(ECT[i_t][i_r] < min_mct) {
18           min_mct = ECT[i_t][i_r];
19           min_mct_r_idx = i_r;
20         }
21       }
22       MCT[i_t] = min_mct;
23       R_t_idx[i_t] = min_mct_r_idx;
24     }
25     //continua en la siguiente figura...
```

Figura 7.5: Método de MaxMin/MinMin que planifica las tareas (parte 1)

```
1       //...continuacion
2       switch(algorithm) {
3         case MaxMin: //Max-Min: get a task with maximum ECT ( t , r ) over
              tasks
4           x_t_idx = -1; x_mct = Double.MIN_VALUE;
5           for(int i_t=0; i_t<availTasks.size(); i_t++) {
6             if(ECT[i_t][R_t_idx[i_t]] > x_mct) {
7               x_mct = ECT[i_t][R_t_idx[i_t]];
8               x_t_idx = i_t;
9             }
10          }
11          break;
12        case MinMin: //Min-Min: get a task with minimum ECT ( t , r ) over
              tasks
13          x_t_idx = -1; x_mct = Double.MAX_VALUE;
14          for(int i_t=0; i_t<availTasks.size(); i_t++) {
15            if(ECT[i_t][R_t_idx[i_t]] < x_mct) {
16              x_mct = ECT[i_t][R_t_idx[i_t]];
17              x_t_idx = i_t;
18            }
19          }
20          break;
21      }
22      //continua en la siguiente figura...
```

Figura 7.6: Método de MaxMin/MinMin que planifica las tareas (parte 2)

```
1       //...continuacion
2       //Schedule T on R_T
3       Task t = availTasks.get(x_t_idx);
4       Resource r = resourceList.get( R_t_idx[x_t_idx] );
5       double duration =  t.getComplexityFactor() / r.getSpeedFactor();
6       double startTime = Math.max(r.getReadyTime(), Utils.parentsReadyTime(t,
            partialSched, w));
7       Schedule s = new Schedule(t,r , duration, startTime);
8       partialSched.add(s);
9       //remote T from availTasks
10      availTasks.remove(t);
11      //update EAT(R_t)
12      //r.addReadyTime( Math.max(duration, Utils.parentsReadyTime(t,
            partialSched, w)) );
13      r.addReadyTime( duration );
14    }
15    return partialSched;
16  }
```

Figura 7.7: Método de MaxMin/MinMin que planifica las tareas (parte 3)

Capítulo 8

Conclusiones

En este trabajo hemos revisado el problema de planificar flujos de trabajo. Primero se describió el concepto de flujo de trabajo, ilustrándolo con algunos ejemplos de aplicación. También vimos que la descripción de las tareas y las dependencias entre ellas juegan un papel muy importante a la hora de planear y administrar los recursos asigados a cada tarea del flujo de trabajo.

Además, se mostró que los flujos de trabajo son ejecutados en sistemas de cómputo distribuido, los cuales se clasificaron en clusters, grids y nubes. Cabe aclarar que esta clasificación está basada en la forma en que los recursos están distribuidos.

Después se estudió la complejidad de planificar flujos de trabajo, comparando este problema con otro problema de planificación más general, orientado a ciencias en computación. Como este problema es NP-completo, se han propuesto varias soluciones heurísticas y metaheurísticas que resuelven el problema bajo ciertas circunstancias.

También, con el objetivo de comprender las diferentes heurísticas propuestas para planificar flujos de trabajo, se utilizó un modelo propuesto en el trabajo elaborado por Wieckzorek et al. para describir de la forma más general posible un flujo de trabajo.

Con el modelo establecido, se realizó una clasificación de los algoritmos de planificación, la cual está basada en el trabajo de Yu et al.; la aportación de este trabajo fue dar otro punto de vista sobre la forma de clasificar las metaheurísticas, ya que éstas, dependiendo de cómo son programadas, pueden hacer optimizaciones que minimicen algún costo (es decir, hacen el mejor esfuerzo por otimizar el costo) o buscar soluciones que cumplan con restricciones (en otras palabras, mantienen la calidad en el servicio).

Finalmente, se describieron algunos sistemas de administración de flujos de trabajo y los algoritmos de planificación que utilizan estos sistemas, clasificándolos de acuerdo a la orientación que tienen los algoritmos para planificar, en otras palabras, si la planificación está pensada para utilizarse en clusters, grids o nubes. Como se puede notar, en estos sistemas puede verse claramente la unión de las dos teorías sobre enfoques de cómputo distribuido y algoritmos de planificación.

8.1. Retos

La planificación es una pieza clave para ejecutar flujos de trabajo en entornos distribuidos. Sin embargo, existen otras cuestiones que deben ser tomadas en cuenta para utilizar estos algoritmos en un sistema real. La primera de ellas es la tolerancia a fallos. Su importancia radica en que en presencia de fallas se puede replanificar el flujo o tratar de recuperar los resultados o el recurso fallido. Sin duda, este es un tema de investigación muy activo en el área de sistemas distribuidos. Otro aspecto a tomar en cuenta es la seguridad, dado que estos sistemas pueden procesar información sensible. Además, el monitoreo de estos sistemas es otro tema a tratar, ya que la complejidad de los sistemas distribuidos y la cantidad de tareas que coordinan los sistemas de administración de flujos de trabajo presentan una complejidad exhorbitante.

8.2. Trabajo futuro

Como trabajo futuro, se hará un algoritmo de planificación de flujos de trabajo que pueda manejar restricciones en calidad en el servicio, con especial orientación al enfoque de cómputo distribuido de la nube, ya que, como se vió en este trabajo, hay una gran área de investigación para explorar y diseñar algoritmos que tengan en cuenta las características únicas de las nubes.

Apéndice A

Pseudocódigos de algoritmos de planificación de flujos de trabajo

A.1. Algoritmos de mejor esfuerzo

A.1.1. Miope

El algoritmo miope [42] es el más simple de todos los algoritmos. Lo único que hace es buscar un recurso disponible que pueda ejecutar la tarea y asignarle dicha tarea. No toma en cuenta otra característica a optimizar. Este algoritmo fue propuesto por Ramamritham et al. [29].

Entrada: Un grafo de flujo de trabajo $w = (\mathcal{V}, \mathcal{E})$
Salida: Una planificación f

1: **while** $\exists v \in \mathcal{V}$ no completada **do**
2: $t \leftarrow$ Obtener una tarea lista, no calendarizada, con padres calendarizados;
3: $r \leftarrow$ Obtener un recurso que pueda empezar la tarea en el menor tiempo;
4: Calendarizar t en r, i.e., $f(r) = t$;
5: **end while**

A.1.2. Definiciones para los algoritmos Max-Min y Min-min

Los algoritmos min-min y max-min utilizan las siguientes estimaciones:

- $EET(t, r)$ – **Tiempo estimado de ejecución:** Tiempo que el recurso (servicio) r tomará en ejecutar la tarea t, desde que la tarea es ejecutada en el recurso
- $EAT(t, r)$ – **Tiempo estimado de disponibilidad:** Tiempo en el que el recurso r estará disponible para ejecutar la tarea t

- $FAT(t, r)$ – **Tiempo de archivo disponible:** Tiempo más pronto en que todos los archivos requeridos por la tarea t están disponibles en el recurso r
- $ECT(t, r)$ – **Tiempo estimado de terminación:** Tiempo estimado en cual la tarea t terminará su ejecución en el recurso r:

$$ECT(t, r) = EET(t, r) + max(EAT(t, r), FAT(t, r))$$

- $MCT(t)$ – **Tiempo mínimo estimado de terminación:** ECT mínimo para la tarea t sobre todos los recursos disponibles, es decir:

$$MCT(t) = \min_{r \in \mathcal{S}} ECT(t, r)$$

A.1.3. Min-Min

El algoritmo min-min está basado en la heurística de terminar las tareas más cortas en el menor tiempo posible. Para ello, hace una estimación del tiempo de ejecución tomando en cuenta el tiempo de preparación de las tareas en los servicios –o recursos– para tener el tiempo y los archivos necesarios disponibles para el recurso en cuestión. El algoritmo fue propuesto por Maheswaran et al. [24].

Entrada: Un grafo de flujo de trabajo $w = (\mathcal{V}, \mathcal{E})$
Salida: Una planificación f

```
1: while ∃v ∈ V no completada do
2:     t ← Obtener conjunto de tareas listas, no calendarizadas, con padres calendarizados;
3:     PLANIFICACION(tasks);
4: end while
5: procedure PLANIFICACION(availTasks)
6:     while ∃t ∈ availTasks no planificadas do
7:         for t ∈ availTasks do
8:             res ← Obtener recursos disponibles para t;
9:             for r ∈ res do
10:                Calcular ECT(t, r);
11:            end for
12:            R_T ← arg mín_{r∈res} ECT(t, r);
13:        end for
14:        T ← arg mín_{t∈availTasks} ECT(t, R_T);
15:        Planificar T en R_T;
16:        Remover T de availTasks;
17:        Actualizar EAT(R_T);
18:    end while
19: end procedure
```

A.1.4. Max-Min

El algoritmo max-min –también propuesto Maheswaran et al. [24]– es muy similar al algoritmo min-min. La diferencia radica en que éste calendariza tareas cuyo tiempo mínimo de ejecución es el mayor, de tal modo que se ejecutan las tares más *largas*.

El único cambio que se necesita hacer al algoritmo A.1.3 es cambiar la línea 14:

$$T \leftarrow \arg \min_{t \in availTasks} ECT(t, r);$$

por

$$T \leftarrow \arg \max_{t \in availTasks} ECT(t, r);$$

Así, el algoritmo modificado puede verse a continuación:

Entrada: Un grafo de flujo de trabajo $w = (\mathcal{V}, \mathcal{E})$
Salida: Una planificación f

```
 1: while ∃v ∈ V no completada do
 2:     t ← Obtener conjunto de tareas listas, no planificadas, con padres planificados;
 3:     PLANIFICAR(t);
 4: end while
 5: procedure PLANIFICAR(tareas)
 6:     while ∃t ∈ tareas no planificadas do
 7:         for t ∈ tareas do
 8:             res ← Obtener recursos disponibles para t;
 9:             for r ∈ res do
10:                 Calcular ECT(t, r);
11:             end for
12:             R_T ← arg mín_{r∈res} ECT(t, r);
13:         end for
14:         T ← arg máx_{t∈availTasks} ECT(t, R_T);
15:         Planificar T en R_T;
16:         Remover T de tareas;
17:         Actualizar EAT(R_T);
18:     end while
19: end procedure
```

A.1.5. Sufragio

El algoritmo Sufragio[1] [24] es una variación del algoritmo Min-min, el cual considera el valor del sufragio para hacer la planificación. Dicho valor es la diferencia entre el menor tiempo de ejecución para una tarea t sobre un conjunto de recursos disponibles y el segundo menor. Se calendariza a la tarea que tenga el valor del sufragio más alto, por el hecho de que las tareas que son muy sensibles a los cambios de los recursos deben ser calendarizadas primero.

[1] También conocido como *Sufferage*

Entrada: Un grafo de flujo de trabajo $w = (\mathcal{V}, \mathcal{E})$
Salida: Una planificación f

while $\exists v \in \mathcal{V}$ *no completada* **do**
  $t \leftarrow$ Obtener conjunto de tareas listas, no planificadas, con padres planificados;
  PLANIFICAR(t);
end while
procedure PLANIFICAR($tareas$)
  while $\exists t \in tareas$ no planificadas **do**
    for $t \in tareas$ **do**
      $res \leftarrow$ Obtener recursos disponibles para t;
      for $r \in res$ **do**
        Calcular $ECT(t, r)$;
      end for
      $R_t^1 \leftarrow \arg\min_{r \in res} ECT(t, r)$;
      $R_t^2 \leftarrow \arg\min_{r \in res, r \neq R_t^1} ECT(t, r)$;
      $suf_t \leftarrow ECT(t, R_t^2) - ECT(t, R_t^1)$;
    end for
    $T \leftarrow \arg\max_{t \in availTasks} suf_t$;
    Planificar T en R_T^1;
    Remover T de $tareas$;
    Actualizar $EAT(R_T)$;
  end while
end procedure

A.1.6. HEFT

El algoritmo HEFT[2] fue propuesto por Topcuoglu et al. [34]. Este algoritmo está dividido en dos fases: (1) priorización de tareas y (2) selección de recursos. En la primera fase, para todas las tareas se calcula un valor llamado $Rango_{Asc}$, que es una estimación del costo de ejecutar una tarea. En la segunda fase, se asigna a cada tarea de la lista un recurso disponible que tenga el tiempo de disponibilidad más corto. Este tiempo se refiere al instante en que un recurso tiene todos los datos necesarios para ejecutar la tarea a planificar.

El algoritmo HEFT utiliza las fórmulas que vamos a describir a continuación:

Sea $time(T_i, r)$ el tiempo de ejecución de la tarea T_i en el recurso r y sea R_i el conjunto de recursos disponibles para ejecutar la tarea T_i. El tiempo de ejecución promedio de la tarea T_i está definido como:

$$\overline{\omega_i} = \frac{\sum_{r \in R_i} time(T_i, r)}{|R_i|} \tag{A.1}$$

Sea $time(e_{ij}, r_i, r_j)$ el tiempo de transferencia de datos entre los recursos r_i y r_j que utilizan el canal e_{ij} para transferir los datos. Sea R_i y R_j los conjuntos de recursos

[2]Heterogeneous Earliest-Finish Time

disponibles para ejecutar las tareas T_i y T_j, respectivamente. El tiempo promedio de transmisión desde T_i hasta T_j está definido por la siguiente ecuación:

$$\overline{c_{ij}} = \frac{\sum_{r_i \in R_i, r_j \in R_j} time(e_{ij}, r_i, r_j)}{|R_i||R_j|} \tag{A.2}$$

Cabe recordar que en la primera fase del algoritmo HEFT, las tareas son ordenads de acuerdo a una función de rango. Esta función está definida por partes. Para una tarea de salida T, es decir, una tarea que no tiene sucesores que dependan de ella, el valor de rango es determinado por:

$$Rango(T) = \overline{\omega_i} \tag{A.3}$$

Para las tareas que no son tareas de salida, el valor de rango se calcula con la siguiente expresión:

$$Rango(T) = \overline{\omega_i} + \max_{T_j \in succ(T_i)} (\overline{C_{ij}} + Rango(T_j)) \tag{A.4}$$

donde $succ(T_i)$ es el conjunto de los sucesores inmediatos de la tarea T_i.

Entrada: Un grafo de flujo de trabajo $w = (\mathcal{V}, \mathcal{E})$
Salida: Una planificación f

1: Calcular *Tiempo de Ejecución Promedio* A.1 para cada tarea $v \in \mathcal{V}$;
2: Calcular *Tiempo de Transferencia de Datos Promedio* A.2 entre tareas y sus sucesores;
3: Calcular $Rango_{Asc}$ para cada tarea, de acuerdo a A.3 y A.4;
4: Ordenar las tareas por $Rango_{Asc}$, en orden decreciente en una lista Q;
5: **while** $Q \neq \varnothing$ **do**
6: $t \leftarrow$ Remover la primera tarea de Q;
7: $r \leftarrow \arg\min_{r_i \in R} EFT(r_i, t)$ usando *planificación basada en inserciones*
8: Planificar t en r;
9: **end while**

A.1.7. Híbrido

Este algoritmo combina dos heurístas: primero agrupa las tareas que tengan dependencias entre sí, de tal modo que haya un balance entre el número de grupos y el número de recursos disponibles para la ejecución de las tareas agrupadas. Luego, con los grupos listos, se utiliza un algoritmo de planificación batch para asignar a cada grupo un recurso. Este algoritmo/heurística fue propuesto por Sakellariou et al [30]. El algoritmo híbrido utiliza las definiciones de tiempos promedio y función de rango del algoritmo HEFT.

Entrada: Un grafo de flujo de trabajo $w = (\mathcal{V}, \mathcal{E})$
Salida: Una planificación f

1: Calcular *Peso* de cada tarea y arista, de acuerdo a A.1 y A.2;

2: Calcular *Rango* para cada tarea, de acuerdo a A.3 y A.4;
3: Ordenar las tareas por *Rango*, en orden decreciente en una lista Q;
4: $i \leftarrow 0$
5: Crear un grupo G_i;
6: **while** $Q \neq \varnothing$ **do**
7: $t \leftarrow$ Remover la primera tarea de Q;
8: **if** t tiene una dependencia con una tarea en G_i **then**
9: $i \leftarrow i + 1$;
10: Crear un grupo G_i;
11: **end if**
12: Agregar t a G_i;
13: **end while**
14: $j \leftarrow 0$;
15: **while** $j \leq i$ **do**
16: Planificar tareas en G_i usando un algoritmo *batch*;
17: $j \leftarrow j + 1$;
18: **end while**

A.1.8. TANH

Bajaj et al. [8] proponen un algoritmo basado en dos heurísticas: en la duplicación de tareas y en el agrupamiento de tareas. La primera heurística se basa en que si dos tareas dependen del resultado de una tarea en común y, estas dos tareas se encuentran en recursos diferentes y con un tiempo muerto previo a la ejecución de estas tareas, entonces, se ejecuta de forma rendundante la tarea común en los dos recursos diferentes, para que las dos tareas puedan continuar su ejecución de manera independiente y no exista un costo de comunicación de datos entre estos dos recursos. La segunda heurística se refiere a que las tareas con dependencias entre sí se deben ejecutar en los mismos recursos, para que los costos de comunicación de datos sean despreciables.

Entrada: Un grafo de flujo de trabajo $w = (\mathcal{V}, \mathcal{E})$
Salida: Una planificación f
1: Calcular *Parámetros* para cada nodo tarea;
2: Agrupar tareas del flujo de trabajo;
3: **if** Número de clusters $\geq$ Número de recursos disponibles **then**
4: Reducir el Número de clusters al Número de recursos disponibles;
5: **else**
6: Ejecutar duplicación de tareas;
7: **end if**

A.1.9. GRASP

El procedimiento de búsqueda aleatoria voraz adaptativa (GRASP por sus siglas en inglés) es una heurística que se utiliza para resolver problemas de optimización

combinatoria. De manera voraz, se construyen planificaciones que reduzcan del tiempo de ejecución total evaluando cada uno de los sucesores de una tarea dada. Así, el algoritmo escoge las tareas de manera aleatoria cuya variación en el tiempo no pase de un rango dado. El algoritmo termina hasta que se cumple un criterio de terminación, como un tiempo límite. Este algoritmo fue propuesto por Blythe et al. [9].

Entrada: Un grafo de flujo de trabajo $w = (\mathcal{V}, \mathcal{E})$
Salida: Una planificación f

```
1: while Criterio de terminación no satisfactorio do
2:     plan ← crearPlanificacion(w);
3:     if plan es mejor que mejorPlanificacion then
4:         mejorPlanificacion ← plan;
5:     end if
6: end while
7: procedure crearPlanificacion(flujo)
8:     solucion ← construirSolucion(flujo);
9:     nSolucion ← busquedaLocal(solucion);
10:    if nSolucion es mejor que solucion then
11:        return nSolucion;
12:    end if
13:    return solucion;
14: end procedure
15: procedure construirSolucion(flujo)
16:    while planificación no completada do
17:        T ← Obtener tareas listas sin asignar;
18:        Crear RCL para cada t ∈ T;
19:        subSolucion ← Seleccionar recurso aleatoriamente para cada t ∈ T de su
   RCL;
20:        solucion ← solucion ∪ subSolucion;
21:        Actualizar información para futuros RCL;
22:    end while
23:    return solucion;
24: end procedure
25: procedure busquedaLocal(solucion)
26:    nSolucion ← Encontar una solución local óptima;
27:    return nSolucion;
28: end procedure
```

Bibliografía

[1] Askalon programming environment for cloud and grid computing. http://www.dps.uibk.ac.at/projects/askalon/. Consultado el 23 de junio de 2014.

[2] Cluster Híbrido de Supercómputo — Xiuhcoatl — Cinvestav. http://clusterhibrido.cinvestav.mx/. Consultado el 10 de noviembre de 2013.

[3] HTCondor - Home. http://research.cs.wisc.edu/htcondor/. Consultado el 22 de junio de 2014.

[4] Laboratorio de Supercómputo y Visualización en Paralelo. http://supercomputo.izt.uam.mx/infraestructura/aitzaloa.php. Consultado el 10 de noviembre de 2013.

[5] Manjrasoft - products. http://www.manjrasoft.com/products.html. Consultado el 23 de junio de 2014.

[6] Pegasus | Publications. https://pegasus.isi.edu/publications/. Consultado el 22 de junio de 2014.

[7] Sebastian Alcerreca Alcocer. Emisión de Comprobantes Fiscales Digitales (CFD) - Desarrollo del sistema de emisión, distribución y almacenamiento de CFD de la corporación Montecito. Tesis de licenciatura, Instituto Tecnológico Autónomo de México, 2013.

[8] 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.

[9] 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.

[10] Grupo BMV. Informe anual 2012. Technical report, Bolsa Mexicana de Valores, 2012.

[11] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation computer systems*, 25(6):599–616, 2009.

[12] Xingchen Chu, Krishna Nadiminti, Chao Jin, Srikumar Venugopal, and Rajkumar Buyya. Aneka: Next-generation enterprise grid platform for e-science and e-business applications. In *e-Science and Grid Computing, IEEE International Conference on*, pages 151–159. IEEE, 2007.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[14] Ewa Deelman, Scott Callaghan, Edward Field, Hunter Francoeur, Robert Graves, Nitin Gupta, Vipin Gupta, Thomas H Jordan, Carl Kesselman, Philip Maechling, et al. Managing large-scale workflow execution from resource provisioning to provenance tracking: The cybershake example. In *e-Science and Grid Computing, 2006. e-Science'06. Second IEEE International Conference on*, pages 14–14. IEEE, 2006.

[15] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[16] Thomas Fahringer, Radu Prodan, Rubing Duan, Francesco Nerieri, Stefan Podlipnig, Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, Alex Villazon, and Marek Wieczorek. Askalon: A grid application development and computing environment. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 122–131. IEEE Computer Society, 2005.

[17] J Octavio Gutierrez-Garcia and Kwang Mong Sim. Agent-based cloud workflow execution. *Integrated Computer-Aided Engineering*, 19(1):39–56, 2012.

[18] Gopi Kandaswamy, Anirban Mandal, and Daniel A Reed. Fault tolerance and recovery of scientific workflows on computational grids. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 777–782. IEEE, 2008.

[19] Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to algorithms*. The MIT press, 2001.

[20] Ke Liu, Hai Jin, Jinjun Chen, Xiao Liu, Dong Yuan, and Yun Yang. A compromised-time-cost scheduling algorithm in swindew-c for instance-intensive cost-constrained workflows on a cloud computing platform. *International Journal of High Performance Computing Applications*, 24(4):445–456, 2010.

[21] 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.

[22] Xiao Liu, Dong Yuan, Gaofeng Zhang, Jinjun Chen, and Yun Yang. Swindew-c: a peer-to-peer based cloud workflow system. In *Handbook of Cloud Computing*, pages 309–332. Springer, 2010.

[23] Ewing Lusk, S Huss, B Saphir, and M Snir. Mpi: A message-passing interface standard, 2009.

[24] 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.

[25] 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.

[26] Daniel A Menasce and Emiliano Casalicchio. A framework for resource allocation in grid computing. In *MASCOTS*, pages 259–267, 2004.

[27] Francesco Nerieri, Mumtaz Siddiqui, Juergen Hofer, Alex Villazon, Radu Prodan, and Thomas Fahringer. Using a heterogeneous service-oriented grid infrastructure for movie rendering. In *1st Austrian Grid Symposium, Schloss Hagenberg, Austria*. Citeseer, 2005.

[28] Angela O'Brien, Steven Newhouse, and John Darlington. Mapping of scientific workflow within the e-protein project to distributed resources. In *UK e-Science All Hands Meeting*, pages 404–409, 2004.

[29] 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.

[30] 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.

[31] 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.

[32] Jamie Shiers. The worldwide lhc computing grid (worldwide lcg). *Computer physics communications*, 177(1):219–223, 2007.

[33] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[34] 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.

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

[36] 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.

[37] Christian Vecchiola, Xingchen Chu, and Rajkumar Buyya. Aneka: a software platform for .net-based cloud computing. *High Speed and Large Scale Scientific Computing*, pages 267–295, 2009.

[38] 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.

[39] 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.

[40] Yun Yang, Ke Liu, Jinjun Chen, Joel Lignier, and Hai Jin. Peer-to-peer based grid workflow runtime environment of swindew-g. In *e-Science and Grid Computing, IEEE International Conference on*, pages 51–58. IEEE, 2007.

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

[42] 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.

[43] 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.