Universidad Nacional de Rosario
Facultad de Ciencias Exactas, Ingeniera y Agrimensura

Tesina de Grado
Licenciatura en Ciencias de la Computación

Partición de Modelos de Gran Escala para Simulación en Paralelo por Eventos Discretos

Franco Sansone
Legajo: S-4912/3

Director: Dr. Joaquín Fernández
Co-Director: Dr. Ernesto Kofman

Marzo, 2025

A Mamá, dondequiera que esté.

Resumen

Los métodos de Sistemas de Estados Cuantificados (QSS por sus siglas en inglés) permiten aproximar un sistema continuo por uno de eventos discretos para su simulación. Estos métodos y su implementación en paralelo permiten reducir en muchos casos los costos computacionales en la simulación de sistemas a gran escala. Los algoritmos de simulación de eventos discretos en paralelo requieren que el modelo original sea dividido en submodelos. Además, para que la paralelización sea efectiva, la división del modelo debe ser balanceada, y la comunicación entre los submodelos debe ser minimizada. Este problema puede tratarse como un problema de particionamiento de grafos, donde cada nodo representa unidades de cálculo del modelo, y las aristas comunicación entre las mismas. La obtención de dicha partición presenta un problema en sí mismo, ya que la representación de grandes modelos mediante grafos implica un alto uso de memoria y a su vez hay un alto costo computacional asociado al refinamiento de las particiones para lograr un balance de carga que minimice la comunicación.

En este trabajo presentamos una metodología novedosa para resolver el problema mencionado de balance de carga cuya principal aplicación es la simulación en paralelo de grandes modelos con métodos de QSS. La novedad principal de la metodología es que utiliza una estructura de datos compacta brindada por el uso de *grafos basados en conjuntos* (SBG por sus siglas en inglés) que permite reducir notablemente los costos computacionales. La metodología consta de algoritmos para representar modelos particionados mediante grafos basados en conjuntos y algoritmos que, tras construir una partición inicial, refinan iterativamente la misma hasta encontrar un particionado sub-óptimo.

Además de describir la metodología desarrollada, el trabajo reporta casos de aplicación donde se mide la calidad del particionado y los costos computacionales asociados utilizando la misma y comparando dichos resultados con los que se obtienen mediante métodos clásicos de balance de carga.

Agradecimientos

A mis directores, Joaquín y Ernesto, por haber confiado en mí para realizar este trabajo, y por haber logrado que el mismo no sea una mera obligación, sino un camino que disfruté transitar. A los evaluadores, Taihú y Gustavo, y a quienes colaboraron en todo este proceso.

Al instituto CIFASIS por haberme brindado un espacio para realizar este trabajo.

A Bianca, mi compañera. A Maca, Fede y Nahuel.

A mi familia de sangre, política y ensamblada.

A los amigos que he hecho durante el recorrido que me trajo hasta aquí.

A los miembros, tanto alumnos como docentes, de esta carrera, quienes no han construido sólo una carrera universitaria, sino una comunidad.

A los apasionados y apasionadas que, de una forma u otra, me he cruzado en el camino y me han transmitido eso, pasión.

> —*El mundo es eso* — reveló—. *Un montón de gente, un mar de fueguitos.*
> Eduardo Galeano, el mundo.

Índice general

Capítulo 1

Introducción

1.1. Motivación y Objetivo

El modelado y la simulación de sistemas continuos está en el centro de la investigación científica y de diferentes áreas de la ingeniería. Profesionales desarrollan modelos más grandes y complejos cada año, y su simulación representa un desafío debido al alto costo computacional. Con la llegada de los procesadores multinúcleo y de los grupos de computadoras de múltiples nodos, la simulación en paralelo de sistemas de tiempo continuo se convirtió en una forma habitual de mitigar los tiempos de ejecución de estas simulaciones.

Los modelos de tiempo continuo usualmente son expresados como un conjunto de Ecuaciones Diferenciales Ordinarias (ODE), o transformadas en un conjunto de las mismas, y se deben aplicar algortimos de integración númerica para resolverlas. La mayoría de estos algoritmos están basados en la discretización del tiempo. Estos algoritmos producen un alto costo computacional. En modelos grandes (millones de variables de estado), la evaluación de las funciones del modelo requieren millones de cálculos.

Un nuevo método fue desarrollado, método de integración de sistemas de eventos cuantificados (QSS), que reemplaza la discretización del tiempo de los algoritmos clásicos por la discretización de variables de estado. Estos métodos de integración cuentan con ciertas características que reducen el costo computacional en sistemas de gran escala.

En [4] y [7] se presentaron implementaciones la simulación en paralelo de métodos QSS para arquitecturas multinúcleo. Un paso fundamental para la correcta simulación en paralelo, es una apropiada de división en submodelos. Esta división debe ser balanceada, y la comunicación entre submodelos debe ser minimizada. Este problema puede presentarse como un problema de particionamiento de grafos, donde los nodos representan unidades de cálculo y las aristas representan comunicación entre las mismas. La comunicación existe cuando una variable es definida en una ecuación y usada en otra. Ante la existencia de un cambio en el valor de una variable, las demás ecuaciones deben ser informadas. Este grafo es conocido como *Grafo Computacional*.

Ya mencionamos que modelos a gran escala pueden contener millones de variables de estado. Su representación en un grafo tradicional implica millones de nodos y de aristas. Es inherente el alto costo computacional así como el alto

uso de memoria para la representación del modelo y sus conexiones. En este trabajo presentamos un nuevo método de representación del grafo computacional usando Grafos Basados en Conjuntos [12], [17]. Esta estructura nos permite representar nodos y aristas en forma compacta, usando conjuntos de intervalos en lugar de conjuntos por extensión. Luego, presentamos dos métodos para obtener particiones iniciales, y un método de refinamiento de las mismas basado en [11].

1.2. Organización de la Tesina

Los capítulos de este trabajo se organizan de la siguiente manera: el Capítulo 2 aporta un recorrido por los distintos métodos de simulación. El Capítulo 3 introduce conceptos teóricos de la simulación en paralelo. En el Capítulo 4 se introduce el concepto de balance de carga para simulación en paralelo como un problema de particionamiento de grafos, y se presenta el algoritmo de particionado [11], la base de nuestro método de refinamiento. El Capítulo 5 presenta la definición de Grafos Basados en Conjuntos y el Capítulo 6 una serie de algoritmos para representar un grafo computacional usando dicha estructura de datos. En el Capítulo 7 se mencionan detalles de la implementación de los algoritmos y en el Capítulo 8 resultados sobre distintos modelos, comparando con otros métodos de particionamiento. Finalmente, en el Capítulo 9, se muestran las conclusiones de este trabajo y propuestas de trabajo a futuro.

Capítulo 2

Simulación de Modelos Matemáticos

Un modelo es una representación simplificada de un sistema. En el contexto de este trabajo trataremos con modelos matemáticos, que pueden definirse como sigue: *un modelo matemático es una contrucción matemática abstracta y simplicada relacionada con una parte de la realidad y creada para un propósito en particular* [2]. Estos modelos utilizan expresiones matemáticas (tales como ecuaciones, inecuaciones, ecuaciones diferenciales) y sobre los mismos pueden realizarse simulaciones, a modo de experimentación, con el objetivo de estudiar su comportamiento. La simulación de un modelo consiste en la ejecución del mismo a través del tiempo.

2.1. Simulación de Modelos de Tiempo Discreto

Los modelos de tiempo discreto son usualmente los más intuitivos para comprender. Como se ilustra en la Figura 2.1, este formalismo asume un modo de ejecución paso a paso. En un instante particular de tiempo el modelo está en un estado particular y define cómo el estado del mismo cambia al instante de tiempo siguiente. El estado siguiente usualmente depende del estado actual y de las influencias del entorno.

Los sistemas de tiempo discreto tienen diversas aplicaciones. Las más populares son en sistemas digitales donde el reloj define los pasos de tiempo discretos. También suelen ser usados como aproximaciones de sistemas continuos. En estos casos se elige una unidad de tiempo, por ejemplo un segundo, un minuto o un año, para definir un reloj artificial y el sistema se representa los cambios de estado de un instante de "observación" al siguiente. Por lo tanto, para definir un modelo de tiempo discreto tenemos que definir cómo el estado actual y las entradas provenientes del entorno determinan el estado siguiente del mismo.

La manera más simple de definir los cambios de estado de un modelo es por medio de una tabla, como la Tabla 2.1, en la cual anotamos todas las combinaciones de estados y entradas y el estado siguiente así como la salida para cada combinación. En la Tabla 2.1 sólo tenemos dos estados posibles (0 y 1) y dos salidas (0 y 1). La primer fila indica que si el estado actual es 0 y la entrada es 0, el estado siguiente y salida serán 0.

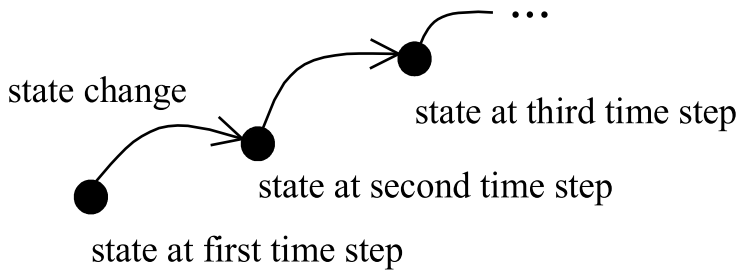


Figura 2.1: Ejecución paso a paso de sistemas de tiempo discreto Imagen extraída de [15].

Estado actual	Entrada actual	Estado siguiente	Salida actual
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	1

Tabla 2.1: Tabla de transición/salida para un sistema de retardo.

La tabla de estados recién descripta puede ser interpretada como la especificación de los cambios de estado a lo largo del tiempo:

Si el estado del modelo en el instante de tiempo t es q y la entrada para el instante de tiempo t es x entonces el estado para el instante $t+1$ es $\delta(q,x)$ y la salida y para el instante t es $\lambda(q,x)$.

Aquí δ es conocida como la función de transición de estado y λ como la función de salida. Una secuencia de estados, $q(0)$, $q(1)$, $q(2)$, ... es llamada trayectoria de estado. Dado un estado inicial arbitrario $q(0)$, los estados subsecuentes se pueden determinar por:

$$q(t+1) = \delta(q(t), x(t))$$

.

Así también, la trayectoria de salida es dada por:

$$y(t) = \lambda(q(t), x(t))$$

.

Podemos escribir un simple algoritmo para computar el las trayectorias de estado y de salida de los un modelo de tiempo discreto definiendo su trayectoria de entrada y estado inicial siguiendo la Tabla 2.2.

Algorithm 1 Computar Trayectorias de Estado y de Salida

$T_i = 0, \ldots, T_f = 9$
$x(0) = 1, \ldots, x(9) = 0$ ▷ la trayectoria de entrada
$x(0) = 0$ ▷ el estado inicial
$t = T_i$
while $t \leq T_f$ **do**
  $y(t) = \lambda(q(t), x(t))$
  $q(t+1) = \delta(q(t), x(t))$
  $t+ = 1$
end while

La ejecución del algoritmo, por ende la simulación de nuestro modelo, produciría trayectorias de estado y de salida como las descriptas en la tabla 2.2.

tiempo	0	1	2	3	4	5	6	7	8	9
trayectoria de entrada	1	0	1	0	1	0	1	0	1	0
trayectoria de estado	0	1	0	1	0	1	0	1	0	1
trayectoria de salida	1	0	1	0	1	0	1	0	1	0

Tabla 2.2: Trayectoria de estado y de salida.

2.2. Simulación de Modelos de Tiempo Continuo

En modelado de tiempo discreto teníamos una función de transición de estado que nos daba la información del estado al instante de tiempo siguiente usando el estado y la entrada actuales. En el enfoque clásico de modelado de ecuaciones diferenciales, La relación de transición de estado es bastante diferente. Para modelos de ecuaciones diferenciales el estado siguiente no es especificado en forma directa sino que se usa una función derivada para especificar la frecuencia de cambio de las variables de estado. En cualquier instante en el eje de tiempo, dado un estado y un valor de entrada, sólo sabemos la frecuencia de cambio del estado. Con esta información, en cualquier momento del futuro el estado debe ser computado.

Los sistemas de tiempo continuo usualmente son expresados usando varias variables de estado. Las derivadas son entonces funciones de algunas o todas las variables de estado. Sean z_1, z_2, …, z_n las variables de estado y u_1, u_2, …, u_n las variables de entrada, un modelos de tiempo continuo está compuesto por un conjunto de ecuaciones diferenciales de primer orden (ODEs):

$$\begin{aligned}
\dot{z_1}(t) &= f_1(z_1(t), \cdots, z_n(t), u_1(t), \ldots, u_m(t)) \\
\dot{z_2}(t) &= f_2(z_1(t), \cdots, z_n(t), u_1(t), \cdots, u_m(t)) \\
&\vdots \\
\dot{z_n}(t) &= f_n(z_1(t), \cdots, z_n(t), u_1(t), \ldots, u_m(t))
\end{aligned}$$

donde $\dot{z_1}$ representa $\frac{\mathrm{d}z_i}{\mathrm{d}t}$. Para una notación más compacta, las variables

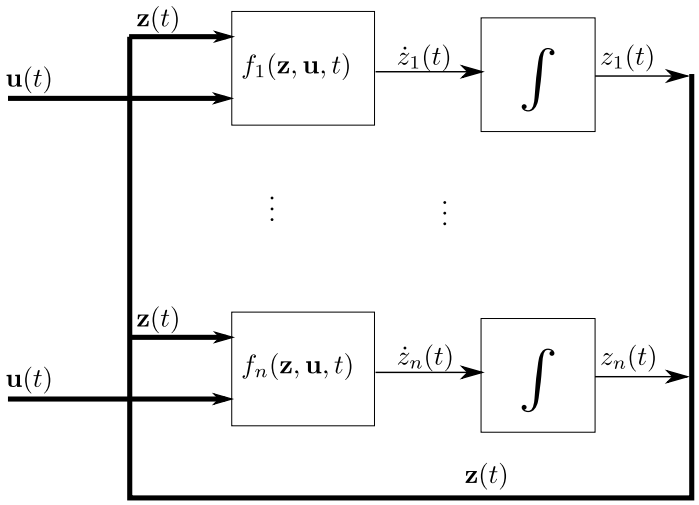


Figura 2.2: Representación en diagrama de bloques de la ecuación 2.1 - Imagen extraída de [16].

pueden agruparse en vectores, $\mathbf{z}(t) \triangleq [z_1, z_2, \ldots, z_n]^\intercal$ $\mathbf{u}(t) \triangleq [u_1, u_2, \ldots, u_n]$, entonces las ecuaciones son escritas así:

$$\begin{aligned} \dot{z_1}(t) &= f_1(\mathbf{z}(t), \mathbf{u}(t)) \\ \dot{z_2}(t) &= f_2(\mathbf{z}(t), \mathbf{u}(t)) \\ &\vdots \\ \dot{z_n}(t) &= f_n(\mathbf{z}(t), \mathbf{u}(t)). \end{aligned} \tag{2.1}$$

Observe que las derivadas de las variables de estado z_i son computadas respectivamente por funciones f_i que toman los vectores de estado y de entrada como argumentos. Esto se puede ver en el Diagrama de bloques en Figura 2.2. Los vectores de entrada y de estado son entradas de la función de frecuencia de cambio f_i. Estos proveen como salida las derivadas $\dot{z_i}$ de las variables de estado z_i que son enviadas al bloque integrador. Las salidas del bloque integrador son las variables de estado z_i.

Funciones f_i en la Ecuación 2.1 pueden ser agrupadas en un vector también $\mathbf{f} \triangleq [f1, \cdots, fn]$, obteniendo así la clásica ecuación de estados de ODEs:

$$\dot{\mathbf{z}}(t) = \mathbf{f}(\mathbf{z}(t), \mathbf{u}(t)). \tag{2.2}$$

En la mayoría de los casos la solución exacta no se puede obtener por lo tanto nos vemos forzados a buscar aproximaciones numéricas para los valores de estado en ciertos instantes de tiempo t_0, t_1, ..., t_n. Estas aproximaciones se obtienen usando algoritmos de integración numérica. A continuación mencionaremos algunos.

2.2.1. Integración de Euler

El método más simple para resolver la Ecuación 2.2 fue propuesta por Leonhard Euler en 1768. Se basa en la aproximación de la derivada de esta-

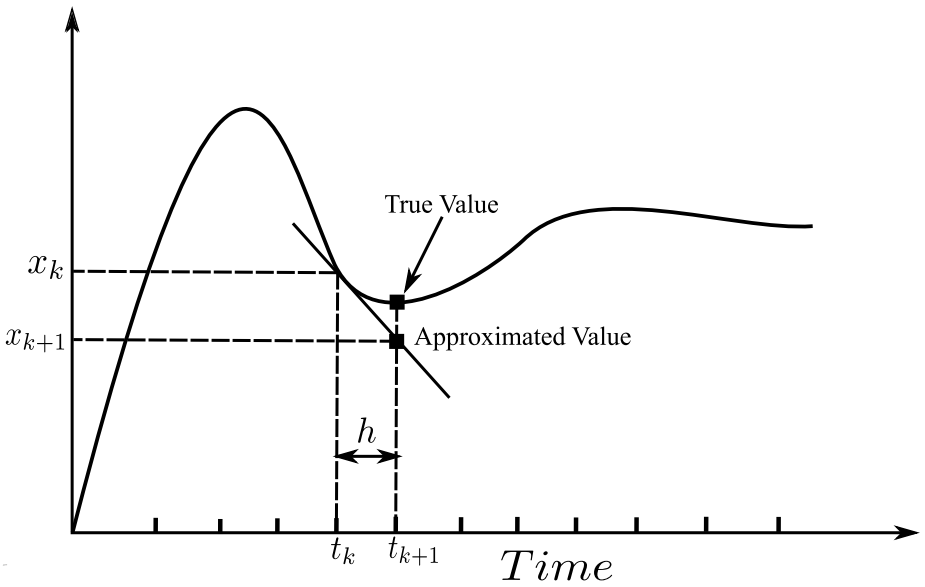


Figura 2.3: Ilustración de principio de discretización de tiempo en el método Forward Euler.

do de la siguiente manera:

$$\dot{\mathbf{z}}(t) \approx \frac{\mathbf{z}(t+h)-\mathbf{z}(t)}{h} \frac{\mathbf{z}(t+h)-\mathbf{z}(t)}{h}$$
$$\implies \mathbf{f}(\mathbf{z}(t),\mathbf{u}(t)) \approx \frac{\mathbf{z}(t+h)-\mathbf{z}(t)}{h}$$

entonces obtenemos

$$\mathbf{z}(t+h) \approx \mathbf{z}(t) + h \cdot \mathbf{f}(\mathbf{z}(t),\mathbf{u}(t))$$

donde h es el tamaño del paso.

Forward Euler Definiendo $t_k = t_0 + k \cdot h$ for $k = 1, \ldots, N$, se define el método Forward Euler (FE) por la siguiente fórmula

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_k) + h \cdot \mathbf{f}(\mathbf{z}(t_k),\mathbf{u}(t_k)) \tag{2.3}$$

La simulación utilizando el método de FE se torna trivial ya que el método de integración utiliza solo valores pasados de las variables de estado y sus derivadas. Un esquema de integración que exhibe esta característica se denomina *algoritmo de integración explícito*.

En la Figura 2.3 muestra una interpretación gráfica de la aproximación realizada por el método de FE. A tamaño de pasos más grandes, aumenta la eficiencia puesto que se realizan menos cálculos, pero también aumenta el error.

Backward Euler Una variante del método Forward Euler se da por la siguiente fórmula

$$\mathbf{z}(t_{k+1}) = \mathbf{z}(t_k) + h \cdot \mathbf{f}(\mathbf{z}(t_{k+1}), \mathbf{u}(t_{k+1})) \tag{2.4}$$

conocido como método *Backward Euler*.

Como puede verse, el valor desconocido $\mathbf{z}(t_{k+1})$ está ambos lados de la ecuación, es considerado un *método implícito*. Teniendo en cuenta que la función $\mathbf{f}()$ usualmente es no lineal, obtener el siguiente estado requiere resolver una ecuación algebraica no lineal. Este tipo de algoritmos se denominan *métodos de integración implícitos*.

2.2.2. Métodos de integración Monopaso

Se denomina métodos de integración monopaso a aquellos que calculan x_{k+1} utilizando únicamente información sobre x_k. Son usualmente llamadas métodos Runge-Kutta (RK) porque fueron los primeros en proponer estos algoritmos a finales del siglo XIX.

Métodos Runge-Kutta Un método de Runge–Kutta es un algoritmo que avanza la solución desde $x_k(t_k)$ hasta $x_{k+1}(t_k + h)$, usando una formula del tipo

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h \cdot (c_1 \cdot \mathbf{k_1} + \ldots + c_n \cdot \mathbf{k_n})$$

donde las llamadas etapas $k_1 \ldots k_n$ se calculan sucesivamente a apartir de las ecuaciones:

$$\begin{aligned}
&\text{etapa 0:} && \mathbf{k_1} = \mathbf{f}(\mathbf{x_k} + b_{1,1} \cdot h \cdot \mathbf{k_1} + \ldots + b_{1,n} \cdot h \cdot \mathbf{k_n}, t_k + a_1 h) \\
&\vdots && \vdots \\
&\text{etapa } n-1: && \mathbf{k_n} = \mathbf{f}(\mathbf{x_k} + b_{n,1} \cdot h \cdot \mathbf{k_1} + \ldots + b_{n,n} \cdot h \cdot \mathbf{k_n}, t_k + a_n h) \\
&\text{etapa } n: && \mathbf{x_{k+1}} = \mathbf{x_k} + c_1 \cdot h \cdot \mathbf{k_1} + \ldots + c_n \cdot h \cdot \mathbf{k_n}
\end{aligned}$$

El número n de evaluaciones de función en el algoritmo se llama 'número de etapas' y frecuentemente es considerado como una medida del costo computacional de la fórmula considerada.

2.2.3. Métodos de integración Multipaso

En la sección 2.2.2, se mostraron métodos de integración que, de una forma u otra, tratan de aproximar la expansión de Taylor de la solución desconocida en torno al instante de tiempo actual. La idea básica era no calcular las derivadas superiores en forma explícita, sino reemplazarlas por distintas evaluaciones de la función $\mathbf{f}$ en varios puntos diferentes dentro del paso de integración.

Una desventaja de este enfoque es que cada vez que se comienza un nuevo paso, se deja de lado todas las evaluaciones anteriores de la función $\mathbf{f}$ en el paso anterior.

Los métodos de integración multipaso, en lugar de evaluar varias veces la función en un paso para incrementar el orden de la aproximación, tratan de

aprovechar las evaluaciones realizadas en pasos anteriores. En tal sentido, estos métodos utilizan como base polinomios de interpolación o de extrapolación de orden alto.

2.2.4. Sistemas Discontinuos

El sistema que se muestra a continuación representa la dinámica de un balón rebotando contra el suelo:

$$\begin{aligned} \dot{z_1}(t) &= z_2(t) \\ \dot{z_2}(t) &= -g - d(t) \cdot \left(\frac{k}{m} z_1(t) + \frac{b}{m} z_2(t) \right) \end{aligned} \tag{2.5}$$

donde

$$d(t) = \begin{cases} 0 & \text{if } z_1(t) > 0 \\ 1 & \text{otherwise.} \end{cases} \tag{2.6}$$

Este modelo indica que la derivada de la posición z_1 es la velocidad z_2, y la derivada de la velocidad (es decir, la aceleración $\dot{z_2}(t)$) depende del estado discreto $d(t)$. Este estado discreto toma el valor 0 cuando el balón está en el aire ($x_1 > 0$), entonces en ese caso la ecuación es la de un modelo de caída libre. De lo contrario, $d(t)$ toma el valor 1 cuando el balón está en contacto con el piso ($x_1 \leq 0$), correspondiente a un modelo resorte-amortiguador. Establecemos los parámetros del modelo con: $m = 1$, $b = 30$, y $g = 9,81$. También consideremos los estados iniciales $z_1(0) = 2$, $z_2(0) = 0$. Luego simulamos este sistema usando RK4 con tres valores diferentes de tamaño de paso: $h = 0{,}002$, $h = 0{,}001$, $h = 0{,}0005$. Los resultados de la simulación se muestran en la Figura 2.4

A pesar del modelo de integración de alto orden usado y del tamaño de paso pequeño h, los resultados son muy distintos y no se ve una convergencia hacia una solución fehaciente.

Los malos resultados se deben a la presencia de una discontinuidad cuando $z_1(t) = 0$. Los métodos de integración numérica se basan en la continuidad de las variables de estado y de sus derivadas. En presencia de discontinuidades, la expansión en serie no es válida y los algoritmos numéricos ya no aproximan la solución analítica en su orden de precisión.

La solución a este problema requiere el uso de rutinas de *detección de eventos*, que detectan la ocurrencia de discontinuidades. Cuando una discontinuidad es detectada, la simulación avanza hasta la ubicación de la misma y se resetea desde ese punto bajo las nuevas condiciones después que el evento ocurrió. Este proceso es conocido como *manejo de eventos*. De esa manera, el algoritmo nunca integra a través de discontinuidades, solo simula una sucesión de sistemas puramente continuos.

2.3. Simulación de Modelos de Eventos Discretos

Cómo hemos visto hasta este punto, los sistemas continuos se representan generalmente mediante ecuaciones diferenciales ordinarias (ODEs), ecuaciones

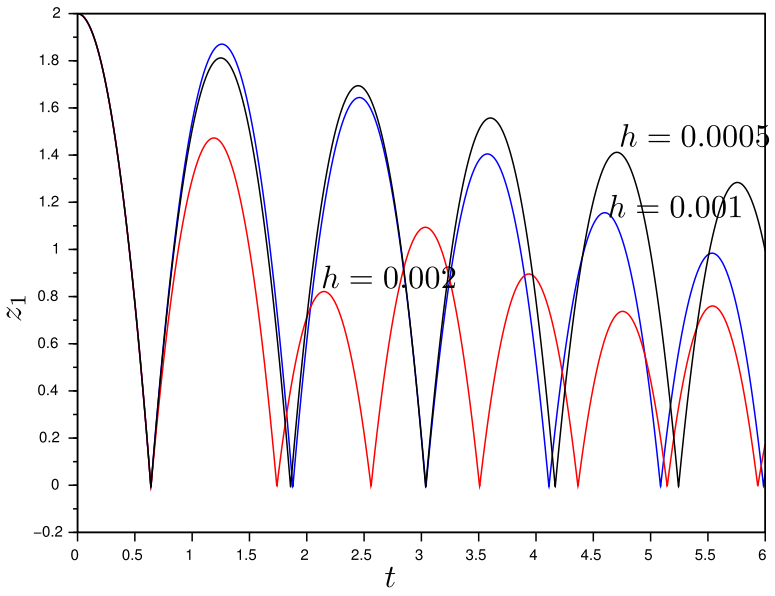


Figura 2.4: Simulación RK4 del modelo del balón rebotando.

en derivadas parciales (PDEs), ecuaciones diferenciales y algebraicas (DAEs) o combinaciones de estas que conforman el modelo del sistema.

Para poder simular estos modelos, los métodos de integración tradicionales, realizan una discretización del tiempo de manera de transformar el modelo de tiempo continuo en un modelo de ecuaciones en diferencias "equivalente". De este modo, la solución del modelo de tiempo discreto en los instantes de muestreo se aproxima a la solución del modelo original en dichos instantes.

Si bien los métodos de integración tradicionales son los más difundidos y han dado respuesta a muchos problemas de simulación, existen también muchos modelos para los cuales la solución brindada por los mismos no ha sido del todo satisfactoria desde el punto de vista de eficiencia. Además, existen modelos de sistemas que, según como se formulen, pueden llegar a ser imposibles de simular a "ciegas", entendiéndose por esto, cargando simplemente el modelo matemático en un simulador sin tener mucha noción del comportamiento del modelo.

A fines de los años noventa se comenzó a desarrollar una metodología alternativa a la clásica discretización temporal para poder aproximar los sistemas continuos. En la misma, en lugar de discretizar el tiempo, se cuantifican las variables de estado manteniendo continua la variable tiempo. De este modo, se verá que en lugar de obtenerse un modelo de tiempo discreto equivalente, se llega a un modelo de eventos discreto equivalente. Y que esta discretización puede representarse fácilmente mediante el formalismo DEVS.

En este capítulo se presentarán los principios de esta metodología y se dará una descripción de los métodos de integración basados en la cuantificación de los estados que se implementaron para este trabajo.

2.3.1. Ejemplo Introductorio de discretización espacial

Un oscilador armónico puede ser representado mediante el modelo de un sistema continuo de segundo orden:

$$\begin{aligned} \dot{z}_{a_1}(t) &= z_{a_2}(t) \\ \dot{z}_{a_2}(t) &= -z_{a_1}(t). \end{aligned} \tag{2.7}$$

Si se saben las condiciones iniciales $z_{a_1}(t_0)$ y $z_{a_2}(t_0)$, dado que el sistema es lineal, es muy fácil encontrar la solución analítica del modelo, siendo ésta $z_{a_i}(t) = c_i \sin(t) + d_i \cos(t)$ con c_i y d_i constantes.

Veamos ahora que ocurre si se modifica el sistema representado en la Ecuación 2.7 de la siguiente manera:

$$\begin{aligned} \dot{z}_1(t) &= \text{floor}[z_2(t)] \triangleq q_2(t) \\ \dot{z}_2(t) &= -\text{floor}[z_1(t)] \triangleq -q_1(t) \end{aligned} \tag{2.8}$$

más cercano a z_i y que es a su vez menor que z_i.

A pesar de que este nuevo sistema es no lineal y discontinuo, se lo puede simular fácilmente. Tomemos como condiciones iniciales $z_1(0) = 4{,}5$ y $z_2(0) = 0{,}5$.

Con estos valores iniciales se tiene que $q_1(0) = 4$ y $q_2(0) = 0$ y se mantienen en esos valores hasta que $z_1(t)$ o $z_2(t)$ cruce a través de alguno de los valores enteros más próximos a ella. En consecuencia, $\dot{x}_1(0) = 0$ y $\dot{x}_2(0) = -4$ por lo que x_1 se mantiene constante y x_2 decrece con pendiente -4.

Luego de $0{,}5/4 = 0{,}125$ segundos (instante $t_1 = 0{,}125$), x_2 cruza por 0 y q_2 toma el valor -1. Entonces, cambia la pendiente de x_1 tomando el valor $\dot{x}_1(t_1^+) = -1$

Luego, z_2 cruza por -1 en el instante $t_2 = t_1 + 1/4$, y q_2 toma el valor -2. En ese instante, $x_1(t_2) = 4{,}5 - 1/4 = 4{,}25$ y $\dot{z}_1(t_2^+) = -2$.

El próximo cambio ocurre cuando z_1 cruza por 4 en el instante de tiempo $t_3 = t_2 + 0{,}25/2$. Luego, $q_1(t_3^+) = 3$ y la pendiente de z_2 pasa a ser -3. Continuando con el análisis de la misma manera, se obtienen los resultados de la *simulación* mostrados en la Figura 2.6. Los resultados son muy similares a la solución analítica del sistema original en la Ecuación 2.7.

Cuando se reemplaza z_i por $q_i = \text{floor}(z_i)$ en un sistema como el de la Ecuación 2.7, se obtiene una aproximación del problema original pero que puede ser resuelta en un número finito de pasos conservando un comportamiento similar al del sistema original.

Mas aún, se puede asociar la solución del sistema presentado en la Ecuación 2.8 con el comportamiento de un sistema de eventos discretos (ya que realizan un número finito de cambios), pero no al de un sistema de tiempo discreto. En este caso, los eventos corresponden a los cruces de las variables de estado por los niveles de discretización de las mismas, cuando ocurren pueden provocar cambios en las derivadas de los estados, que conducen a una reprogramación del tiempo en que ocurre el próximo cruce de nivel.

Para poder ver como se puede generalizar esta idea, se deberá introducir antes algunas herramientas que permiten representar y simular sistemas como el de la Ecuación 2.8.

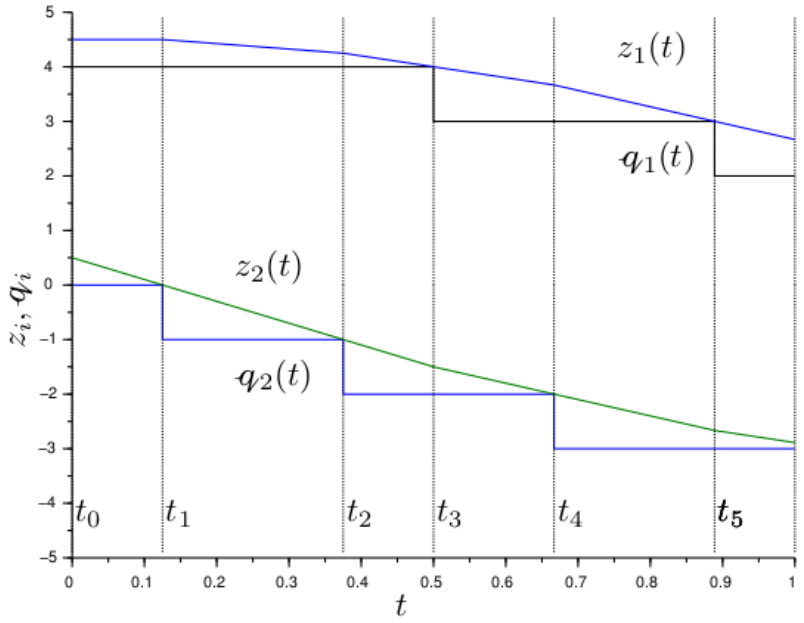


Figura 2.5: Simulación de las trayectorias del sistema de la Ecuación 2.8 (Inicio).

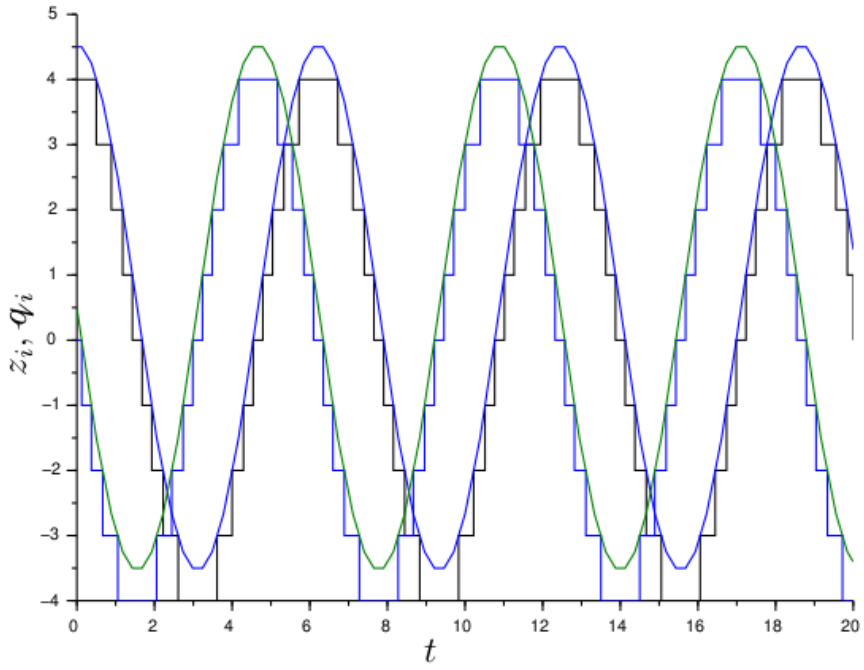


Figura 2.6: Simulación de las trayectorias del sistema de la Ecuación 2.8

2.3.2. Sistemas de Eventos Discretos y DEVS

Todos los métodos de tiempo discreto aproximan las ecuaciones diferenciales por sistemas de tiempo discreto (ecuaciones en diferencia) de la forma:

$$\mathbf{z}(t_{k+1}) = \mathbf{f}(\mathbf{u}(t_k), t_k) \tag{2.9}$$

Con la nueva idea de cuantificar las variables de estado, sin embargo, se obtienen sistemas que son discretos (ya que realizan un número finito de cambios), pero no son de tiempo discreto.

Se verá que esta nueva manera de aproximar las ecuaciones nos lleva a *sistemas de eventos discretos*, dentro del formalismo DEVS.

DEVS, cuyas siglas provienen de *Discrete EVent System specification*, fue introducido por Bernard Zeigler a mediados de la década de 1970. DEVS permite representar todos los sistemas cuyo comportamiento entrada/salida pueda describirse mediante secuencias de eventos.

En este contexto, un *evento* es la representación de un cambio instantáneo en alguna parte del sistema. Como tal, puede caracterizarse mediante un valor y un tiempo de ocurrencia. El valor puede ser un número, un vector, una palabra, o en general, un elemento de algún conjunto.

2.4. Método de los Sistemas de Estados Cuantificados (QSS)

La mayor desventaja de la idea de la cuantificación de estado presentada anteriormente era la aparición de oscilaciones infinitamente rápidas en las variables de estado y de estado cuantificadas. Estas oscilaciones ocurren cuando algún estado z_i se mueve alrededor de algún valor cuantificado. Un pequeño cambio en z_i provoca un gran cambio en q_i, lo que a su vez puede provocar otro pequeño cambio en z_i logrando un comportamiento cíclico.

Una solución a esto es utilizar histéresis en la cuantificación. Agregando histéresis a la relación entre $z(t)$ y $q(t)$, las oscilaciones en esta última pueden ser sólo debidas a oscilaciones grandes en $z(t)$. Si la derivada $\dot{z}(t)$ es finita, una oscilación grande en $z(t)$ no puede ocurrir instantáneamente sino que tiene una frecuencia máxima acotada.

En base al cambio de la función de cuantificación sin histetéresis por una con histéresis surgió toda una familia de métodos de integración por cuantificación denominados QSS (Quantaized State System). Dentro de esta familia existen métodos de primer, segundo y tercer orden denominados QSS1, QSS2 y QSS3 respectivamente.

2.4.1. Solver QSS

Como ya mencionamos en la Sección 2.3.2, los métodos de los sistemas de Estados Cuantificados (QSS) reemplazan la discretización del tiempo de los algoritmos de integración numérica clásica por la cuantificación de las variables de estado por medio de una función de cuantificación histerética.

El solver QSS independiente simula modelos que pueden contener discontinuidades. Estos modelos son representados de la siguiente manera:

$$\dot{z}(t) = \mathbf{f}(\mathbf{z}, \mathbf{d}, t) \tag{2.10}$$

donde $\mathbf{d}$ es un vector de variables discretas que puede cambiar cuando una condición

$$ZC_i(\mathbf{z}, \mathbf{d}, t) = 0 \tag{2.11}$$

para algún $i \in \{1, \ldots, z\}$ se cumple. Los componentes de ZC_i pueden agruparse en un vector de funciones de cruce cero. Cuando una de estas condiciones se cumple, el estado y las variables discretas pueden cambiar de acuerdo a el correspondiente manejador de eventos

$$(\mathbf{z}(t), \mathbf{d}(t)) = H_i(\mathbf{z}(t^-), \mathbf{d}(t^-), t). \tag{2.12}$$

Estos modelos son simulados usando métodos QSS que aproximan la Ecuación 2.10 por medio de:

$$\dot{z}(t) = \mathbf{f}(\mathbf{q}, \mathbf{d}, t) \tag{2.13}$$

donde cada $q_i(t)$ es una aproximación polinómica por partes de la componente correspondiente al estado $x_i(t)$.

La simulación se realiza mediante tres módulos que interactúan en tiempo de ejecución:

1. El *Integrador*, que integra la Ecuación 2.13 asumiendo que la trayectoria de estado cuantificado $\mathbf{q}(t)$ es conocida.

2. El *Cuantificador*, que compute $\mathbf{q}(t)$ a partir de $\mathbf{z}(t)$ de acuerdo a el método QSS en uso y sus ajustes de tolerancia (hay un *Cuantificador*) distinto para cada método QSS). De esa manera, los coeficientes polinomiales de cada estado cuantificado $q_i(t)$ y calcula la próxima vez que comienza una nueva sección polinómica (es decir, cuando se cumple la condición $|q_i(t) - z_i(t)| = \Delta Q_i$).

3. El *Modelo* que calcula las derivadas del estado escalar $z_i = f_i(\mathbf{q}, \mathbf{d}, t)$, las funciones de cruce cero $ZC_i(\mathbf{z}, \mathbf{d}, t)$, y los correspondientes manejadores de eventos $H_i(\mathbf{q}, \mathbf{d}, t)$. Además, provee información estructural requerida por los algoritmos.

La información estructural del modelo es extraída automáticamente en tiempo de compilación por el módulo Generador del Modelo. Este módulo toma un modelo estándar descripto en un subconjunto del lenguaje Modelica y produce una instancia del módulo Modelo según sea requerido por el solver QSS incluyendo información estructural y la posibilidad de evaluar derivadas de estado escalares por separado.

La información estructural se compone por cuatro matrices de incidencia binarias.

- *SD* (derivadas a estados) es tal que $SD_{i,j} = 1$ si z_i está involucrada en el cálculo de z_j.

- *SZ* (derivadas a funciones de cruce cero) es tal que $SZ_{i,j} = 1$ si z_i está involucrada en el cálculo de $\dot{z_j}$.

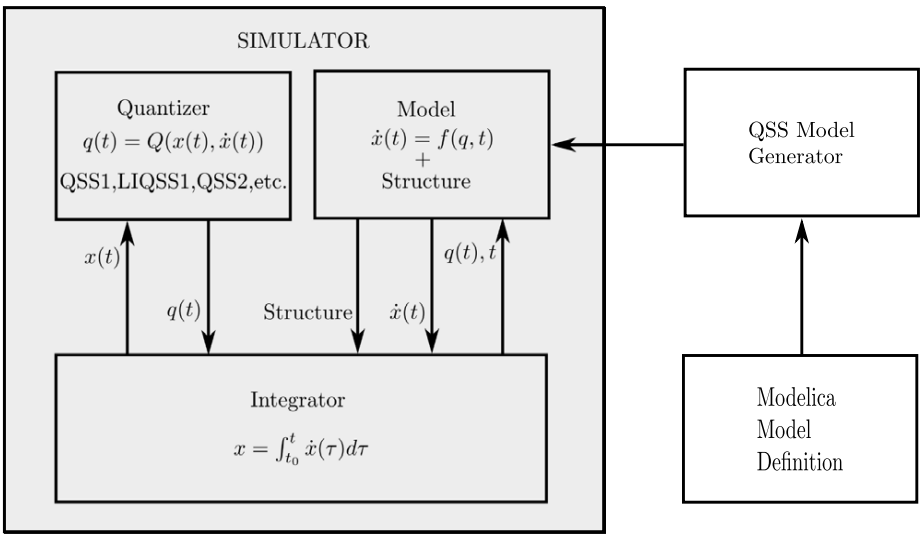


Figura 2.7: Solver QSS independiente — esquema básico de interacciones.

- *HD* (manejadores a derivadas de estado) es tal que $HD_{i,j} = 1$ si la ejecución del manejador H_i cambia algún estado o variable discreta involucrada en el cálculo de $\dot{z}$.
- *HZ* (manejadores a funciones de cruce cero) es tal que $HZ_{i,j} = 1$ si la ejecución del manejador H_i cambia algún estado o variable discreta involucrada en el cálculo de ZC_j.

Teniendo en cuenta que la mayoría de los sistemas grandes están débilmente acoplados, la matrices estructuras se almacenan en forma dispersa.

La simulación es llevada a cabo por el módulo *Integrador*, que avanza el tiempo de simulación ejecutando los pasos de simulación. Cada paso de simulación puede corresponder a un cambio en la variable cuantificada o a la ejecución de un manejador de eventos desencadenado por una función de cruce cero. El integrador almacena el estado, el estado cuantificado y los valores de las variables discretas. También almacena el instante de tiempo del siguiente cambio en el estado cuantificado y el instante de tiempo de la siguiente función de cruce cero.

2.5. Modelica

Modelica [9] y [8] es un lenguaje orientado a objetos para modelos de sistemas físicos con el propósito de una simulación eficiente. El lenguaje unifica y generaliza previous lenguajes de modelado orientados a objetos.

En Modelica los programas son construídos con clases. Como en cualquier otro lenguage orientado a objetos, las clases contienen variables, es decir atributos de clase que representan datos. La principal diferencia comparado con lenguajes orientados a objetos tradicionales es que en lugar de funciones (métodos) se usan ecuaciones para especificar comportamiento. Las ecuaciones pueden escribirse explícitamente (`a=b`), o por medio de herencia de otras clases. Las

ecuaciones también pueden ser especificadas por medio de la sentencia `connect`. La sentencia `connect(v1, v2)` expresa acoplamiento entre las variables `v1` y `v2`. Estas variables son llamadas conectores.

```
1 class ClassName
2     Declaration1
3     Declaration2
4     ...
5 equation
6     equation1
7     equation2
8     ...
9 end ClassName ;
```

Esta es la estructura típica de una clase en Modelica. `ClassName` es su nombre, posee un espacio de declaraciones y debajo de la sentencia `equation`, una serie de ecuaciones. En este trabajo no entraremos en detalles, sí nos interesa la sección de ecuaciones, puesto nos servirán de ejemplo para entender la comunicación entre variables y su necesidad de sincronización.

Capítulo 3

Simulacion en Paralelo de Modelos Matemáticos

El modelado y la simulación de sistemas continuos está en el centro de las investigaciones científicas y de las distintas áreas de la ingeniería. Cada año se desarrollan modelos más grandes y más complejos y sus simulaciones acarrean un alto costo computacional. Con la aparición de procesadores multinúcleo y clústeres de computadoras de múltiples nodos, la simulación en paralelo de sistemas de tiempo continuo se volvió una forma usual de reducir el tiempo de estas simulaciones. Como mencionamos anteriormente, Modelos de tiempo continuo son usualmente expresados como (o transformados a) un conjunto de ecuaciones diferenciales ordinarias (ODE), donde deben aplicarse algoritmos de integración numérica para resolverlos. [5, 6]

3.1. Simulación en Paralelo de Modelos de Tiempo Continuo

Diferentes estrategias han sido propuestas a lo largo de los años para simulación ODE en paralelo basadas en algoritmos de integración numérica clásicos y están caracterizadas acorde a las computaciones que estas paralelizan. Las principales categorías son la siguientes:

- *Paralelismo a través del método:* Esta técnica consiste en adaptar el solver para realizar cálculos en paralelo. Por ejemplo, inversión de matrices en métodos implícitos o permientiendo la computación de varios pasos de tiempo simultáneamente.

- *Paralelismo a través del modelo:* Esta técnica particiona las ecuaciones del modelo de forma óptima que luego se computarán en paralelo.

- *Paralelismo entre los pasos del tiempo:* Los sucesivos pasos de la simulación son computados en paralelo.

3.2. Simulación en Paralelo de Sistemas de Eventos Discretos

La idea básica es dividir el modelo en varios sub-modelos y simularlos concurrentemente en diferentes *procesadores lógicos* (LPs), cada uno teniendo su propio tiempo lógico. Como los sub-modelos usualmente son interdependientes, el tiempo lógico de las diferentes subsimulaciones se deben sincronizar para satisfacer la restricción de causalidad, es decir, la preservación del orden cronológico de los eventos.

La bibliografía divide los los diferentes enfoques en tres categorías:

- *Algoritmos conservadores*, donde hay un mecanismo de sincronización que fuerza a cada LP a esperar hasta estar seguro de que no recibirá mensajes del "pasado". Usualmente logran pequeñas mejoras de tiempo y sufren de ciertos problemas, como posibles bloqueos mutuos.

- *Algoritmos optimistas*, donde cada LP avanza su tiempo lógico tanto como puede y retrocede cuando encuentra una inconsistencia entre los mensajes intercambiados. Este enfoque permite más paralelismo que los algortimos conservadores a costa de tener que guardar estados de simulación intermedios y costosos mecanismos de retroceso.

- *Algoritmos totalmente desincronizados* evitan completamente la sincronización, es decir, cada LP avanza su tiempo lógico tan rápido como puede sin importar el tiempo lógico de otro LP. Este enfoque alcanza grandes mejoras de tiempo a costa de la introducción de errores debido a violaciones de la restricción de causalidad.

3.3. Simulación en Paralelo QSS

Ninguno de estos enfoques encaja bien en el contexto de los algoritmos QSS. En este caso, la sincronización estricta no permite casi ninguna computación concurrente. También, métodos optimistas requerirían una gran cantidad de memoria para implementar los mecanismos de retroceso de sistemas grandes. Finalmente, técnicas totalmente desincronizados introducirían errores numéricos inaceptables.

Una nueva técnica para simulación en paralelo de modelos QSS fue introducida en [7].

3.3.1. Idea general

La técnica de paralelización presentada se basa en particionar el modelo entre P submodelos, donde cada submodelo es simulado por un procesador lógico diferente.

En cada paso de simulación de cada submodelo, el correspondiente LP chequea si la variable que cambió debe ser comunicada a otros LPs usando información estructural. En ese caso, los nuevos valores y las correspondientes marcas de tiempo son informadas a través de mecanismos de comunicación entre procesos.

Esta sincronización no estricta se logra calculando un tiempo virtual global *gvt* (igual al tiempo lógico mínimo de todos los LPs) y no permitiendo que los

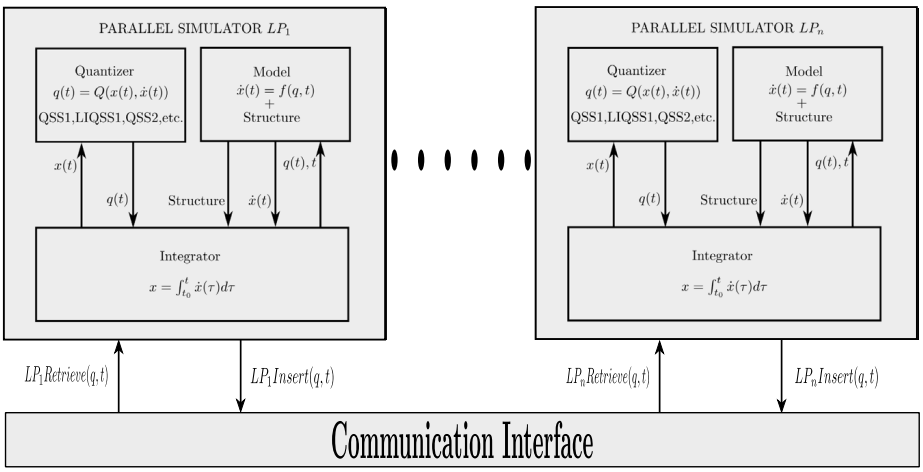


Figura 3.1: Solver QSS en Paralelo - esquema básico de interacciones.

LPs avancen más allá de $gvt + \Delta t$. Para eso, cuando un LP prepara su siguiente paso de simulación más allá de $gvt + \Delta t$, ingresa en una rutina de espera hasta que gvt avanza o se detecta un cambio en una variable calculada por otro LP.

Cada LP está compuesto por un módulo *Integrador*, un módulo *Cuantificador* y un módulo *Modelo*, como en la simulació secuencial descripta en la Sección 2.4.1.

3.3.2. Partición del modelo

Los algoritmos de simulación en paralelo QSS requiere que el modelo original sea divido en p submodelos, de modo que cada submodelo sea simulado por un LP diferente.

Para obtener una paralelización efectiva, la partición del modelo debe ser balanceada y, al mismo tiempo, la comunicación entre disferentes LPs debe ser minimizada.

3.3.3. Estructura entre procesos

Como explicamos en la Sección 2.4.1, la información estructural del modelo es presentada en las matrices SD, SZ, HD y HZ. Estas matrices estructurales representan la influencias directas de estados y manejadores de eventos en las derivadas de estado y las funciones de cruce cero.

Una vez que se particionó el modelo en p submodelos, puede pasar que un cambio en una variable de estado o en la ejecución un manejador de eventos, calculado en un determinado submodelo tenga influencia directa en otras derivadas de estado o funciones cruce cero calculadas en otros submodelos.

En consecuencia, es necesario un mecanismo de comunicación entre procesos, lo que a su vez requiere el conocimiento de información de la estructura entre procesos.

Esta información puede ser proveída mediantes dos matrices de incidencia en cada LP:

- SO^k sea tal que $SO^k_{i,l} = 1$ si el i-ésimo estado del k-ésimo submodelo tiene influencia sobre alguna derivada de estado o función de cruce cero calculada por el l-ésimo submodelo.

- HO^k sea tal que $HO^k_{i,l} = 1$ si el i-ésimo manejador de eventos se ejecuta en el k-ésimo submodelo tiene influencia sobre alguna derivada de estado o función de cruce cero calculada por el l-ésimo submodelo.

Las matrices SO^k y HO^k se calculan en el momento de la inicialización basándose en la información estructural de las matrices SD, SZ, HD, HZ y la información del particionado del modelo. Como las demás matrices estructurales, se almacenan en forma dispersa.

3.3.4. Algoritmo de simulación

El algoritmo de simulación paralela se implementa localmente en cada procesador lógico. Como ya hemos explicado, cada LP contiene un módulo *Integrador*, un módulo *Cuantificador* y un módulo *Modelo*. Los modulos *cuantificador* y *Modelo* son idénticos a sus homólogos secuenciales, mientras que el *Integrador* ahora incluye los mecanismos de comunicación y sincronización.

La comunicación involucra informar los nuevos valores de variables de estado y discretas a otros LPs después de cambios de estado o ejecuciones de manejadores de eventos que, según la estructura entre procesos, afectan a derivadas de estado o funciones de cruce cero calculadas en otros LPs. Por otro lado, los LP deben verificar si esos nuevos valores fueron cambiados.

Como mencionamos antes, los mecanismos de sincronización limitan el avance de la simulación en cada proceso para mantener una diferencia acotada entre los tiempos de simulación lógicos entre los distintos LPs.

El módulo *Integrador* debe calcular el mínimo tiempo virtual global gvt, que consiste en el mínimo entre los tiempos de simulación local de todos los LPs y una rutina de sincronización para evitar que el tiempo local t^k avance mucho más allá de gvt. Adicionalmente, tiene en cuenta los cambios entrantes en las variables de entrada informados por diferentes LP.

Cuando el siguiente paso en un LP corresponde a un cambio en una variable cuantificada sólo actualiza las derivadas de estado que son calculadas en el LP actual, y, adicionalmente, informa los valores y la correspondiente marca de tiempo a los otros LPs que computan las derivadas de estado restantes afectadas por el cambio de estado cuantificado.

Una versión detallada del algoritmo se puede encontrar en [7, Sección 3.5].

3.3.5. Sincronización entre procesos

Como indicamos en la Sección 3.3.1, la simulación en paralelo del solver QSS usa un mecanismo de sincronización no estricta que es necesario para asegurar que los cálculo son realizados en paralelo.

Este mecanismo de sincronización requiere que, antes de cada paso de simulación, los LPs calculen

$$gvt = \min_{1 \le l \le P} (t^l) \tag{3.1}$$

como el tiempo lógico global mínimo de todos los LPs. Entonces, cada LP limita el avance de su tiempo de simulación local T_k al valor $gvt + \Delta t$, donde Δt es un parámetro definido por el usuario. Notar que Δt es una cota superior de la diferencia entre los tiempos de simulación de todos los LPs y define qué tan estricta es la sincronización.

El procedimiento de sincronización de cada LP implementa una rutina de espera mientras la condición

$$t^k > gvt + \Delta t \tag{3.2}$$

se verifica. Cuando gvt avanza (porque otros LPs actualizaron sus tiempos de simulación local) o cuando T_k vuelve hacia atrás (porque el k-ésimo detectó un cambio externo) y la condición de la Ecuación 3.2 ya no se cumple, la simulación del k-ésimo continúa.

3.3.6. Comunicación entre procesos

Ya fue mencionado que los diferentes LPs informan cambios de variables de estado y discretas durante la simulación. Para eso, cada proceso lógico tiene una lista de cambios ocurridos en las variables computadas en otros LPs que son necesarias para los cálculos locales.

Si es detectado un cambio con una marca de tiempo $m.t$ menor que el último paso realizado por LP^{k-}, entonces se modifica la marca de tiempo a t^{k-}. De esa manera, evitamos que la simulación vuelva hacia átrás en el tiempo.

Notar que la modificación de esta marca de tiempo es el único que efecto tienen los errores de sincronización en los resultados de la simulación. De todos modos, la diferencia entre $m.t$ y t^{k-} está limitada por Δt.

El mecanismo de comunicación entre hilos es híbrido:

- Los cambios de estados son asíncronamente informados usando una lista de cambios.
- Los cambios discretos también son informados en una lista de cambios, pero de manera síncrona donde el LP que comunica el nuevo valor debe esperar que todos los LPs los procesen.

El último caso asegura que se procesen los nuevos valores de todas las variables discretas por todos los LPs afectados antes de que el LP responsable por el cambio continúe con la simulación. Teniendo en cuenta que cambios discretos pueden provocar grandes cambios en variables discretas y de estado, esta política previene la introducción de errores números grandes debido a la falta de sincronización después de discontinuidades.

Es importante mencionar que en la mayoría de los modelos que contienen discontinuidades el número de cambios continuos es significativamente más grande que el número de cambios discretos. Por lo tanto, en la práctica, la comunicación es en su mayoría asíncrona.

Capítulo 4

Balance de Carga para Simulación QSS en Paralelo

4.1. Conceptos teóricos

Un componente importante de la simulación numérica en paralelo es la asignación de trabajo a los procesadores de forma óptima. La optimalidad que involucra la minimización del error de simulación alcanzado así como la minimización del tiempo y recursos computacionales necesarios para la simulación. En otras paralabras, el objetivo, dada una arquitectura multinúcleo, es distribuir los cálculos entre los procesadores disponibles de forma tal que se obtenga una solución correcta y eficiente. En general, este objetivo se persigue distribuyendo cantidades de trabajo aproximadamente iguales entre las tareas, de esa manera todas las tareas se mantienen ocupadas todo el tiempo, mientras intentamos minimizar la comunicación y sincronización entre procesos durante la simulación y es conocido como *balance de carga*.

El balance de carga es importante para los programas en paralelo por razones de rendimiento. Por ejemplo, si todas las tareas están sujetas a un punto de sincronización de barrera, la tarea más lenta determinará el rendimiento general.

Podemos distinguir tres tipos de problemas de balance de carga:

- Balance de carga *estático*, donde la asignación a los procesadores se realiza al inicio de la simulación, sin tener ninguna informacion sobre los resultados reales de la simulación.
- Balance de carga *semi-estático*, donde la asignación a los procesadores ocurre al comienzo de la simulación, pero un conocimiento de dominio extra u otra información relacionada puede ser aprovechada para mejorar la solución alcanzada.
- Balance de carga *dinámico*, donde la asignación a los procesadores se hace dinámicamente en el transcurso de la simulación.

En este trabajo, sólo abordamos el primer caso, balance de carga *estático*, donde el balance se realiza antes de la simulación real.

En la comunidad de computación paralela, el balance de carga *estático* es frecuentemente formulado como un problema de particionamiento de grados,

donde los grafos se usan para representar los cálculo, con nodos representando unidades de cálculo y aristas representando dependencia de datos.

4.1.1. Métodos de Particionado de Grafos Estático

Métodos Geométricos

Técnicas geométricas son ampliamente utilizadas en la paralelización de métodos de elementos finitos y partículas. Solo usan coordenadas geométricas de objetos y asignan objetos de iguales pesos a los procesadores mientras agrupan físicamente objetos cercanos dentro de subdominios independientemente de si esos vértices están muy conectados. Son generalmente rápidos y fáciles de implementar, pero inducen costos de comunicación más altos que los particionadores de grafos.

Métodos Combinatorios

Intentan agrupar vértices áltamente conectados basándose en la información de adyacencia del grafo, en lugar de las coordenadas de los nodos como en los métodos geométricos. Por lo tanto, suelen alcanzar balances de carga de mejor calidad pero con la desventaja de ser más demandantes computacionalmente.

Métodos Espectrales

Calculan la matriz de adyacencia de Laplacian del grafo $\mathcal{G}$, y luego calculan el vector de Fiedler. Usando el vector de Fiedler, se calcula una distancia basada en conectividad, que luego se usa para dividir vértices en dos grupos balanceados. Estos métodos en general producen particionamientos de alta calidad pero escalan ineficientemente cuando el tamaño del grafo crece.

Métodos Multinivel

Funciona aplicando una o más etapas de particionado. Cada etapa reduce el tamaño del grafo colapsando vértices y aristas, particiona el grafo más pequeño, luego vuelve al grafo original, mapeando correctamente nodos y aristas, y refina esta partición. En particular, METIS [10], es una implementación de métodos multinivel y será utilizado para comparar los resultados obtenidos.

4.1.2. Grafo Computacional QSS

Los cálculos se pueden modelar de forma natural mediante un grafo en donde los vértices representan unidades computacionales o tareas y las aristas representan dependencias entre unidades de computación individuales.

En líneas generales, un grafo computacional puede modelarse como un grafo dirigido, $\mathcal{G}$, con pesos definidos en sus nodos y en sus aristas.

Más especificamente, $\mathcal{G} = (\mathcal{V}, \mathcal{E}, l, w)$, donde:

$\mathcal{V}$ el conjunto de nodos/las unidades de computación más pequeñas
$\mathcal{E} : \mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ el conjunto de aristas/dependencias entre unidades
$l : \mathcal{V} \to \mathcal{R}^+$ asigna un peso computacional a cada nodo
$w : \mathcal{E} \to \mathcal{R}^+$ asigna un peso comunicacional a cada arista

4.1.3. Balance de carga como problema de particionamiento de grafos

Queremos elegir una partición de nodos $\mathcal{C}$ en P particiones disjuntas $\mathcal{V} = (\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_p)$ tales que:

- La suma del peso de los nodos es casi la misma (particionamiento equilibrado).
- Se minimiza la suma del peso de todas las aritas conectan las diferentes particiones.

Así, denotamos como costo la comunicación entre clusters (mejor conocido en la bibliografía como *edge-cut*).

$$cost(\mathcal{C}) = edgeCut(\mathcal{C}) = \sum_{\mathcal{C}_i,\mathcal{C}_j} \{w(u,v) | u \in \mathcal{C}_i, v \in \mathcal{C}_j, i \neq j\}, \tag{4.1}$$

$$\begin{aligned} &\underset{\text{todo } \mathcal{C}}{\text{minimizar}}\ edgeCut(\mathcal{C}) \\ &\text{sujeto a } \sum_{u \in \mathcal{C}_i} l(u) = \frac{\sum_{v \in \mathcal{V}}}{P}, \forall i. \end{aligned} \tag{4.2}$$

El problema anterior de partición balanceada de grafos se puede mostrar que es un problema NP-completo [1]. Por lo tanto, las soluciones existentes de este problema son derivadas usando heurísticas y algoritmos de aproximación. Típicamente, la restricción del balance de carga de la Ecuación 4.2 se relaja obteniendo una ecuación de la siguiente forma:

$$\begin{aligned} &\underset{\text{todo } \mathcal{C}}{\text{minimizar}}\ edgeCut(\mathcal{C}) \\ &\text{sujeto a } (1-\epsilon)\frac{\sum_{v \in \mathcal{C}}}{P} \leq sum_{u \in \mathcal{C}_i} l(u) \leq (1+\epsilon)\frac{\sum_{v \in \mathcal{C}} l(v)}{P} \end{aligned} \tag{4.3}$$

donde el parámetro $0 < \epsilon < 1$ controla cuanto peso puede diferir entre el peso medio de las particiones y se llama *desbalance*.

Habiendo mencionado el problema de optimización en términos de teoría de grafos, necesitamos una forma de obtener dicho grafo computacional $\mathcal{G}$ de un modelo implementado en Modelica.

Podemos identificar los siguientes tipos de cálculos "elementales". Estos serán los nodos de nuestro grafo

- *Nodos Continuos* que son los responsables de calcular $\dot{x}_1 = f_1(x_1, \ldots, x_n, d_1, \ldots, d_m, t)$ y luego integrar y cuantificar para obtener las variables de estado cuantificadas $q_i = \mathcal{Q}(x_i) = \mathcal{Q}(\int \dot{x}_i, dt)$. Podemos interpretarlos como una Función Estática acoplada con un Integrador Histerético Cuantificado en serie.
- *Nodos Discontinuos* que contienen funciones de cruce cero y sus correspondientes funciones manejadoras que actualizan las variables discretas necesarias (y posiblemente variables de estado por medio de una sentencia `reinit`) cuando se cumple la condición de la función de cruce cero.

- *Nodos Algebraicos* que contienen ecuaciones algebraicas y variables.

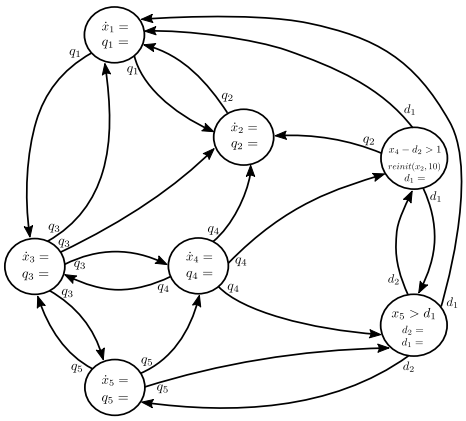
Las aristas de los Grafos Computacionales (en el conjunto $\mathcal{E}$), denotan dependencias entre variables que son calculadas en nodos y necesarias en una expresión contenida en otro nodo. Más específicamente, una arista $e = (i, j)$ que conecta los nodos u_i y u_j, significa que una variable (de estado o discreta) computada en el nodo u_i es contenida en una expresión (ecuación o condición de cruce cero) del nodo u_j o visceversa.

En cuanto a los pesos asignados a las aristas $w : \mathcal{E} \rightarrow \mathcal{R}^+$, relacionados al costo en la comunicación, en este trabajo son ingresados manualmente por el usuario, al igual que los pesos asignados a los nodos en la función $l : \mathcal{V} \rightarrow \mathcal{R}^+$.

Un ejemplo ilustrativo de un Grafo Computacional

```
1   equation
2       der(x1) = x1+x2-2*x3
            *d1;
3       der(x2) = 2*x1-3*
            x3-x4;
4       der(x3) = x1-x3 +(
            x4-x5)^2;
5       der(x4) = x3+x5;
6       der(x5) = 2*x3-d2;
7   algorithm
8       when (x4-d2>1) then
9       d1:=0;
10      reinit(x2, 10);
11      end when;
12      when (x5>d1) then
13      d1:=1;
14      d2:=x5-x4;
15  end when;
```

(a) Ejemplo de un modelo en Modelica

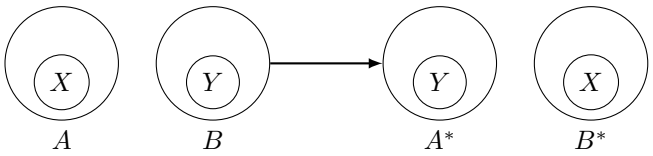


(b) Grafo computacional correspondiente

Figura 4.1: Un ejemplo ilustrativo de un grafo computacional

En una plataforma paralela distribuida, el peso de las aristas significa costo de comunicación, tal como es mencionado en la Sección 4.1.3, que debe ser minimizado para acelerar la ejecución del los cálculos en general.

Una arista no significa comunicación precisamente, sino el lanzamiento de un cálculo debido a actualizaciones de variables y la necesidad de sincronizar los respectivos Procesadores Lógicos (LPs). El peso de una arista representa la *cantidad* de sincronización necesaria durante la simulación. Como ya fue mencionado brevemente en la Sección 3.3.2, el principal objetivo del balance de carga involucra la minimización del *edgeCut*. En el mejor de los casos, lograr *edgeCut* cero para un cierto particionamiento, resultaría en una simulación totalmente en paralelo y sin necesidad de sincronización. La minimizacion de la sincronización está directamente relacionada a la minimización del error que se genera debido a latencias y errores de sincronización.



$$A^* = A - X + Y$$
$$B^* = B - Y + X$$

Figura 4.2: Descripción gráfica de la fase de optimización

4.2. Algoritmo Clásico de Kernighan–Lin

El problema de particionar los nodos de un grafo con costos en sus aristas en particiones de un tamaño dado así como la minimización de la suma del costo del *edgeCut* fue presentado en [11] y es la base del algoritmo de particionamiento de grafos computacionales de este trabajo. Daremos una introducción al mismo en esta sección.

Tomemos un grafo computacional como el definido en la Ecuación 4.1.2. Sea k un entero positivo, buscamos particionar $\mathcal{G}$ en k particiones disjuntas.

Sea S un conjunto de $2n$ puntos, con una matriz de costo asociada $C = (c_{ij})$, $i,\ j = 1, \ldots, 2n$. Asumimos sin pérdida de generalidad que C es una matriz simétrica y que $c_{ii} = 0$ para todo i. Queremos partir S en dos conjuntos A y B, cada una con n puntos, tales que el "costo externo" $T = \sum_{A \times B} c_{ab}$ se minimice.

En esencia, el método es este: comenzando con cualquier partición A, B de S, intentar reducir el costo externo T por medio de una serie de intercambios de subconjuntos de A y B; los subconjuntos se eligen por un algoritmo a describir. Cuando no se encuentra una mejora posible, la partición resultante A', B' es localmente mínimas con respecto al algoritmo. La partición resultante tiene altas probabilidad de ser una partición mínima global.

Este proceso puede repetirse con tantas particiones iniciales arbitrarias como se desee, hasta obtener tantas particiones localmente mínimas como se desee.

Dados S y (c_{ij}) supongamos que (A^*, B^*) es una partición con costo mínimo. Sean A y B dos particiones arbitrarias. Claramente hay subconjuntos $X \subset A$, $Y \subset B$ con $|X| = |Y| \leq n/2$ tales que intercambiando X e Y se obtiene A^* y B^* como se muestra en la Figura 4.2.

El problema es identificar X e Y a partir de A y B, sin considerar todas las posibles opciones. El proceso que estamos describiendo encuentra X e Y aproximadamente, identificando secuencialmente sus elementos.

Definamos para cada $a \in A$, un *costo externo* E_a

$$E_a = \sum_{y \in B} c_{ay} \tag{4.4}$$

y un *costo interno* I_a

$$I_a = \sum_{x \in A} c_{ax} \tag{4.5}$$

Análogamente, definimos E_b, I_b para cada $b \in B$.

Sea

$$D_x = E_x - I_x \text{ para todo } x \in S; \tag{4.6}$$

D_x es la diferencia entre costo externo e interno.

Considerando cualquier $a \in A$, $b \in B$. Si a y b son intercambiados, la *ganancia* (esto es, la reducción del costo) es precisamente

$$D_a + D_b - 2c_{ab} \tag{4.7}$$

Esto es presentado como un lema en [11] con su respectiva demostración.

Explicado esto, el algoritmo hace lo siguiente:

Primero, calcula los valores D para todos los elementos de S.

Elige los $a_i \in A$, $b_i \in B$ tales que:

$$g_1 = D_{a_i} + D_{b_i} - 2c_{a_i b_i}$$

es máximo; a_i y b_i corresponden a la ganancia más alta de un posible intercambio.

Reservar a_i y b_i temporalmente, y llamarlos a'_i y b'_i respectivamente.

Recalcular los valores D para los elementos de $A - \{a_i\}$ y $B - \{b_i\}$, por medio de:

$$D'_x = D_x + 2c_{xa_i} - 2c_{xb_i} \qquad x \in A - \{a_i\},$$
$$D'_y = D_y + 2c_{yb_i} - 2c_{xa_i} \qquad x \in B - \{b_i\}.$$

Repetir el segundo paso, eligiendo un par a'_2, b'_2 a partir de $A - \{a'_i\}$ y $B - \{b'_j\}$ tales que $g_2 = D_{a'_2} + D_{b'_2} - 2c_{a'_2 b'_2}$ (a_i y b_i no son tenidos en cuenta en esta elección). Así, g_2 es la ganancia adicional cuando los puntos a'_2 y b'_2 son intercambiados junto con a'_1 y b'_1.

Continuar hasta que se hayan agotado todos los nodos.

Si $X = a'_1, a'_2, \ldots a'_k$, $Y = b'_1, b'_2, \ldots, b'_k$, entonces la disminución en costo cuando los conjuntos X y Y son intercambiados es exactamente $g_1 + g_2 + \ldots, g_k$. Por supuesto, $\sum_1^n g_i = 0$, puesto que significaría intercambiar $X = A$ con $Y = B$ completos. Notar que algunos de los g_i's son negativos (esto significa que el intercambio genera un aumento en la comunicación externa) a menos que todos sean cero.

Elegir k tal que la suma parcial $\sum_{i=1}^k g_i = G$ sea máxima. Si $G > 0$, el intercambio signifacaría una reducción del costo. Después, el resultado de este intercambio puede tomarse como la partición inicial y el procedimiento se repite desde el primer paso.

Si $G = 0$, hemos llegado a una partición localmente óptima.

Notar que el proceso no termina inmediatamente cuando algún g_i es negativo. Esto significa que el proceso puede secuencialmente identificar conjuntos cuyo intercambio de solo algunos elementos incrementaría el costo, mientras que el intercambio de los conjuntos completos produciría ganancia.

Capítulo 5

Grafos Basados en Conjuntos

Los Grafos Basados en Conjuntos (GBC) fueron introducidos en [17], y tienen por objeto representar de manera compacta grafos de gran tamaño.

5.1. Definición

Vértice-conjunto Es un conjunto de vértice $V = \{v_1, v_2, \ldots, v_n\}$.

Arista-conjunto Dados dos vértices-conjunto V^a y V^b, con $V^a \cap V^b = \emptyset$, una arista-conjunto que une V^a con V^b es un conjunto de aristas no repetidas $E[\{V^a, V^b\}] = \{e_1, e_2, \ldots, e_n\}$ donde cada arista es un conjunto de dos vértices, $e_1 = v_k^a \in V^a, v_l^b \in V^b$ donde $1 \leq i \leq n$.

Una manera simple y conveniente de representar aristas-conjuntos es la que se propone a continuación.

Mapa Es una colección no ordenada de pares finita, donde el segundo valor de cada par representa el valor asignado al primer elemento del par. Dado el mapa $map = \{(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)\}$, se usará la notación $map(a_i) = b_i$ para indicar que b_i es el resultado de aplicar el mapa map a a_i.

Se define $map^{a,b}$ como un mapa, cuyo dominio serán los elementos de una arista-conjunto $E^a \in G$, y cuya imagen estará contenida en $V^b \in V$.

Sea $E^h = [\{V^i, V^j\}]$. Se puede caracterizar esta arista-conjunto usando dos mapas que relacionan cada arista "común" $e_k^h = \{v_r^i, v_s^j\} \in E^h$ con los vértices que unce, $v_r^i = map^{h,i}(e_s^j)$ y $v_s^j = map^{h,j}(e_k^j)$. De este modo, la arista-conjunto queda definida por compresión como:

$$E^h = \bigcup_{i=1}^{n} \{\{map^{h,i}(e_k^h), map^{h,j}(e_k^h)\}\}$$

Grafo Basado en Conjuntos Es un par $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ donde:

- $\mathcal{V} = \{V^1, \ldots, V^n\}$ es un conjunto de vértices-conjunto disjuntos.

- $\mathcal{E} = E^1, \ldots, E^m$ es un conjunto de aristas-conjunto que une vértices-conjunto de $\mathcal{V}$.

5.2. Ejemplo de un Grafo Basado en Conjuntos

Podemos tomar un grafo tradicional no dirigido $G = (V, E)$ donde:

- $V = \{1, \ldots, 10\}$ son sus nodos.
- $E = \{\{1,6\}, \{2,7\}, \{3,8\}, \{4,9\}, \{5,10\}, \{1,7\}, \{2,8\}, \{3,9\}, \{4,10\}, \{5,6\}\}$ son sus aristas.

Y podemos dar una definición equivalente del mismo usando un GBC $\mathcal{G} = (\mathcal{V}, \mathcal{E}, map_1, map_2)$ donde:

- $\mathcal{V} = \{[1:5], [6:10]\}$ son sus nodos.
- $\mathcal{E} = \{[1:5], [6:9], [10:10]\}$ son sus aristas.
- Y sus mapas pueden defirse por medio de funciones por partes

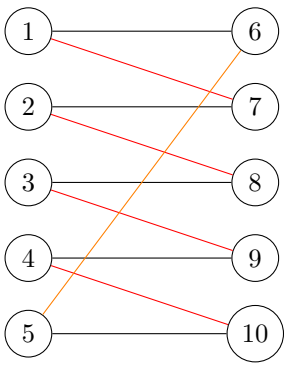
$$map_1(e) = \begin{cases} e & \text{si } e \in [1:5] \\ e-5 & \text{si } e \in [6:9] \\ e-5 & \text{si } e \in [10:10] \end{cases}$$

$$map_2(e) = \begin{cases} e+5 & \text{si } e \in [1:5] \\ e+1 & \text{si } e \in [6:9] \\ e & \text{si } e \in [10:10] \end{cases}$$

Tener en cuenta que, en cuanto a los mapas, al tratarse de funciones, podemos obtener su imagen, que estaría contenida en $\mathcal{V}$ y su preimagen, contenida en $\mathcal{E}$.

En la descripción gráfico del grafo G, o su equivalente $\mathcal{G}$, definido previamente:

- las aristas correspondientes al intervalo $[1:5]$ están en color negro.
- las correspondientes al intervalo $[6:9]$ en rojo.
- y las del intervalo $[10:10]$ en color naranja.



En este ejemplo, el mapa map_1, en su definición para el dominio $\{[1:5]\}$, nos está indicando las conexiones en una de las direcciones de las aristas. Y el mapa map_2, en su definición para el mismo dominio, las conexiones en la otra dirección. Por lo tanto, la arista 1 conecta a los nodos $\{map_1(1), map_2(1)\} = \{1, 6\}$. Siguiendo esta misma idea se puede entender esta representación compacta de grafos.

Capítulo 6

Algoritmo de Balance de Carga para Sistemas de Eventos Discretos basado en Grafos SBG

En este capítulo, describiremos en detalle tanto los algoritmos implementados como así también las estructuras de datos utilizadas para la construcción del grafo computacional utilizado como entrada de los mismos.

6.1. Grafo Computacional SBG

Este trabajo se basa en la idea de implementar un algoritmo, basado en [11], pero usando grafos basados en conjuntos en lugar de su representación tradicional. Para esto, necesitamos representar grafos computacionales, como los explicados en la Sección 4.1.2, pero usando grafos basados en conjuntos, presentados en el Capítulo 5. La idea es compactar los nodos en intervalos, y representar las conexiones usando mapas como fue mostrado en la Sección 5.2. Los pesos de los nodos se representan con funciones constantes $w : \mathcal{V} \to R^+$ donde $\mathcal{V}$ es un conjunto de nodos, análogo para las aristas.

Entonces, nuestro grafo computacional puede definirse de la siguiente forma, sea $\mathcal{G} = (\mathcal{V}, \mathcal{E}, l, w)$:

- $\mathcal{V} = \{V^1, \ldots, V^n\}$ es un conjunto de vértices-conjunto disjuntos.
- $\mathcal{E} = E^1, \ldots, E^m$ es un conjunto de aristas-conjunto que une vértices-conjunto de $\mathcal{V}$.
- $l : \mathcal{V} \to \mathcal{R}^+$ asigna un peso computacional a cada nodo.
- $w : \mathcal{E} \to \mathcal{R}^+$ asigna un peso comunicacional a cada arista.

Dado que la aplicación principal del algoritmo aquí presentado es obtener una partición de un modelo Modelica dado para luego poder simularlo en paralelo,

a continuación presentamos las estructuras de datos utilizadas para representar información estructural que se puede obtener facilmente a partir de un modelo definido en ese lenguaje.

6.1.1. Estructura de Datos de Entrada

Si definimos la ocurrencia de una variable en una expresión dada como:

Algorithm 2 Tipo Var

Var: {
id : string
cost : int
exp : (a, b) donde $a, b \in N^0$ y estos representan constante y pendiente en la expresión $a \times i + b$
def : $\{n_1, \ldots, n_k\}$ donde n_i es el id de un conjunto de nodos donde se calcula la variable. Es decir, aparece en el lado izquierdo de la ecuación.
}

Luego, un nodo individual que representa una unidad de cálculo en el modelo puede ser definido así:

Algorithm 3 Tipo Node

Node: {
id: string
weight: int
interval: $[1, \ldots, M]$
lhs: $\{L_1, \ldots, L_k\}$ donde L_i tiene tipo **Var**
rhs : $\{R_1, \ldots, R_g\}$ donde R_i tiene tipo **Var**
}

Y finalmente podemos definir: Nodes : $\{I_1, \ldots, I_m\}$ con I_j del tipo Node como entrada del algoritmo de particionado.

6.1.2. Ejemplo: Modelo Advección–Difusión–Reacción

El siguiente modelo, presentado en [3] representa una ecuación de Advección-Difusión–Reacción (ADR) en una dimensión. Esta ecuación puede describir, por ejemplo, un río transportando una substancia que experimenta una reacción química.

Las Ecuaciones 6.1 6.2 se obtienen luego de aplicar el Método de Líneas a las ecuaciones en Derivadas Parciales (PDE) originales.

$$\dot{u_1} = -a\frac{(u_N - 1)}{\Delta x} + d\frac{-2u_1}{\Delta x^2} + r(u_1^2 - u_1^3) \tag{6.1}$$

y

$$\dot{u_i} = -a\frac{(u_i - u_{i-1})}{\Delta x} + d\frac{(u_{i+1} - 2u_i + u_{i-1})}{\Delta x^2} + r(u_i^2 - u_i^3) \tag{6.2}$$

para $i = 2, \cdots, N$

A partir de estas ecuaciones se puede definir el modelo Modelica descripto en la Sección 6.1.2

```
1 model adv_dif_reac
2   constant Integer N=1000;
3   parameter Real a=1;
4   parameter Real d=1e-4;
5   parameter Real r=10;
6   parameter Real L=10;
7   parameter Real dx=L/N;
8   Real u[N];
9
10 initial algorithm
11   for i in 1:N/3 loop
12     u[i]:=1;
13   end for;
14
15 equation
16   der(u[1])=-a*(u[1]-1)/dx+d*(-2*u[1]+1)/(dx^2)+r*(u
       [1]^2)*(1-u[1]);
17   for i in 2:N loop
18     der(u[i])=-a*(u[i]-u[i-1])/dx+ d*(-2*u[i]+u[i-1])
         /(dx^2)+ r*(u[i]^2)*(1-u[i]);
19   end for;
20 end adv_dif_reac;
```

En las siguientes secciones utilizaremos este modelo para dar un ejemplo detallado de cómo se construyen las estructuras de datos de entrada del algoritmo.

Representación de las conexiones usando el tipo Node

El primer `for` en la línea 11 sólo inicializa variables, luego en la línea 11 se define la variable `u[1]`, usando el tipo `Var` podemos representarlo de la siguiente manera:

Lado izquierdo de la expresión en línea 16:

$$
\begin{aligned}
V_1 = \{ & \\
& id = u \\
& cost = 1 \\
& exp = (0, 1) \rightarrow \text{para formar la expresión} \\
& 0 \times +1 \text{ que corresponde a } u[1] \\
& defs = \{\} \\
\}&
\end{aligned}
$$

Lado derecho de la expresión en línea 16:

$$
\begin{aligned}
V_2 = \{ & \\
& id : u \\
& cost = 1 \\
& exp = (0, 1) \rightarrow \text{para formar la expresión} \\
& 0 \times +1 \text{ que corresponde a } u[1] \\
& defs = \{1, 2\} \\
\} &
\end{aligned}
\tag{6.3}
$$

Para la expresión presentada la Ecuación 6.3, `u[1]` aparece varias veces en la expresión, pero es suficiente con considerar una.

Luego, de la línea 17 a la 19 se define un bucle `for`, el cual podemos representar con la siguientes expresiones del tipo `Var`.

Lado izquierdo de la expresión en línea 18:

$$
\begin{aligned}
V_3 = \{ & \\
& id : u \\
& cost = 1 \\
& exp = (1, 0) \rightarrow \text{para formar la expresión} \\
& 1 \times +0 \text{ que corresponde a } u[j] \\
& defs = \{\} \\
\} &
\end{aligned}
$$

Lado derecho de la expresión en línea 18:

$$
\begin{aligned}
V_4 = \{ & \\
& id : u \\
& cost = 1 \\
& exp = (1, 0) \rightarrow \text{para formar la expresión} \\
& 1 \times +0 \text{ que corresponde a } u[j] \\
& defs = \{1, 2\} \\
\} & \\
V_5 = \{ & \\
& id : u \\
& cost = 1 \\
& exp = (1, -1) \rightarrow \text{para formar la expresión} \\
& 1 \times -1 \text{ que corresponde a } u[j-1] \\
& defs = \{1, 2\} \\
\} &
\end{aligned}
$$

El elemento `defs` de las expresiones de tipo `Var` correspondientes a los lados izquierdos son conjuntos vacíos, esto sucede porque buscamos ocurrencias de

las variables a partir de los lados derechos, esto se explicará en detalle en la Sección 6.1.3.

Usando esas expresiones podemos definir estas expresiones del tipo `Node`.

Expresión de tipo Node correspondiente a la línea 11 del modelo:

$$
\begin{aligned}
N_1 = \{ & \\
& id : 1 \\
& weight = 1 \\
& interval = [1, 1] \\
& lhs = \{V_1\} \\
& rhs = \{V_2\} \\
\} &
\end{aligned}
$$

[H]

Expresión de tipo Node correspondiente a la línea 13 del modelo:

$$
\begin{aligned}
N_2 = \{ & \\
& id : 2 \\
& weight = 1 \\
& interval = [2, 1000] \\
& lhs = \{V_3\} \\
& rhs = \{V_4, V_5\} \\
\} &
\end{aligned}
$$

El elemento del tipo `Nodes` quedaría definido N : `Node` $= \{N_1, N_2\}$, siendo la entrada para el algoritmo presentado en esta sección que representa al modelo ADR presentado en la Sección 6.1.2.

6.1.3. Generación del Grafo

Los nodos son representados como números naturales incluyendo al cero. Para distinguirlos apropiadamente, debemos evitar solapamiento en los intervalos de los nodos. Por lo tanto, debemos sumar una compensación entre los nodos. Por ejemplo, para el modelo ADR, el conjunto de nodos sería de la forma $\mathcal{V} = \{[0 : 0], [1 : 999]\};$, donde el primer intervalo representa al nodo N_1 y el segundo al nodo N_2.

Con la estructura de datos presentada en la Sección 6.1.1, el algoritmo que construye las aristas del grafo computacional puede definirse así:

Algorithm 4 Construír grafo computacional SBG

1: **function** BUILD_SBG_GRAPH($G, P = (A, B), C$)
2: **for** $I_j \in$ Nodes **do**
3: **for** $R_i \in I.rhs_j$ **do**
4: $IM(R_i) \leftarrow$ Construye el conjunto imagen para la expresión de R_i
5: **for** $D_k \in R.defs_i$ **do**
6: $I_s \leftarrow$ El Nodo con id D_k en $Nodes$
7: $L_w \leftarrow$ la variable del lado izquierdo que coincide con R_i en el conjunto LHS de entrada.
8: $IM(L_w) \leftarrow$ Calcula el conjunto imagen para la expresión de la variable L_w.
9: $D \leftarrow IM(R_i) \cap IM(L_w)$
10: **if** $D \neq \emptyset$ **then** Esto significa que el nodo I_s computa a la variable (o parte de la variable) L_w que se debería comunicar a I_j. En este caso:
11: Armar una conjunto arista entre (I_j, I_s) con cardinalidad $\#D$
12: **end if**
13: **end for**
14: **end for**
15: **end for**
16: **end function**

En la línea 4 del Algoritmo 3, lo que hacemos es calcular el uso de la variable en ese nodo. Por ejemplo, si observamos el elemento V_4 del lado derecho del nodo N_2, el uso de la variable u es igual a la imagen de la expresión $1 \times x + 0$ con dominio $\{[2 : 1000]\}$. Por lo tanto, la variable u se usa en el intervalo $\{[1 : 1000]\}$. En los pasos siguientes iteramos sobre los nodos en los cuales u está definida. Luego, en la línea 10 chequeamos si la intersección de las imágenes no es vacía, es decir, si u fue definida en otro nodo en el intervalo. Supongamos que encontramos un nodo N_x donde u aparece del lado izquierdo y accede a los elementos $\{[1 : 1000]\}$, o un subconjunto de los mismos. En ese caso, al cambiar un valor de u en uno de estos dos nodos, debe informar los cambios al otro. Por lo tanto, existe una necesidad de sincronización entre los mismos, lo que representamos con aristas en el grafo computacional.

Ejemplo de Ejecución del Algoritmo de Generación de grafos SBG

Para explicar más en detalle este algoritmo, tomaremos como ejemplo la ejecución del mismo con la representación del modelo ADR definida en la Sección 6.1.2.

El primer elemento que tomaría el tipo definido en el Algoritmo 3 sería el elemento N_1 definido en la Expresión 6.1.2.

Lo primero que buscaremos son las *conexiones* de este nodo. Procedemos a iterar los elementos del lado derecho *rhs* que contienen un solo elemento, V_2 definido en la Ecuación 6.3. Lo que pretendemos saber en este paso es el *uso* de esta variable, es decir, que porción de la variable u es calculada en esta expresión. Tomando el intervalo que involucra a este nodo como dominio, que en este caso es $[1 : 1]$, un solo elemento, puesto que no estamos dentro de un bucle,

calculamos la imagen de la expresión $0 \times i + 1$ que es $[1:1]$. A continuación, buscamos los nodos donde la variable u está definida, es decir, aparece del lado izquierdo de la ecuación. Está definida en ambos nodos, primero en el nodo uno, calculamos imagen usando el mismo dominio, obtenemos el conjunto $[1:1]$. Para saber si esta variable se computa en el nodo, chequeamos si la intersección de las imágenes no es vacía. Si no lo es, la variable u para los valores $[1:1]$ se define en el nodo N_1 y se usa en el nodo N_1. Este caso no aporta información, puesto que es la variable comunicándose consigo misma y no es útil en cuanto a como balancear la carga. Es una limitación de este algoritmo que resolvemos a nivel de implementación, ignorándolos.

Para la siguiente definición, N_2, tiene un solo elemento del lado izquierdo, V_3, y las imágenes tienen intersección vacía.

Pasemos al siguiente nodo, N_2. Primero calculamos la imagen del primer elemento del lado derecho que es V_4, $IM(V_4) = [2:1000]$

Tomamos el primero de los lugares en dónde están definidas las variables del nodo, el N_1. Calculamos la imagen de la única variable del lado izquierdo usando el dominio del nodo N_1, el resultado es $[1:1]$ y la intersección es vacía.

Tomamos luego el lado izquierdo de N_2, calculamos la imagen del lado izquierdo y la intersección de las imágenes nos da $[2:1000]$, pero la ignoramos puesto que la conexión es reflexiva. Por otro lado, para el elemento V_5 la imagen es $[1:999]$ y tiene intersección no vacía con el elemento V_1 presente en el lado izquierdo de N_1, la intersección es $[1:1]$. Para este caso, crearemos un conjunto de conexiones de la misma cardinalidad de la imagen. V_5 también tiene intersección no vacía con el lado izquierdo de N_2, V_3, cuya imagen es $[2:1000]$ su intersección es $D = [2:999]$.

Ahora nos proponemos crear un conjunto de aristas por medio de dos mapas con el mismo dominio, como fue explicado en la Sección 5.1, que conecte las variables de los lados izquierdos de los nodos.

Construir mapas de conjunto de aristas

Primero creamos un conjunto de aristas de cardinalidad $\#D$. luego, Si el factor de la expresión es cero, es decir, el nodo es constante, una expresión con esa constante. De lo contrario, debemos calcular un nuevo mapa lineal de la forma: $Var.exp - E_{offset} + M_{offset} + D_{offset} + V_{offset}$

Donde $Var.exp$ es la expresión del elemento de tipo `Var` del lado izquierdo del nodo (`lhs`), E_{offset} es la compensación del conjunto de aristas, es decir, el mínimo elemento del conjunto de aristas, V_{offset} es la compensación del conjunto de nodos, es decir, el valor que le fue sumado (o restado) al intervalo de nodos para evitar solapamientos y M_{offset} es la compensación que se obtiene así:

- Calcular la preimagen (antes llamada D) del mapa usado para verificar la existencia de conexiones.
- Calcular la imagen del lado izquierdo de la expresión usando como dominio el conjunto obtenido en el paso anterior.
- Buscar el mínimo elemento del conjunto imagen obtenido.

Acorde a la ejecución del algoritmo sobre el modelo advección-reacción, debemos crear dos conjuntos de aristas, uno que conecte N_1 con N_2 y otro que conecte N_2 con N_2.

Para el primer caso, el conjunto de aristas tienen cardinalidad 1. La expresión del lado izquierdo del nodo N_1 es de la forma $0 \times x + 1$, lo que llamamos *expresión constante*, el mapa será de la forma $[N : N] \mapsto m_1 1$. La expresión del lado izquierdo de N_2 es de la forma $x + 0$, por lo tanto el mapa tendrá la forma $x - E_{offset} + V_{offset} + M_{offset}$. Sea 1000 el elemento mínimo del conjunto de aristas y -1 la compensación del intervalo del nodo. Luego, para calcular M_offset:

- Calculamos la preimagen de $V_5.exp$ para el conjunto $\{[1 : 1]\}$, obtenemos $\{[2 : 2]\}$.
- Ahora calculamos la imagen de $V_4.exp$ para $\{[2 : 2]\}$, que es $\{[2 : 2]\}$.
- El mínimo $\{[2 : 2]\}$ es 2.

Finalmente, obtenemos una expresión de la forma $x - 1000 - 1 + 2 = x - 999$.

Lo mismo debe hacerse para las conexiones de N_2 con N_2, que conecta las variables $u[2 : 999]$ con $u[3 : 1000]$.

El grafo obtenido es un Grafo Computacional SBG como el que definimos en la Sección 6.1:

```
V = {[0:0], [1:999]};
E = {[1000:1000], [1001:1998]}
map1 = <<{[1000:1000]} -> 0,{[1001:1998]} -> x-1000>>
map2 = <<{[1000:1000]} -> x-999,{[1001:1998]} -> x
   -999>>
node weight = << {[0:0]} -> 1,{[1:999]} -> 1 >>
edge costs = << {[1000:1000]} -> 1,{[1001:1998]} -> 1
    >>
```

El grafo aquí presentado incluye los elementos `node weight` y `edge costs` que representan el peso de los nodos y el costo de las aristas respectivamente.

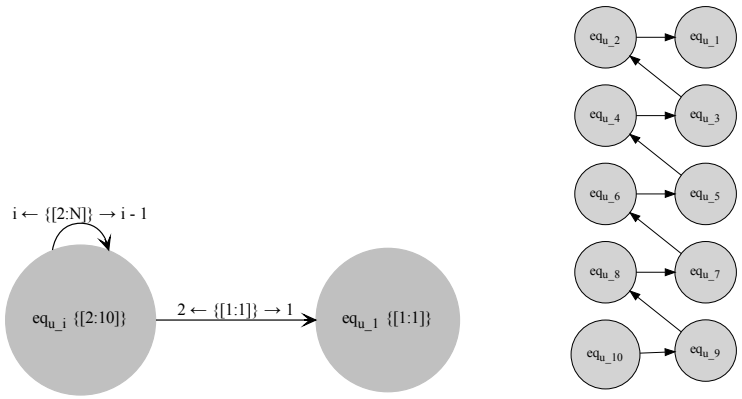
A modo de ejemplo, en la Figura 6.1a podemos ver el grafo computacional SBG generado para un modelo ADR con $N = 10$, asimismo en la Figura 6.1b se muestra el grafo expandido equivalente que es utilizado como entrada en algoritmos de particionado tradicionales como METIS.

6.1.4. Partición Inicial

Como explicamos en la Sección 4.2, necesitamos una partición inicial para nuestro algoritmo. Hemos implementado dos estrategias, una estrategia distributiva y una estrategia greedy.

Recorrer un Grafo Basado en Conjuntos

Para recorrer un grafo basado en conjuntos $\mathcal{G}$ tomamos uno de los intervalos del conjuto de nodos $\mathcal{V}$ y buscamos con qué otros nodos está conectado. Para esto tomamos el conjunto de mapas $map1$ y calculamos su preimagen, tomando al intervalo elegido como dominio. El conjunto preimagen obtenido se lo pasamos



(a) Grafo computacional SBG para $N = 10$

(b) Grafo computacional expandido correspondiente

Figura 6.1: Descripción gráfica de un grafo compacto y su equivalente expandido.

como entrada a $map2$ y computamos la imagen. De esta manera obtenemos los nodos que se conectan con el intervalo seleccionado, partiendo de $map1$. Repetimos el proceso invirtiendo los mapas. Podemos continuar recorriendo con los nodos obtenidos.

Estrategia Greedy

Sea N la cantidad total de nodos (supongamos que todos los nodos tienen peso 1 y todas las aristas tienen costo de comunicación 1), sea P el número deseado de particiones, entonces es $m = N/P$ el número deseado de nodos (en caso de haber resto, se reparte entre tantas particiones como se pueda). Luego, a medida que se recorre el grafo, se agregarán todos los elementos a una partición hasta que esté completa, luego se continuará con la siguiente, hasta haber recorrido todos los nodos.

Estrategia Distributiva

Sea N la cantidad total de nodos (supongamos que todos los nodos tienen peso 1 y todas las aristas tienen costo de comunicación 1), sea P la cantidad de particiones deseada, N/P lo deseado para cada una, el resto se resuelve de la misma manera que en la estrategia Greedy. Mientras se corre el grafo, cada intervalo se parte en P partes iguales y se reparten una parte para cada partición. Si hay un restante, se reparte entre las particiones con menos elementos.

6.2. Algoritmo de Balance de Carga usando Grafos Computacionales Basados en Conjuntos (KL-SBG)

En esta sección explicaremos el proceso de balance y optimización de carga para la simulación en paralelo de modelos QSS usando grafos computacionales basados en conjuntos. El mismo es una adaptación del algoritmo presentado en [11], pero durante la etapa de optimización usaremos conjuntos de intervalos de nodos en lugar de intercambiar nodos individuales. Aquí hacemos uso de la estructura cíclica de los modelos representados en Modelica, buscando acelerar este proceso y obtener mejores particiones (con menor comunicación externa).

6.2.1. Definiciones Básicas

Dado un grafo computacional $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ y sea C su matriz de costo asociada. Supondremos que tanto el peso de los nodos como el costo de comunicación es uno para todos los casos, por lo que podemos ignorar las funciones de costo y peso y obtener idénticos resultados. Sea (A, B) una partición inicial de $\mathcal{V}$, definimos *costo externo* como:

$$\forall a \in A \rightarrow E_a = \sum_{y \in B} c_{ay} \tag{6.4}$$

y *costo interno*:

$$\forall a \in A \rightarrow I_a = \sum_{x \in A} c_{ax} \tag{6.5}$$

entonces ahora podemos definir:

$$\forall a \in A \rightarrow {}_a = E_a - I_a \tag{6.6}$$

6.2.2. Optimización de Particiones

Esta etapa del algoritmo recibe dos particiones e intenta intercambios de intervalos de igual tamaño, buscando obtener una mejora en la partición, es decir, una reducción en la *comunicación externa*.

Algorithm 5 Algoritmo de optimización Kernighan-Lin para SBG

function KL-SBG($G, P = (A, B), C$)
  $A_c = A$ ▷ Hacer una copia de las particiones originales
  $B_c = B$
  $max\_par\_sum = 0$ ▷ Inicializar variables
  $max\_par\_sum\_set = \emptyset$
  $par\_sum = 0$
  $A_v = \emptyset$
  $B_v = \emptyset$
  $GM =$ **compute_diff**(A_c, B_c, G, C) ▷ Computar matriz de ganancia
  while $A_c \neq \emptyset \wedge B_c \neq \emptyset$ **do** ▷ Mientras nos queden elementos en alguna de las particiones, seguir iterando
    $< i, j, g, s >=$ **max_diff**(GM) ▷ Obtener intercambio con mayor ganancia
    $(A', B') =$ **update_sets**$(A_c, B_c, A_v, B_v, i, j, s)$ ▷ Actualizar conjuntos acorde a los cambios propuestos
    update_sum$(par\_sum, g, max\_par\_sum, max\_par\_sum\_set, A', B')$ ▷ Actualizar suma parcial
    update_diff$(A_c, B_c, i, j, s, GM, G)$ ▷ Actualizar las copias A_c y B_c
  end while
  if $max\_par\_sum\_set \neq \emptyset$ **then** ▷ Si existió alguna ganancia positiva, efectuar el cambio
    $(A^\star, B^\star) = max\_par\_sum\_set$
    $A = (A/A^\star) \cup B^\star$
    $B = (B/B^\star) \cup A^\star$
  end if
  return
end function

6.2.3. Algoritmo KL-SBG para dos particiones

Esta etapa del algoritmo toma dos particiones y las mejora hasta la obtención de un mínimo local, es decir, hasta que la ganancia no sea mayor a cero.

Algorithm 6 KL-SBG para dos particiones

function KL-SBG-BIPART($G, P = (A, B), C$)
  $A' = A$
  $B' = B$
  $\mathbf{KL - SBG}(G, (A, B), C)$
  while $A' \neq A \wedge B' \neq B$ **do**
    $A' = A$
    $B' = B$
    $\mathbf{KL - SBG}(G, (A, B), C)$
  end while
  return
end function

6.2.4. Funciones Auxiliares de KL-SBG

Aquí presentaremos y explicaremos algunas funciones con propósitos específicos

Actualización de conjuntos

A medida que el Algoritmo 5 avanza, las variables involucradas deben ir actualizándose, tanto para guardar los cambios ante la posibilidad de una mejora, como para cumplir con la condición de terminación, que es cuando todas las posibilidades de intercambio se agotaron, al menos para una de las particiones. A y B representa los elementos aún no intercambiados, A_v y B_v los elementos que ya fueron intercambiados, s es el tamaño de los intervalos.

Algorithm 7 Actualización de conjuntos

1: **function** UPDATE_SETS(A, B, A_v, B_v, i, j, s)
2: $A' = \{a_k \in A_i | \#a_k \leq s\}$
3: $B' = \{b_k \in B_j | \#a_k \leq s\}$
4: $A_v = A_v \cup A'$
5: $B_v = B_v \cup B'$
6: $A_c = A_c / A'$
7: $B_c = B_c / B'$
8: **return** (A', B')
9: **end function**

Cálculo de Costo Externo y Costo Interno

Esta es, quizás, una de las partes más interesantes de este trabajo, puesto que en base a los resultados de esta función se toman decisiones en la fase de optimización, es nuestra adaptación para la Ecuación 4.6.

Los argumentos de esta función son S, que es el intervalo o conjunto de intervalos sobre el cual queremos calcular costo externo e interno, P que es la partición a la cual pertenece actualmente S, Q es la otra partición, map_1 y map_2 son los mapas del grafo. Aquí lo que se busca son las aristas en la cuales S está involucrada, para luego determinar si las mismas pertenecen a comunicación con nodos en la misma partición (comunicación interna) o nodos de la partición Q.

Algorithm 8 Computer EC & IC

1: **function** COMPUTE_EC_IC(S, P, map_1, map_2)
2: $D = map_1.preImage(S)$
3: $I = map_2.image(D)$
4: $IC' = (P \cap I)/S$
5: $IC = map_2.preImage(IC') \cap D$
6: $EC' = (I/IC')$
7: $EC = map_2.preImage(EC') \cap D$
8: **return** (EC, IC)
9: **end function**

Si quisiéramos usar `KL-SBG-BIPART` para múltiples particiones, tomando de a dos, la versión de `compute_ec_ic` no nos serviría, pues al calcular `EC`, S podría comunicarse con nodos que estén fuera de la partición con la estamos lidiando, por eso implementamos esta segunda versión:

Algorithm 9 Computer EC & IC

1: **function** COMPUTE_EC_IC(S, P, Q, map_1, map_2)
2: $D = map_1.preImage(S)$
3: $I = map_2.image(D)$
4: $IC' = (P \cap I)/S$
5: $IC = map_2.preImage(IC') \cap D$
6: $EC' = (I/IC') \cap Q$
7: $EC = map_2.preImage(EC') \cap D$
8: **return** (EC, IC)
9: **end function**

Detalles de Implementación

En esta versión, en la línea 6 se hace una intersección con la partición Q. Esto no tiene sentido si pensamos que trabajamos con dos particiones, pero sí lo tiene si trabajamos con más, puesto que garantizamos que la comunicación externa sólo involucra a la partición Q y no a las demás.

En la línea 4 se hace la intersección de los nodos con los que se comunica I y luego la diferencia con S, porque pretendemos calcular la comunicación interna que se convertirá en externa en caso de mover S a Q, si los elementos de S se comunican entre sí, no aportarán comunicación externa al moverlos a Q.

En las líneas 5 y 7 se intersectan las aristas con D, calculadas al comienzo, esto es para prevenir contar aristas que no correspondan, quedarnos solo con las aristas con las que S está involucarado.

Esta función solo computa los costos en una dirección, es decir, tomando map_1 como salida y map_2 como llegada. Para un cálculo de los costos correcto, debe llamarse dos veces, una con map_1 como salida y map_2 como llegada y viceversa, y los costos se obtienen haciendo la unión de los resultados obtenidos.

Cálculo de la Ganancia

Ya habiendo explicado como obtener los costos externos e internos, ahora presentamos como obtener la ganancia a partir del intercambio de dos intervalos o conjuntos de intervalos de una partición a la otra.

Algorithm 11 Cálculo del elemento *diff*

1: **function** COMPUTE_DIFF(A, B, G, C)
2: $D_A =$ **compute_diff**(A, G)
3: $D_B =$ **compute_diff**(B, G)
4: $GM =$ **compute_gain_matrix**(D_A, D_B, C)
5: **return** GM
6: **end function**

Algorithm 10 Calcular Costos

```
1: function COMPUTE_COSTS(P = {P_1, ··· , P_k}, G)
2:     D =<>
3:     for P_i ∈ P do
4:         (EC_i, IC_i) = compute_EC_IC(P, P_i, G.map_1(), G.map_2())
5:         (EC_i, IC_i) = (EC_i, IC_i)∪compute_EC_IC(P, P_i, G.map_2(), G.map_1())
6:         D.push_back((EC_i, IC_i))
7:     end for
8:     return D
9: end function
```

Algorithm 12 Cálculo de la Matriz de Ganancia

```
1: function COMPUTE_GAIN_MATRIX(D_A, D_B, C)
2:     G =<>
3:     for d_i ∈ D_A do
4:         gain_{d_i} =<>
5:         for d_j ∈ D_B do
6:             s = min(#d_i, #d_j)
7:             g = diff(d_i, s) + diff(d_j, s) − 2 × c_{a,b}
8:             gain_{d_i}.insert(i, j, g, s)
9:         end for
10:        G.insert(gain_{d_i})
11:    end for
12:    return
13: end function
```

Aquí **diff** refiere a la tupla $d_i = (EC_i, IC_i)$ restringida al máximo número de elementos posibles s.

En la línea 7 calculamos nuestra adaptación de la ecuación 4.7 para calcular la ganancia del intercambio de los dos intervalos, restando la comunicación entre los mismos para evitar redundancia.

La **Matriz de Ganancia** nos provee una forma práctica de representar información, donde un elemento d_{ij} nos provee la ganancia de intercambiar el elemento $a_i \in A$ con el elemento $b_j \in B$.

Algorithm 13 Obtener intercambio con mayor ganancia

```
1: function MAX_DIFF(GM)
2:     gain = GM.pop()
3:     g_max =< i, j, g, s >= gain.pop()
4:     gain.update(s)
5:     GM.insert(gain)
6:     return g_max
7: end function
```

Aquí buscamos el elemento de la matriz que representa el intercambio que mayor ganancia provee. El elemento s nos indica el mínimo de la cardinalidad de ambos intervalos. Por ejemplo, si pretendemos intercambiar $a_i \in A$ con $b_i \in B$,

donde $\#a_i = 50$ y $\#b_i = 100$, intercambiarlos completos provocaría un desbalance de cincuenta nodos de más en A. Para evitar que eso suceda, tomamos la cardinalidad mínima, en este caso tomaríamos los primeros cincuenta elementos de b_i, los demás permanecerían en B.

Algorithm 14 Actualizar diff

```
1: function UPDATE_DIFF(P, D_A, D_B, i, j, s, GM, G, lookup)
2:     I_upd = {P_1, ··· , p_n} = image(EC'_j, G) ∩ P
3:     for p_k ∈ I_upd : do
4:         if lookup = row then
5:             GM.update_row(k, j, s)
6:         else
7:             GM.update_row(j, k, s)
8:         end if
9:     end for
10:    E_upd = {P_1, ··· , p_n} = image(IC'_i, G) ∩ P
11:    for p_k ∈ I_upd : do
12:        if lookup = row then
13:            GM.update_row(k, j, s)
14:        else
15:            GM.update_row(j, k, s)
16:        end if
17:    end for
18:    return
19: end function
```

La función `update_diff` actualiza las filas y columnas involucradas en el intercambio.

Algorithm 15 Actualizar Matriz de Ganancia

```
1: function UPDATE_GAIN_MATRIX(A_c, B_c, D_A, D_B, i, j, s, GM, G)
2:     update_diff(A_c, D_A, D_B, i, j, s, GM, G, row)
3:     update_diff(B_c, D_A, D_B, j, i, s, GM, G, column)
4:     return
5: end function
```

Algorithm 16 Actualizar suma parcial

```
1: function UPDATE_SUM(par_sum, g, max_par_sum, max_par_sum_set, A', B')
2:     par_sum = par_sum + g
3:     if par_sum > max_par_sum then
4:         max_par_sum = par_sum
5:         max_par_sum_set = max_par_sum_set ∪ (A', B')
6:     end if
7:     return
8: end function
```

La suma parcial representa la suma actual contanto todos los intercambios realizados hasta el momento. Las variables `max_par_sum` y `max_par_sum_set`

representan la ganancia y los conjuntos a ser intercambiados para obtenerla, respectivamente.

Capítulo 7

Implementacion

En este capítulo se explican detalles y decisiones tomadas durante la implementación del algortimo propuesto en el Capítulo 6. La implementación fue realizada desde cero, y se encuentra disponible en el siguiente repositorio de GitHub https://github.com/CIFASIS/sbg-partitioner.

El diseño e implementación de este proyecto se caracterizó por: primero desarrollar algoritmos en pseudocódigo, que han proporcionado una imagen de alto nivel a la hora de la implementación. Luego la implementación de los mismos. A la hora de escribir código, se cubren una serie de detalles que escapan a la especificación de alto nivel propuestas al comienzo, pero no sin dejar de respertarla. En el Capítulo 6 se exhiben los algoritmos en pseudocódigo mencionados, con explicaciones sobre la idea y el propósito de cada uno, la implementación respeta esas intenciones y resuelve una serie de detalles que escapan a ellos.

7.1. Lenguaje y Librerías de Terceros

La implementación fue realizada enteramente en C++ y usamos el estándar C++17. Fue desarrollado y probado en un entorno con sistema operativo Linux Ubuntu 20.04.6 LTS y fue pensado para correr en entornos con sistema operativo Linux. Si bien no hemos probado en una variedad de sistemas Linux, cualquier entorno de este tipo, con las dependencias debidamente instaladas, debería poder correr e integrar el particionador SBG sin mayores inconvenientes. La decisión de usar C++ como lenguaje se sustenta en la búsqueda de una ejecución rápida y con bajo uso de memoria, por lo que lenguajes interpretados no ofrecían características de rendimiento cautivantes, también en la experiencia y conocimiento de quienes la llevaron adelante la implementación, así como en compatibilidad con algunas librerías usadas y la integración con sistemas existentes.

Para la implementación se usaron una serie de librerías de terceros, todas de libre acceso.

- Set-Based Graph Library para grafos basados en conjuntos en su rama `sb-graph-dev` creada para este propósito.
- boost en su versión 1.8.4, principalmente por dependencia de la librería sbg.

- RapidJSON para leer archivos en formato JSON, principalmente la entrada y salida.
- Google Test para implementar casos de test.

Cabe destacar que, en paralelo a este trabajo, se implementaron una serie de mejoras a la librería SBG. Este trabajo sería actualizado para usar la última versión de la misma y se esperan obtener importantes mejoras en cuanto al tiempo de ejecución. También nos dará la posibilidad de lidiar con intervalos de más de una dimensión. Podríamos tener como parte de la entrada una expresión de tipo `Node` como la siguiente:

```
{
    "id": 1,
    "interval": [[1, 1],[1, 1]],
    "lhs": [
        {
            "id": "u",
            "exp": [[0, 1],[0, 1]],
            "defs": []
        }
    ],
    "rhs": [
        {
            "id": "u",
            "exp": [[0, 1],[0, 1]],
            "defs": [1, 2, 3, 4]
        }
    ]
}
```

7.2. Compilar y Correr sbg-partitioner

Este proyecto puede compilarse y generar un binario en la carpeta `bin`, binario con el nombre `sbg-partitioner`, o bien puede generar una librería, generada en la carpeta `lib` bajo el nombre `libsbg-partitioner.a`.

También existe la posibilidad de generar otros binarios con propósitos específicos, el binario `sbg-partitioner-metrics` particiona un modelo dado como entrada y devuelve las métricas de particionado detalladas en la sección 8.1.

Desde el directorio raíz del respositorio del proyecto, el mismo puede compilarse corriendo `make all` o simplemente `make`. Para correrlo, deben pasarse los siguientes argumentos:

```
sbg-partitioner uso

-f, --filename   Directorio al archivo de entrada
    , un archivo json
que representa el modelo que queremos particionar
    .
```

```
-p, --partitions Numero de particiones.
-h, --help       Muestra esta informacion y
   termina.
-v, --version    Muestra informacion de la
   version y termina.
-g               Directorio del archivo de salida
    (no obligatorio).
```

También se puede compilar corriendo `make sbg-partitioner-lib` para compilarlo como una librería. Corriendo `make test` se lanza la plataforma de test.

7.3. Detalles de Implementación

7.3.1. Formato de Entrada

El formato de entrada del particionador es en formato JSON, una lista de elementos `Node` con sus respectivos elementos. La entrada para un modelo como el presentado en la Implementación 6.1.2 sería la siguiente:

```
{
    "nodes": [
        {
            "id": 1,
            "interval": [[1, 1]],
            "lhs": [
                {
                    "id": "u",
                    "exp": [[0, 1]],
                    "defs": []
                }
            ],
            "rhs": [
                {
                    "id": "u",
                    "exp": [[0, 1]],
                    "defs": [1, 2]
                }
            ]
        },
        {
            "id": 2,
            "interval":[ [2, 100]],
            "lhs": [
                {
                    "id": "u",
                    "exp": [[1, 0]],
                    "defs": []
                }
            ],
```

```
31              "rhs": [
32                  {
33                      "id": "u",
34                      "exp": [[1, 0]],
35                      "defs": [1, 2]
36                  },
37                  {
38                      "id": "u",
39                      "exp": [[1, -1]],
40                      "defs": [1, 2]
41                  }
42              ]
43          }
44      ]
45  }
```

La entrada es consumida por nuestro particionador y traducida a una representación interna que refleja lo definido en la Sección 6.1.1:

```
1  struct Var {
2      string id;
3      vector<pair<INT, INT>> exps;
4      vector<int> defs;
5      unsigned cost = 1;
6  };
7
8  struct Node {
9      int id;
10     int weight;
11     vector<pair<int, int>> intervals;
12     vector<Var> rhs;
13     vector<Var> lhs;
14 };
```

7.3.2. Creación del Grafo Computacional Basado en Conjuntos

Una vez leída la entrada y traducida a una estructura intermedia, debemos construír el grafo computacional. Para representar el grafo computacional basado en conjuntos presentado en la Sección 6.1, extedemos el tipo CanonSBG para representar las funciones de peso de nodos y costo de aristas. Esta representación la llamamos `WeightedSBGraph`. La función `create_sb_graph` es la encargada de tomar la entrada en la estructura antes definida y construír el grafo computacional.

Tanto la lectura de la entrada como la creación del grafo se encuentran declarados e implementados en los archivos `src/build_sb_graph.hpp` y `src/build_sb_graph.cpp` respectivamente.

Los `CanonSBG` nos proveen grafos basados en conjuntos, con conjuntos de intervalos de una dimensión ordenados de menor a mayor. El orden es una ventaja, pues acelera los computos y nos permite referirnos a intervalos de nodos por medio de índices, y así serán tratados en este trabajo. Cuando se mencione el `Nodo 0` nos referimos al intervalo de nodos en la posición 0 de un determinado conjunto. Por ejemplo, veamos nuevamente el grafo computacional basado en conjuntos del modelo advección-reacción:

```
V = {[0:0], [1:999]};
E = {[1000:1000], [1001:1998]}
map1 = <<{[1000:1000]} -> 0, {[1001:1998]} -> x
   -1000>>
map2 = <<{[1000:1000]} -> x-999, {[1001:1998]} -> x
   -999>>
node weight = << {[0:0]} -> 1, {[1:999]} -> 1 >>
edge costs = << {[1000:1000]} -> 1, {[1001:1998]} ->
   1 >>
```

Si nos referimos al `Nodo 1` de `V`, nos referimos al intervalo $[1 : 999]$. También por cuestiones de implementación, las aristas no deben solaparse con los nodos, por eso se lleva un offset extra al calcularlas.

El solo poder lidiar con una dimensión ha sido una desventaja y algo a mejorar.

7.3.3. Partición Inicial

El particionador tiene la capacidad de generar cuatro particiones distintas, basándose en dos estrategias: greedy y distributiva. Si la macro `TRY_MULTIPLE_SRATEGIES` está definida y es distinta de cero, se crearán 4 particiones distintas, usando las dos estrategias y visitando el grafo preorden y postordenadamente. Si no lo está, se usará la estrategia distributiva y el grafo se visitará preordenadamente.

Las dos estrategias son una implementación de la interfaz PartitionStrategy. Esto nos da flexibilidad para implementar diferentes estrategias, así como para crear tantas particiones distintas como querramos. Para usar una estrategia, debemos crear un objeto de la misma y agregarlo llamando a la función

```
1 sbg_partitioner::search::add_strategy(
2     sbg_partitioner::PartitionStrategy &strategy,
3     bool pre_order)
```

como se visitarán los nodos.

Las estrategias indican el modo en que se dividen los nodos entre las particiones, pero existe una clase encargada de recorrer el grafo, DFS, que se encarga de recorrer los intervalos de los nodos siguiendo sus conexiones por medio del cómputo de los mapas del grafo.

Si se utilizó más de una estrategia, se tomará como partición inicial la que tenga menos comunicación entre particiones, es decir, la que tenga menor *edgeCut*.

Particiones inciales del Modelo advección-reacción

Para el modelo advección-reacción podemos obtener 4 particiones iniciales usando las dos estrategias mencionadas visitando los nodos preordenada y postordenadamente. El nodo inicial (con nodo nos referimos a intervalo de nodos) es el que más conexiones tenga.

```
// Distributiva preorden
0, {[1:250]}
1, {[251:500]}
2, {[501:750]}
3, {[0:0], [751:999]}

// Distributiva postorden
0, {[0:0], [1:249]}
1, {[250:499]}
2, {[500:749]}
3, {[750:999]}

// Greedy preorden
0, {[1:250]}
1, {[251:500]}
2, {[501:750]}
3, {[0:0], [751:999]}

// Greedy postorden
0, {[0:0], [1:249]}
1, {[250:499]}
2, {[500:749]}
3, {[750:999]}
```

7.3.4. Particionador KL-SBG para dos particiones

Una vez obtenidas la o las particinoes iniciales y seleccionada la mejor, esta se pasa como argumento al módulo encargado de optimizarlas.

La función `kl_sbg_bipart_imbalance` es la encargada de lanzar la optimización para dos particiones dadas, buscando encontrar un mínimo local. El término *imbalance* refiere a la posibilidad de desbalancear las particiones un porcentaje dado por el usuario. No ahondamos en esto, puesto que aún se encuentra en etapa experimental. Si el particionador se ejecuta como fue explicado en la Sección 7.2, la posibilidad de desbalance no será tenida en cuenta.

En la implementación, no definimos ninguna matriz C que represente los costos de comunicación entre los intervalos de nodos. Dado el hecho de que, en este particionador, no lidiamos con nodos individuales sino con intervalos y conjuntos de nodos, llevar una matriz de costos representa una complejidad extra.

Supongamos que tenemos una matriz de costo de las particines A y B. Dados los intervalo $a_i \in A$ y $b_j \in B$, con $a_i = [0 : 49]$ y $b_j = [200 : 299]$. El elemento c_{ij} debería tener el costo de comunicación entre a_i y $b'_j = [200 : 249]$, que serían los nodos intercambiados, en caso de que sean los que proevan mayor

ganancia. Más aún, supongamos que tras una interación, b_j es *partido*, y se conserva $b'_j = [220 : 299]$ en B. En ese caso debe recalcularse c_{ij} acorde a los nuevos cambios.

Si hemos definido una matriz de ganancia, `CostMatrixImbalance`, la cual es poblada con elementos `GainObjectImbalance` que contienen ubicación en la partición de los intervalos de nodos, así como las aristas de comunicación y el tamaño máximo en caso de un intercambio.

Supongamos la ejecución del particionador SBG con el modelo de advección-reacción, usando la partición inicial distributiva preorden:

```
0: {[1:250]}
1: {[251:500]}
2: {[501:750]}
3: {[0:0], [751:999]}
```

Buscamos optimizar las particiones 2 y 3:

```
{[501:750]}, {[0:0], [751:999]}
```

La matriz de ganancia sería la siguiente:

```
< Node: 0, size: 249; Node: 1, size: 249, gain: 0 >
< Node: 0, size: 1; Node: 0, size: 1, gain: -1 >
```

El primer elemento indica que el intercambio de los intervalos de nodos `[501:750]` y `[751:999]` tienen ganancia 0 y pueden intercambiarse 249 de cada uno para mantener el balance, puesto que el intervalos de nodos 0 de la partición 2 tiene doscientos cincuenta elementos, mientras que el nodo 1 de la partición 3 tiene 249.

El segundo elemento indica que la ganancia de intercambiarlos es -1 y se podría mover un elemento de cada uno.

Veamos otro caso, quizás más interesante:

```
\\ particiones iniciales
0: {[51:74], [99:99]}
1: {[75:98], [100:100]}

\\matriz de ganancia:
< Node: 1, size: 1; Node: 0, size: 1, gain: 3 >
< Node: 1, size: 1; Node: 1, size: 1, gain: 1 >
< Node: 0, size: 1; Node: 1, size: 1, gain: 1 >
< Node: 0, size: 24; Node: 0, size: 24 gain: 1 >
```

El intercambio que mejor ganancia provee es el del intervalo 1 de la partición 0, el $[99 : 99]$ con el intervalo 0 de la partición 1, el $[75 : 98]$. Dado que tienen distinto tamaño, el intervalo $[75 : 98]$ será partido en dos intervalos, $[75 : 75]$ y $[76 : 98]$. Luego, el $[75 : 75]$ será *movido* a la partición 0 y el $[99 : 99]$ la partición 1. En el paso siguiente la matriz de ganancia es la siguiente:

```
\\particiones actuales
0: {[51:74]}
1: {[76:98], [100:100]}

< Node: 0, size: 23; Node: 0, size: 23, gain: -1 >
< Node: 0, size: 1; Node: 1, size: 1, gain: -1 >
```

La función encargada de actualizar la matriz de ganancia es `update_diff`, pero su implementación es ligeramente distinta de la descripta en la Sección 6.2.4. Esta función toma la matriz de ganancia así como información de los cambios realizados. Si alguno de los nodos fue usado por completo, borra todos los elementos asociados y acomoda los índices restantes. Luego, a cada uno de los nodos restantes:

- Si están en la misma partición, calcula la diferencia entre las aristas que pertenecían al nodo movido y las comunicación interna del intervalo. Con esto, la comunicación interna compartida entre estos es descartada.
- Si están en particiones distintas, calcula la diferencia de las aristas del nodo movido con la comunicación externa. Con esto, la comunicación externa compartida entre estos se descarta.

Es proceso se repite hasta que alguna de las particiones se vacía, es decir, todos los nodos fueron intercambiados, y se devuelve el intercambio de mayor ganancia.

7.3.5. Particionador KL-SBG para múltiples particiones

Es deseable que nuestro particionador funcione para múltiples particiones. Lo que hacemos es tomar particiones de a dos, optimizarlas como fue explicado en la Sección 7.3.4, y aplicar las modificaciones a los pares de particiones que tuvieron mejor ganancia. Para esto, tenemos dos estrategias distintas:

1. Aplicar el intercambio entre las dos particiones que mayor ganancia obtuvieron y reiniciar el proceso de optimización.
2. Aplicar todas las ganancias positivas entre particiones, sin repetir.

La estrategia número 2 es la que nos ha dado mejores resultados, pues converge a una solución más rápido para la mayoría de los casos. Supongamos que estamos particionando un modelo en 4 particiones, las llamaremos A, B, C y D. Tras ejecutar la optimización para dos particiones entre todas las combinaciones posibles, tenemos que la ganancia más alta es el par A, C. Ese cambio se efectúa, y se descartan todas las posibilidades de intercambio entre subconjuntos de A y de C. Luego, supongamos que existe una posibilidad de intercambio entre B y D con ganancia positiva, es efectuado y el proceso se reinicia. Esto se repite hasta tras intentar optimizar las particiones de a dos, ninguna retorna ganancia positiva.

7.3.6. Evitar Correr KL-SBG para particiones sin comunicación

Explicamos en la Sección 7.3.5 como se ejecuta KL-SBG para múltiples particiones. Cuando la número de particiones es alto, el cómputo para buscar una posible optimización es alto. Supongamos el caso de querer particionar un modelo en 24 particiones, pero que solo 4 puedan mejorar entre sí mediante una serie de intercambios. En un escenario ideal, sólo computaríamos KL-SBG entre aquellas particiones que mejorarán. Eso no podemos garantizarlo, pero si podemos chequear si, al menos, existe comunicación entre las particiones antes

de buscar una mejora. Una forma de hacerlo es buscar los nodos adyacentes de una partición dada, si la intersección con los nodos de otra partición es vacía, entonces no existe comunicación entre ellas y no vale la pena buscar ninguna mejora.

7.3.7. Multihilos en sbg-partitioner

Previo a correr KL-SBG para dos particiones, realizamos una copia de las particiones según su estado actual. Las mismas sólo serán modificadas en caso de que la ganancia sea positiva, y no haya otro intercambio que las involucre con ganancia más alta. Esto nos indica que no hay memoria compartida entre las ejecuciones de `kl_sbg_bipart`, aún cuando las particiones se repitan. Esto permite ejecutar estas funciones en paralelo con cierta facilidad. Mediante el uso de std::future y std::promise podemos lanzar múltiples llamados a `kl_sbg_bipart` en paralelo y guardar los resultados a medida que terminen su ejecución.

7.3.8. Peso en los Nodos y Costo en las Aristas

Hemos mencionado dos funciones, l y w, que asignan peso a los nodos y costo a las aristas respectivamente. Esto ha quedado en estado experimental, en los ejemplos mostrados tanto nodos como aristas tienen peso y costo igual a uno, respectivamente. En caso de que alguna de las aristas tengan costo distintos de 1, en la matriz de costos asociada C, en c_{ij} deben tener en cuenta no solo la cantidad de aristas involucradas, sino el costo de cada una. En el caso de los nodos, si buscamos una partición de los nodos de un grafo $\mathcal{G}$ en P particiones, cada partición debe tener N/P tal que $N = \sum v \in \mathcal{V} w(v)$. La misma precaución debe tenerse al mover intervalos de nodos de una partición a la otra.

Capítulo 8

Resultados

En esta sección presentamos una serie de ejemplos ordenados de menor a mayor por su complejidad y mediremos los resultados obtenidos comparándolos con otros algoritmos de balance de carga.

Para la comparación se utilizó una máquina con sistema operativo Linux (Ubuntu 20.04.6 LTS, desktop), procesador Intel® Core™ i98950HK CPU @ 2.90GHz × 12 y 32 GB de RAM.

A continuación se detallan las métricas utilizadas para comparar la calidad de los resultados.

8.1. Métricas de particionado

Las métricas de particionado evalúan la calidad de las particiones generadas. Las utilizadas para estos ejemplos son:

- **Edge-Cut**: La suma del peso de las aristas que conectan nodos entre particiones. Puede representarse como la siguiente función:

$$edgeCut(C, G) = \sum_{C_i, C_j} \{w(u, v) | u \in C_i, v \in C_j, i \neq j\}$$

.

- **Volumen de comunicación total**:

$$totCommVol(C, G) = \sum_{i=1}^{P} \sum_{u \in C_i} w(u, .)(\lambda(u, C) - 1)$$

donde $w(u, .)$ es el peso de una de las aristas de salida del nodo u (puesto que en estos ejemplos, todas tienen el mismo peso), mientras que $(\lambda(u, C) - 1)$ denota el número de particiones únicas a las que u o alguno de sus vecinos pertenece. Esta métrica también es conocida en al literatura como $(\lambda - 1)$.

- **Volumen de comunicación máximo**: Calcula el volumen de comunicación máximo entre particiones:

$$maxCommVol(C,G) = \max_{C_i} \sum_{u \in C_i} w(u,.)(\lambda(u,C) - 1).$$

- **Desbalance máximo**: Calcula el desbalance máximo entre particiones.

$$maxImbal(C,G) = \max_{C_i} \left| \frac{\sum_{u \in C_i}[l(u)] - (\sum_{v \in V} l(v))/P}{(\sum_{v \in V} l(v))/P} \right|$$

donde C es una partición obtenida, $G = (V, E, l, w)$ es un grafo de conexiones donde V son sus nodos, E sus aristas, l la función que asigna un peso computaciónal a cada nodo y w la función que asigna un peso computacional a cada arista.

8.2. Métricas de rendimiento

Las métricas de rendimiento buscan comparar tiempo de ejecución entre distintos algoritmos en distintas etapas.

- **Tiempo de Inicialización:** Es el tiempo que le toma al particionador en leer la entrada y construir el grafo de conexiones[1].
- **Tiempo de Particionado:** Es el tiempo que le toma al algoritmo generar las particiones iniciales y optimizarlas[1].

8.3. Ejemplos

8.3.1. Población de Aires Acondicionados

Este modelo, tomado de [13], estudia la dinámica de un conjunto de aires acondicionados (AAs) cuando siguen la misma temperatura de referencia. El i–ésimo AA es utilizado para controlar la temperatura de la i–ésima habitación, modelada por:

$$\dot{\theta}_i(t) = -\frac{1}{C_i \cdot R_i}[\theta_i(t) - \theta_a + R_i \cdot P_i \cdot m_i + w_i(t)] \tag{8.1}$$

donde R_i representa la resistencia térmica, C_i es la capacidad térmica, P_i representa el consumo de energía de la unidad de AA cuando está encendido, θ_a es la temperatura ambiente (común a todas las habitaciones), y $w_i(t)$ es una señal de ruido que representa perturbaciones térmicas.

La variable $m_i(t)$ representa el estado del i–ésimo AA que asume el valor 1 cuando está encendido y 0 cuando está apagado. Esta variable sigue la siguiente ley de control encendido–apagado con histéresis:

$$m_i(t^+) = \begin{cases} 0 & \text{if } \theta_i(t) \leq \theta_r^i(t) - 0{,}5 \text{ and } m_i(t) = 1 \\ 1 & \text{if } \theta_i(t) \leq \theta_r^i(t) + 0{,}5 \text{ and } m_i(t) = 0 \\ m_i(t) & \text{otherwise} \end{cases} \tag{8.2}$$

[1] El resultado que se exhibe es el promedio de 5 ejecuciones.

donde $\theta_r^i(t)$ es la temperatura de referencia, que tiene la siguiente forma:

$$\theta_r^i(t) = \begin{cases} 20 & \text{if } 0 \leq t < 1000 \\ 20{,}5 & \text{if } 1000 \leq t < 2000 \\ 20 & \text{if } 2000 \leq t \leq 3000 \end{cases} \tag{8.3}$$

Finalmente, las perturbaciones térmicas $w(t)$ son actualizadas cada minuto, asumiendo valores seudo-aleatorios en el rango $(-1, 1)$.

Este modelo se puede representar en código Modelica como:

```
1   model airconds
2     equation
3       for i in 1:N loop
4         der(th[i]) = (THA/RES[i]-POT[i]*on[i]-th[i]/RES[i]+
              noise[i]/RES[i])/CAP[i];
5       end for;
6
7     algorithm
8       for i in 1:N loop
9         when th[i] - tref[i] + on[i] - 0.5 > 0 then
10          on[i] := 1;
11        elsewhen th[i] - tref[i] + on[i] - 0.5 < 0 then
12          on[i] := 0;
13        end when;
14      end for;
15      for i in 1:N loop
16        when time > nextTref[i] then
17          if (nextTref[i] == 1000) then
18            tref[i] := 20.5;
19          else
20            tref[i] := 20;
21          end if;
22          nextTref[i] := 2000;
23        end when;
24      end for;
25      for i  in 1:N loop
26        when time > nextSample[i] then
27          nextSample[i] := nextSample[i]+1;
28          noise[i] := 2*abs(sin(i*time))-1;
29        end when;
30      end for;
31  end airconds;
```

Cabe mencionar, que en este modelo cada AA está modelado por una ecuación diferencial (líneas 3-5) y tres eventos: uno asociado a la ley de control con histéresis (líneas 8-14), otro correspondiente a la evolución de la temperatura de referencia (líneas 15-24) y el último asociado a la actualización de la señal de ruido (líneas 25-30).

Debido a esto, la dinámica de cada AA es independiente de la evolución de los restantes AAs y como consecuencia, se puede encontrar una partición ideal en la cual no existe comunicación.

En las Tablas 8.1, 8.2, 8.3 y 8.4 se muestran los resultados obtenidos para obtener 4 particiones variando la cantidad de AAs en el modelo.

Como mencionamos, este modelo permite encontrar una partición ideal en la cual no existe comunicación, esto se refleja en las métricas de calidad de las particiones, por otro lado, también podemos observar que aumentar la cantidad de AAs no afecta los tiempos de inicialización y particionado en nuestro algoritmo, en tanto que para METIS el crecimiento de los ambos tiempos es lineal.

Cabe mencionar que para SBG-Particioner, cuando se genera la partición inicial, la estrategia de particionamiento distributiva acomoda los nodos de modo tal que solo tengan comunicación interna y el algoritmo de optimización no itera.

A modo de ejemplo, la partición generada para $N = 10000$ es la siguiente:

```
{"partitions":[
    {"nodes":[[[0,2499]],[[10000,12499]],[[20000,
        22499]],[[30000,32499]]]},
    {"nodes":[[[2500,4999]],[[12500,14999]],[[22500,
        24999]],[[32500,34999]]]},
    {"nodes":[[[5000,7499]],[[15000,17499]],[[25000,
        27499]],[[35000,37499]]]},
    {"nodes":[[[7500,9999]],[[17500,19999]],[[27500,
        29999]],[[37500,39999]]]}
    ]
}
```

Donde cada elemento `nodes` indica una partición en forma de lista de intervalos. Los intervalos son representados con arreglos de dos elementos que indican principio y final del mismo. Teniendo en cuenta que:

- Los nodos $[0 : 9999]$ se corresponden con las líneas 3-5 del modelo.
- Los nodos $[10000 : 19999]$ se corresponden con las líneas 8-14 del modelo.
- Los nodos $[20000 : 29999]$ se corresponden con las líneas 15-24 del modelo.
- Los nodos $[30000 : 39999]$ se corresponden con las líneas 25-30 del modelo.

Podemos corroborar que la partición generada respeta la distribución ideal mencionada anteriormente.

Tabla 8.1: Modelo con tamaño 1000

Particionador	**Inicialización (ms)**	**Particionado (ms)**	**Edge cut**	**Communication volume**	**Maximum volume**	**Maximum imbalance**
SBG-Partitioner	$0,4$	$3,1$	0	0	0	0
METIS	$1,2$	$0,9$	0	0	0	$2,8e{-2}$

Tabla 8.2: Modelo con tamaño 10000

Particionador	**Inicialización (ms)**	**Particionado (ms)**	**Edge cut**	**Communication volume**	**Maximum volume**	**Maximum imbalance**
SBG-Partitioner	$0,4$	$2,4$	0	0	0	0
METIS	$12,9$	$7,4$	0	0	0	$1,4e{-2}$

Tabla 8.3: Modelo con tamaño 100000

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	0,4	2,8	0	0	0	0
METIS	139	141	0	0	0	1,7e−2

Tabla 8.4: Modelo con tamaño 1000000

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	0,3	2,2	0	0	0	0
METIS	1350	1422	0	0	0	0,9

La partición obtenida para el ejemplo 8.2 con `sbg-partitioner` es:

8.3.2. Modelo Advección–Difusión–Reacción

En este segundo ejemplo, evaluamos el modelo ADR presentado en la Sección 6.1.2, en las Tablas 8.5, 8.6, 8.7 y 8.8 se muestran los resultados modelos de diferente tamaño generando 4 particiones.

En este caso, dado que las variables que comunican datos entre las distintas particiones son $u[i]$ con $u[i-1]$ para un determinado rango. Por eso, el $edgeCut$ es igual a 3, pues es la arista que comunica los dos nodos adyacentes entre particiones es independientemente del tamaño.

Nuevamente podemos observar que el tiempo de inicialización y particionado se mantiene constante para SBG-Partitioner, mientras que crece de manera lineal para METIS.

Tabla 8.5: Modelo con tamaño 1000

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	0,5	4,73	3	6	2	0
METIS	4,4	0,2	3	6	2	2e − 2

Tabla 8.6: Modelo con tamaño 10000

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	0,4	5,31	3	6	2	0
METIS	2,9	1,2	3	6	2	1,28e2

Tabla 8.7: Modelo con tamaño 100000

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	0,5	5,51	3	6	2	0
METIS	26	12	3	6	2	1,76e−2

Tabla 8.8: Modelo con tamaño 1000000

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	0,7	5,42	3	6	2	0
METIS	272	190	3	6	2	1,76e−4

8.3.3. Población de Aires Acondicionados con Controlador

Este modelo, también presentado en [13], extiende el modelo presentado en la Sección 8.3.1, agregando un control centralizado del consumo de energía total. Con este fin, el conjunto de AA es dividido en secciones locales $S = \{s_1, \cdots, s_n\}$ y el consumo de energía de cada sección es calculado como:

$$p_i(t_k) = \sum_{j \in s_k} P_j \tag{8.4}$$

y el consumo total de energía se obtiene de la siguiente manera:

$$P(t_k) = \sum_{i \in S} p_i \tag{8.5}$$

El sistema de control global regula la energía total $P(t_k)$ para poder seguir el control deseado $P_r(t)$. Para esto, se utiliza una ley de control de integral proporcional (PI) para calcular la temperatura de referencia común:

$$\theta_r(t_k) = K_P \cdot [P_r(t_k) - P(t_k)] + K_L \cdot \int_{\tau=0}^{t_k} [P_r(\tau) - P(\tau) d\tau \tag{8.6}$$

El modelo Modelica correspondiente es el siguiente:

```
1  model airconds_cont
2    equation
3      for i in 1:N loop
4        der(th[i]) = (THA/RES[i]-POT[i]*on[i]-th[i]/RES[i]+
             noise[i]/RES[i])/CAP[i];
5      end for;
6      der(ierr) = pref-ptotals/pmax;
7     der(ptotal) = 0;
8
9    algorithm
10     for i in 1:SECTIONS loop
11       when time > partSample[i] then
12         partSample[i] := partSample[i]+1;
13         update[i] := 1;
14         sendPartTotal[i] := partTotal[i];
15         partTotal[i] := 0;
16       end when;
17     end for;
18
19     for i in 1:SECTIONS loop
20       when update[i] > 0.5 then
21         reinit(ptotal, ptotal + sendPartTotal[i]);
```

```
22          update[i] := 0;
23        end when;
24      end for;
25
26      when time > 1000 then
27        pref := 0.4;
28      end when;
29      when time > 2000 then
30        pref := 0.5;
31      end when;
32
33      when time > nextSample then
34        nextSample := nextSample+1;
35        ptotals := ptotal;
36        dtref := Kp*(ptotals/pmax-pref)-Ki*ierr;
37      end when;
38
39      for i in 1:N loop
40        when th[i] - tref -dtref + on[i] - 0.5 > 0 then
41          on[i] := 1;
42          partTotal[sections[i]]:= partTotal[sections[i]] +
                POT[i];
43        elsewhen th[i] - tref -dtref + on[i] - 0.5 < 0 then
44          on[i] := 0;
45          partTotal[sections[i]]:= partTotal[sections[i]] -
                POT[i];
46        end when;
47      end for;
48
49      for i in 1:N loop
50        when time > sampleNoise[i] then
51          sampleNoise[i] := sampleNoise[i] + 1;
52           inputs[i] := inputs[i] + 1;
53          noise[i] := 2*sin(i*inputs[i])^2;
54        end when;
55      end for;
56  end airconds_cont;
```

En este caso, el consumo de energía de cada sección se calcula en las líneas 10-17, en tanto que el consumo total se calculo con el evento definido en las líneas 19-24 y la temperatura de referencia se calcula en las líneas 33-37.

En este modelo, cada sección define un nodo en el grafo SBG y estos nodos se conectan con el controlador central. Los AAs pueden ser agrupados por sección por lo que se generan muchas conexiones a los nodos controladores parciales, esto complica el particionamiento.

En las Tablas 8.9, 8.10, 8.11 y 8.12 observamos los resultados obtenidos, si bien el tiempo de ejecución de nuestro algoritmo no es bueno, puesto que no se comporta bien cuando la cantidad de conexiones es alta y esto lo obliga a iterar sobre una gran cantidad de mapas cada vez que se desean encontrar conexiones entre nodos para calcular costo externo e interno. Como explicamos en la Sección 7.1, esperamos una mejora en el tiempo de ejecución de nuestro algoritmo al actualizar la librería SBG a su última versión.

También se puede observar a pesar de la diferencia en tiempo de particiona-

do, la calidad de las particiones generadas supera a las generadas por METIS en todos los casos. Por lo que podemos esperar mejores resultados durante la simulación del modelo.

Tabla 8.9: Modelo con tamaño 1000

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	5	12845	754	1505	751	1,3e−3
METIS	1,3	1,9	1252	1505	753	0,23

Tabla 8.10: Modelo con tamaño 10000

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	4	9580	7504	15005	7499	1,3e−4
METIS	12	9	7552	14983	7422	0,03

Tabla 8.11: Modelo con tamaño 100000

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	2	7275	75004	150005	74999	1,3e−5
METIS	182	105	149925	149927	75004	1,3e−5

Tabla 8.12: Modelo con tamaño 1000000

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	3	7360	750004	1500005	749999	1,3e−6
METIS	14741	1501	1500003	1500004	749973	6,5e−5

Los ejemplos con mayor número de nodos iteran menos veces para optimizar, esto genera una mejora en el rendimiento.

Ejemplo cambiando el número de particiones

Tabla 8.13: Modelo con tamaño 10000 con diferentes particiones

Particiones	Inicialización (ms)	Particionado (ms)
4	4	9580
8	4	40387
16	3,1	167404
32	3,3	639270

La construcción del grafo computacional no se ve afectada por el número de particiones. En cuando al tiempo de particionado, crece debido a que el

algoritmo optimiza las particiones buscando la óptima de dos en dos. Si bien se ejecutan en paralelo, a mayor cantidad de particiones, mayor cantidad de hilos de ejecución, se superan los núcleos disponibles y aumenta el tiempo de espera.

8.3.4. Red de Neuronas Pulsantes

El último modelo, adaptado de [14], representa una red de neuronas pulsantes. La i–ésima neurona es modelada por tres ecuaciones diferenciales:

$$\begin{aligned} \tau \cdot \dot{v}_i(t) &= v_{rest} - v_i(t) + g_i^{ex}(t) \cdot (E^{ex} - v_i(t)) + g_i^{inh}(t) \cdot (E^{inh} - v_i(t)) \\ \tau^{ex} \cdot \dot{g}_i^{ex}(t) &= -g_i^{ex}(t) \\ \tau^{inh} \cdot \dot{g}_i^{inh}(t) &= -g_i^{inh}(t) \end{aligned}$$

donde $v_i(t)$ representa el potencial de membrana, $g_i^{ex}(t)$ es la conductancia exitatoria, y $g_i^{inh}(t)$ es la conductancia inhibitoria. La definición y los valores de los restantes parámetros se pueden encontrar en [14].

Cuando el potencial de membrana $v_i(t)$ alcanza el valor límite -50, la neurona se descarga, lo que a su vez cambia su valor de manera instantánea a $v_i(t) = v_{rest} = -60$. La descarga es comunicada a todas las neuronas postsinápticas conectadas, que cambian el valor de las conductancias exitatorias e inhibitorias dependiendo del tipo de neurona que provoca el cambio. Si la i–ésima neurona es exitatoria, las neuronas postsinápticas conectadas cambian su conductancia exitatoria de acuerdo a:

$$g_j^{ex}(t^+) = g_h^{ex}(t) + \Delta g^{ex}$$

para todo $j \in Post_i$ (el conjunto de neuronas postsinápticas de la neurona i). Por otro lado, si la i–ésima neurona es de tipo inhibitorio, la conductancia inibitoria es modificada de la siguiente manera:

$$g_j^{inh}(t^+) = g_h^{inh}(t) + \Delta g^{inh}$$

para todo $j \in Post_i$. Para estas conexiones sinápticas, utilizamos parámetros $\Delta g^{ex} = 0{,}4$ y $\Delta g^{inh} = 1{,}6$.

Luego de la descarga de una neurona, la misma entra en un período refractario en el que no cambia su potencial de membrana.

En modelo Modelica en este caso es el siguiente:

```
1   model spl_neurons
2   equation
3     for i  in 1:N loop
4       der(v[i])=active[i]*((vrest-v[i])+gex[i]*(Eex-v[i])
            +ginh[i]*(Einh-v[i]))/tau;
5       der(gex[i])=-gex[i]/tauex;
6       der(ginh[i])=-ginh[i]/tauinh;
7       end for;
8
9   algorithm
10    for i in 1:N loop
11      when exReinit[i] > 0.5 then
12         reinit(gex[i],gex[i]+dgex);
13         exReinit[i]:=0;
```

```
14          end when;
15        end for;
16
17        for i in 1:N loop
18          when inhReinit[i] > 0.5 then
19             reinit(ginh[i],ginh[i]+dgin);
20             inhReinit[i]:=0;
21          end when;
22        end for;
23
24        for i in 1:N loop
25          when time>tfire[i]+Trefrac then
26            active[i]:=1;
27          end when;
28        end for;
29
30        for i in 1:M loop
31          when time>nextev[i] then
32            exReinit[i+100] := 1;
33             inputs[i] := inputs[i] + 1;
34            nextev[i]:=time+10*sin(i*inputs[i])^2;
35          end when;
36        end for;
37
38        for i in 1:N-OFFSET-CONN loop
39          when v[i]*ex[i]>vthres then
40            reinit(v[i],vrest);
41            reinit(ginh[i],0);
42            reinit(gex[i],0);
43             exReinit[i+100] := 1;
44             exReinit[i+101] := 1;
45             exReinit[i+102] := 1;
46             exReinit[i+103] := 1;
47             exReinit[i+104] := 1;
48             exReinit[i+105] := 1;
49             exReinit[i+106] := 1;
50             exReinit[i+107] := 1;
51             exReinit[i+108] := 1;
52             exReinit[i+109] := 1;
53            active[i]:=0;
54            tfire[i]:=time;
55          end when;
56        end for;
57
58        for i in 1:N-OFFSET-CONN loop
59          when v[i]*(1-ex[i])>vthres then
60            reinit(v[i],vrest);
61            reinit(ginh[i],0);
62            reinit(gex[i],0);
63             inhReinit[i+100] := 1;
64             inhReinit[i+101] := 1;
65             inhReinit[i+102] := 1;
66             inhReinit[i+103] := 1;
67             inhReinit[i+104] := 1;
```

```
68              inhReinit[i+105] := 1;
69              inhReinit[i+106] := 1;
70              inhReinit[i+107] := 1;
71              inhReinit[i+108] := 1;
72              inhReinit[i+109] := 1;
73             active[i]:=0;
74             tfire[i]:=time;
75           end when;
76         end for;
77   end spl_neurons;
```

Usualmente, en este tipo de modelos, las conexiones entre neuronas son aleatorias, pero debido a que esto nos impide medir la calidad de las particiones, modificamos el modelo para que tenga conexiones fijas con 10 neuronas a una distancia predeterminada entre ellas. De esta manera podemos esperar que la cantidad de conexiones se mantenga constante independientemente del tamaño del modelo.

Este modelo tiene además la dificultad de que cada conexión entre neuronas es individual (líneas 38-76 del modelo), lo cuál afecta el rendimiento del algoritmo dado que no se puede aprovechar la representación por intensión de SBG.

Las Tablas 8.14, 8.15, 8.16 y 8.17 muestran los resultados obtenidos, donde se puede ver como afecta la cantidad de aristas individuales al algoritmo dado que aumenta la cantidad de conexiones y el algoritmo de optimización debe iterar los mapas para calcular comunicación entre particiones. A pesar de esto, el tiempo necesario para obtener las particiones se mantiene constante y nuevamente la calidad de las particiones supera en todos los casos a las obtenidas por METIS.

Tabla 8.14: Modelo con tamaño 1000 y 10 conexiones

Tamaño	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	12	51289	5875	2430	869	0
METIS	5.47	9.86	8270	2309	624	0,04

Tabla 8.15: Modelo con tamaño 10000 y 10 conexiones

Partitionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	14	51289	5875	2430	869	0
METIS	46	47	17520	4098	1235	0,015

Tabla 8.16: Modelo con tamaño 100000 y 10 conexiones

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	19	54873	5875	2430	869	0
METIS	525	485	17640	3982	1196	0,002

Tabla 8.17: Modelo con tamaño 1000000 y 10 conexiones

Particionador	Inicialización (ms)	Particionado (ms)	Edge cut	Communication volume	Maximum volume	Maximum imbalance
SBG-Partitioner	10	51455	5875	2430	869	0
METIS	5136	5842	17490	4127	1248	0,001

Ejemplo aumentando el número de conexiones

Tabla 8.18: Modelo con tamaño 10000 para 4 particiones con diferente número de conexiones

Conexiones	Inicialización (ms)	Particionado (ms)
10	12	51289
20	15	78853
50	56	243786
100	90	377736

El algoritmo es sensible a la cantidad de conexiones, se ve obligado a computar mapas para distintos dominios en busca de conexiones para calcular costo interno y externo de comunicación. Esto escala mal, puesto que a mayor comunicación, debe iterar más veces para poder optimizar.

Capítulo 9

Conclusiones y Trabajo futuro

El algoritmo cumple con lo propuesto. Nuestra expectativa era un algoritmo que genere particiones de buena calidad, y no sea sensible al aumento de la cantidad de nodos, y eso se cumple según hemos visto en los ejemplos. Si bien no hemos medido el uso de memoria, tenemos herramientas para creer que es constante, puesto que la cantidad de intervalos es la misma, y la memoria necesaria para alojarlos no varía si contienen números más grandes.

En cuanto al trabajo futuro, esto ha sido una prueba de concepto satisfactoria, pero muchas de las decisiones tomadas, tras haber realizado una serie de experimentos, notamos que pueden mejorarse.

Mencionamos la intención de evitar conexiones *reflexivas*, aristas cuyos bordes son el mismo nodo. Para resolver esto lo que hacemos es chequear si las imágenes de ambos mapas son iguales. Esto sirve para los ejemplos que hemos visto, pero no para todos los casos. Supongamos la existencia de los siguientes mapas:

$$map_1 : [0:5] \to \mathcal{N}$$
$$map_1(x) = x$$
$$map_2 : [0:5] \to \mathcal{N}$$
$$map_2(x) = -x + 5$$

La imagen de ambos mapas es la misma:

$$IM(map_1) = IM(map_2) = \{0, 1, 2, 3, 4, 5\}$$

Pero las aristas que forman no son bucles:

$$map_1(0) = 0$$
$$map_2(0) = 5$$

El algoritmo itera entre las particiones actuales, de dos en dos, buscando mejorarlas. Para esto, crea una estructura de datos que aloja información correspondiente a la comunicación interna y externa de cada intervalo de las particiones. Esto es información local de la optimización entre las dos particiones y se pierde cuando termina. En la siguiente iteración, aún cuando estas

dos particiones no se hayan modificado, todos los cálculo deben repetirse. Tener en cuenta si las particiones cambiaron, así como mantener un *estado global* de la comunicación entre los intervalos de nodos, ayudaría a evitar recomputar operaciones costosas innecesariamente.

En muchos casos, la posibilidad de desbalancear las particiones, dentro de un valor dado por el usuario, puede mejorar la calidad de las particiones. Nuestro algoritmo es muy rígido en cuanto a esto, solo intercambia igual cantidad de nodos entre particiones. Si bien contamos con una implementación experimental, causaba una gran sobrecarga en el algoritmo.

En cuanto a la fase de optimización, lo que buscamos es disminuir el *edgeCut*, pero no tenemos en cuenta que el cambio de una variable podría implicar la sincronización con muchas otras particiones, aún teniendo *edgeCut* relativamente bajo. Esto podría mejorarse si calculamos las componentes conexas del grafo, en lugar de las aristas individuales. Aquí buscaríamos disminuir el volumen de comunicación.

Tras haber implementado una prueba de concepto satisfactoria usando Grafos Basados en Conjuntos para un algoritmo de particionamiento estático, a futuro sería interesante implementar algoritmos de particionamiento semi-estático y dinámico. Principalmente para el particionado dinámico, puesto que el costo computacional y el uso de memoria son los mayores inconvenientes, y el uso de esta estructura de datos compacta comienza a mostrar resultados prometedores.

Bibliografía

[1] Konstantin Andreev and Harald Räcke. Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, page 120–124, New York, NY, USA, 2004. Association for Computing Machinery.

[2] Edward A Bender. *An introduction to mathematical modeling*. Courier Corporation, 2000.

[3] Federico Bergero, Joaquín Fernández, Ernesto Kofman, and Margarita Portapila. Time Discretization versus State Quantization in the Simulation of a 1D Advection-Diffusion-Reaction Equation. *Simulation: Transactions of the Society for Modeling and Simulation International*, 92(1):47–61, 2016.

[4] Federico Bergero, Ernesto Kofman, and François Cellier. A novel parallelization technique for devs simulation of continuous and hybrid systems. *SIMULATION*, 89(6):663–683, 2013.

[5] François Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer, New York, 2003. In preparation.

[6] Spyros Chatzivasileiadis, Marco Bonvini, Javier Matanza, Rongxin Yin, Thierry S Nouidui, Emre C Kara, Rajiv Parmar, David Lorenzetti, Michael Wetter, and Sila Kiliccote. Cyber–physical modeling of distributed resources for distribution system operations. *Proceedings of the IEEE*, 104(4):789–806, 2016.

[7] Joaquín Fernández, Ernesto Kofman, and Federico Bergero. A parallel quantized state system solver for odes. *Journal of Parallel and Distributed Computing*, 106:14–30, 2017.

[8] Peter Fritzson and Peter Bunus. Modelica-a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Proceedings 35th Annual Simulation Symposium. SS 2002*, pages 365–380. IEEE, 2002.

[9] Peter Fritzson and Vadim Engelson. Modelica—a unified object-oriented language for system modeling and simulation. In *ECOOP'98—Object-Oriented Programming: 12th European Conference Brussels, Belgium, July 20–24, 1998 Proceedings 12*, pages 67–90. Springer, 1998.

[10] George Karypis and Vipin Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing

orderings of sparse matrices. Technical report, University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN, 1997.

[11] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[12] Denise Marzorati. Aplanado eficiente de grandes sistemas de ecuaciones algebraico-diferenciales. Master's thesis, Facultad de Ciencias Exactas, Ingeniería y Agrimensura - Universidad Nacional de Rosario, Diciembre 2020.

[13] C. Perfumo, E. Kofman, J. Braslavsky, and J.K. Ward. Load Management: Model-Based Control of Aggregate Power for Populations of Thermostatically Controlled Loads. *Energy Conversion and Management*, 55:36–48, 2012.

[14] Tim P Vogels and Larry F Abbott. Signal propagation and logic gating in networks of integrate-and-fire neurons. *Journal of neuroscience*, 25(46):10786–10795, 2005.

[15] Bernard Zeigler. *Theory of Modeling and Simulation.* John Wiley & Sons, New York, 1976.

[16] Bernard P Zeigler. Theory of modeling and simulation: Discrete event & iterative system computational foundations, 2018.

[17] Pablo Zimmermann, Joaquín Fernández, and Ernesto Kofman. Set-based graph methods for fast equation sorting in large dae systems. In *Proceedings of the 9th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT '19, page 45–54, New York, NY, USA, 2020. Association for Computing Machinery.

Apéndice A

Salida estándar del Particionador

Aquí exhibimos un fragmento de la salida estándar de la ejecución del particionador `sbg-partitioner` en su fase de optimización para el ejemplo cuyos resultados fueron presentados en la Tabla 8.2.

Entre las líneas 1 y 4 indica las particiones con las que inicia la fase de optimización. Los nodos son referenciados por medio de índices. Por ejemplo, el intervalo con índice 1 de la primer partición es `[10000:10000]`.

Luego desde la línea 5 hasta la 94 se imprimió la comunicación externa e interna de los intervalos de cada partición.

De la 95 a la 186, es cada fila de la matriz de ganancia, ordenada según la ganancia de cada intercambio.

El elemento

```
< Node: 3, size: 1 - Node: 7, size: 1, gain: -1 >
```

Indica que el intervalo 3 de la partición 1 y el intervalo 7 de la partición 2 tienen ganancia -1 en caso de ser intercambiados, y pueden cambiar un elemento de cada uno para evitar desbalance. En la siguiente muestra cual es el intercambio elegido, realiza el intercambio y actualiza los conjuntos y la matriz de ganancia, con una ganancia de 2500.

```
1 Algorithm starts with:
2       {[0:2499], [10000:10000], [10002:10002], [10006:10006],
            [10012:10012], [10013:12512], [20013:20637],
           [22513:23137], [25013:25637], [27513:28137]},
3       {[2500:4999], [10003:10003], [10007:10007],
            [10011:10011], [12513:15012], [20638:21262],
            [23138:23762], [25638:26262], [28138:28762]}
4
5 Node 0, {[0:2499]} ec: {[30638:31262], [60657:61281],
     [63158:63782], [65659:66283], [68160:68784]} and ic:
     {[30013:30637], [40013:42512], [60032:60656],
     [62533:63157], [65034:65658], [67535:68159]}
6 Node 0, {[0:2499]} ec: {} and ic: {[30013:30013],
     [40013:40013], [60032:60032], [62533:62533],
     [65034:65034], [67535:67535]}
```

```
7 Node 0, {[0:2499]} ec: {} and ic: {[30013:30013],
      [40013:40013], [60032:60032], [62533:62533],
      [65034:65034], [67535:67535]}
8 Node 0, {[0:2499]} ec: {} and ic: {[30013:30013],
      [40013:40013], [60032:60032], [62533:62533],
      [65034:65034], [67535:67535]}
9 Node 0, {[0:2499]} ec: {[30638:31262], [60657:61281],
      [63158:63782], [65659:66283], [68160:68784]} and ic:
      {[30013:30637], [40013:42512], [60032:60656],
      [62533:63157], [65034:65658], [67535:68159]}
10 Node 0, {[0:2499]} ec: {} and ic: {[30013:30637],
      [40013:40637], [60032:60656], [62533:63157],
      [65034:65658], [67535:68159]}
11 Node 0, {[0:2499]} ec: {} and ic: {[30013:30637],
      [40013:40637], [60032:60656], [62533:63157],
      [65034:65658], [67535:68159]}
12 Node 0, {[0:2499]} ec: {} and ic: {[30013:30637],
      [40013:40637], [60032:60656], [62533:63157],
      [65034:65658], [67535:68159]}
13 Node 0, {[0:2499]} ec: {} and ic: {[30013:30637],
      [40013:40637], [60032:60656], [62533:63157],
      [65034:65658], [67535:68159]}
14 Node 1, {[10000:10000]} ec: {[50014:50014]} and ic:
      {[50015:50015], [60031:60031]}
15 Node 1, {[10000:10000]} ec: {[50014:50014]} and ic:
      {[50015:50015], [60031:60031]}
16 Node 1, {[10000:10000]} ec: {[50014:50014]} and ic:
      {[50015:50015], [60031:60031]}
17 Node 1, {[10000:10000]} ec: {[50014:50014]} and ic:
      {[50015:50015], [60031:60031]}
18 Node 1, {[10000:10000]} ec: {[50014:50014]} and ic:
      {[50015:50015], [60031:60031]}
19 Node 1, {[10000:10000]} ec: {[50014:50014]} and ic:
      {[50015:50015], [60031:60031]}
20 Node 1, {[10000:10000]} ec: {[50014:50014]} and ic:
      {[50015:50015], [60031:60031]}
21 Node 1, {[10000:10000]} ec: {[50014:50014]} and ic:
      {[50015:50015], [60031:60031]}
22 Node 1, {[10000:10000]} ec: {[50014:50014]} and ic:
      {[50015:50015], [60031:60031]}
23 Node 2, {[10002:10002]} ec: {[50641:51265]} and ic:
      {[50016:50640], [60016:60016], [60021:60021]}
24 Node 2, {[10002:10002]} ec: {[50641:51265]} and ic:
      {[50016:50640], [60016:60016], [60021:60021]}
25 Node 2, {[10002:10002]} ec: {[50641:51265]} and ic:
      {[50016:50640], [60016:60016], [60021:60021]}
26 Node 2, {[10002:10002]} ec: {[50641:51265]} and ic:
      {[50016:50640], [60016:60016], [60021:60021]}
27 Node 2, {[10002:10002]} ec: {[50641:51265]} and ic:
      {[50016:50640], [60016:60016], [60021:60021]}
28 Node 2, {[10002:10002]} ec: {[50641:51265]} and ic:
      {[50016:50640], [60016:60016], [60021:60021]}
29 Node 2, {[10002:10002]} ec: {[50641:51265]} and ic:
      {[50016:50640], [60016:60016], [60021:60021]}
```

```
30 Node 2, {[10002:10002]} ec: {[50641:51265]} and ic:
       {[50016:50640], [60016:60016], [60021:60021]}
31 Node 2, {[10002:10002]} ec: {[50641:51265]} and ic:
       {[50016:50640], [60016:60016], [60021:60021]}
32 Node 3, {[10006:10006]} ec: {} and ic: {[60016:60016],
       [60021:60021]}
33 Node 3, {[10006:10006]} ec: {} and ic: {[60016:60016],
       [60021:60021]}
34 Node 3, {[10006:10006]} ec: {} and ic: {[60016:60016],
       [60021:60021]}
35 Node 3, {[10006:10006]} ec: {} and ic: {[60016:60016],
       [60021:60021]}
36 Node 3, {[10006:10006]} ec: {} and ic: {[60016:60016],
       [60021:60021]}
37 Node 3, {[10006:10006]} ec: {} and ic: {[60016:60016],
       [60021:60021]}
38 Node 3, {[10006:10006]} ec: {} and ic: {[60016:60016],
       [60021:60021]}
39 Node 3, {[10006:10006]} ec: {} and ic: {[60016:60016],
       [60021:60021]}
40 Node 3, {[10006:10006]} ec: {} and ic: {[60016:60016],
       [60021:60021]}
41 Node 4, {[10012:10012]} ec: {[60030:60030]} and ic:
       {[50015:50015], [60031:60031], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
42 Node 4, {[10012:10012]} ec: {[60030:60030]} and ic:
       {[50015:50015], [60031:60031], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
43 Node 4, {[10012:10012]} ec: {[60030:60030]} and ic:
       {[50015:50015], [60031:60031], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
44 Node 4, {[10012:10012]} ec: {[60030:60030]} and ic:
       {[50015:50015], [60031:60031], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
45 Node 4, {[10012:10012]} ec: {[60030:60030]} and ic:
       {[50015:50015], [60031:60031], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
46 Node 4, {[10012:10012]} ec: {[60030:60030]} and ic:
       {[50015:50015], [60031:60031], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
47 Node 4, {[10012:10012]} ec: {[60030:60030]} and ic:
       {[50015:50015], [60031:60031], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
48 Node 4, {[10012:10012]} ec: {[60030:60030]} and ic:
       {[50015:50015], [60031:60031], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
49 Node 4, {[10012:10012]} ec: {[60030:60030]} and ic:
       {[50015:50015], [60031:60031], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
50 Node 5, {[10013:12512]} ec: {} and ic: {[40013:42512]}
51 Node 5, {[10013:12512]} ec: {} and ic: {[40013:40013]}
52 Node 5, {[10013:12512]} ec: {} and ic: {[40013:40013]}
53 Node 5, {[10013:12512]} ec: {} and ic: {[40013:40013]}
54 Node 5, {[10013:12512]} ec: {} and ic: {[40013:42512]}
```

```
55 Node 5, {[10013:12512]} ec: {} and ic: {[40013:40637]}
56 Node 5, {[10013:12512]} ec: {} and ic: {[40013:40637]}
57 Node 5, {[10013:12512]} ec: {} and ic: {[40013:40637]}
58 Node 5, {[10013:12512]} ec: {} and ic: {[40013:40637]}
59 Node 6, {[20013:20637]} ec: {} and ic: {[30013:30637],
       [50016:50640], [60032:60656], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
60 Node 6, {[20013:20637]} ec: {} and ic: {[30013:30013],
       [50016:50016], [60032:60032], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
61 Node 6, {[20013:20637]} ec: {} and ic: {[30013:30013],
       [50016:50016], [60032:60032], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
62 Node 6, {[20013:20637]} ec: {} and ic: {[30013:30013],
       [50016:50016], [60032:60032], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
63 Node 6, {[20013:20637]} ec: {} and ic: {[30013:30637],
       [50016:50640], [60032:60656], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
64 Node 6, {[20013:20637]} ec: {} and ic: {[30013:30637],
       [50016:50640], [60032:60656], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
65 Node 6, {[20013:20637]} ec: {} and ic: {[30013:30637],
       [50016:50640], [60032:60656], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
66 Node 6, {[20013:20637]} ec: {} and ic: {[30013:30637],
       [50016:50640], [60032:60656], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
67 Node 6, {[20013:20637]} ec: {} and ic: {[30013:30637],
       [50016:50640], [60032:60656], [62532:62532],
       [65033:65033], [67534:67534], [70035:70035]}
68 Node 7, {[22513:23137]} ec: {[32513:33137], [52516:53140]}
       and ic: {[62533:63157]}
69 Node 7, {[22513:23137]} ec: {[32513:32513], [52516:52516]}
       and ic: {[62533:62533]}
70 Node 7, {[22513:23137]} ec: {[32513:32513], [52516:52516]}
       and ic: {[62533:62533]}
71 Node 7, {[22513:23137]} ec: {[32513:32513], [52516:52516]}
       and ic: {[62533:62533]}
72 Node 7, {[22513:23137]} ec: {[32513:33137], [52516:53140]}
       and ic: {[62533:63157]}
73 Node 7, {[22513:23137]} ec: {[32513:33137], [52516:53140]}
       and ic: {[62533:63157]}
74 Node 7, {[22513:23137]} ec: {[32513:33137], [52516:53140]}
       and ic: {[62533:63157]}
75 Node 7, {[22513:23137]} ec: {[32513:33137], [52516:53140]}
       and ic: {[62533:63157]}
76 Node 7, {[22513:23137]} ec: {[32513:33137], [52516:53140]}
       and ic: {[62533:63157]}
77 Node 8, {[25013:25637]} ec: {} and ic: {[65034:65658]}
78 Node 8, {[25013:25637]} ec: {} and ic: {[65034:65034]}
79 Node 8, {[25013:25637]} ec: {} and ic: {[65034:65034]}
80 Node 8, {[25013:25637]} ec: {} and ic: {[65034:65034]}
81 Node 8, {[25013:25637]} ec: {} and ic: {[65034:65658]}
```

```
Node 8, {[25013:25637]} ec: {} and ic: {[65034:65658]}
Node 8, {[25013:25637]} ec: {} and ic: {[65034:65658]}
Node 8, {[25013:25637]} ec: {} and ic: {[65034:65658]}
Node 8, {[25013:25637]} ec: {} and ic: {[65034:65658]}
Node 9, {[27513:28137]} ec: {} and ic: {[67535:68159]}
Node 9, {[27513:28137]} ec: {} and ic: {[67535:67535]}
Node 9, {[27513:28137]} ec: {} and ic: {[67535:67535]}
Node 9, {[27513:28137]} ec: {} and ic: {[67535:67535]}
Node 9, {[27513:28137]} ec: {} and ic: {[67535:68159]}
Node 9, {[27513:28137]} ec: {} and ic: {[67535:68159]}
Node 9, {[27513:28137]} ec: {} and ic: {[67535:68159]}
Node 9, {[27513:28137]} ec: {} and ic: {[67535:68159]}
Node 9, {[27513:28137]} ec: {} and ic: {[67535:68159]}
{
 < Node: 7, size: 625 - Node: 5, size: 625, gain: 2500 >
 < Node: 9, size: 625 - Node: 5, size: 625, gain: 1250 >
 < Node: 8, size: 625 - Node: 5, size: 625, gain: 1250 >
 < Node: 7, size: 625 - Node: 8, size: 625, gain: 1250 >
 < Node: 7, size: 625 - Node: 7, size: 625, gain: 1250 >
 < Node: 5, size: 625 - Node: 5, size: 625, gain: 1250 >
 < Node: 7, size: 1 - Node: 3, size: 1, gain: 3 >
 < Node: 1, size: 1 - Node: 5, size: 1, gain: 2 >
 < Node: 9, size: 1 - Node: 3, size: 1, gain: 1 >
 < Node: 8, size: 1 - Node: 3, size: 1, gain: 1 >
 < Node: 5, size: 1 - Node: 3, size: 1, gain: 1 >
 < Node: 3, size: 1 - Node: 5, size: 1, gain: 1 >
 < Node: 9, size: 625 - Node: 8, size: 625, gain: 0 >
 < Node: 9, size: 625 - Node: 7, size: 625, gain: 0 >
 < Node: 8, size: 625 - Node: 8, size: 625, gain: 0 >
 < Node: 8, size: 625 - Node: 7, size: 625, gain: 0 >
 < Node: 7, size: 625 - Node: 6, size: 625, gain: 0 >
 < Node: 7, size: 625 - Node: 4, size: 625, gain: 0 >
 < Node: 5, size: 625 - Node: 8, size: 625, gain: 0 >
 < Node: 5, size: 625 - Node: 7, size: 625, gain: 0 >
 < Node: 3, size: 1 - Node: 3, size: 1, gain: 0 >
 < Node: 2, size: 1 - Node: 3, size: 1, gain: 0 >
 < Node: 1, size: 1 - Node: 8, size: 1, gain: 0 >
 < Node: 1, size: 1 - Node: 7, size: 1, gain: 0 >
 < Node: 7, size: 1 - Node: 2, size: 1, gain: -1 >
 < Node: 3, size: 1 - Node: 8, size: 1, gain: -1 >
 < Node: 3, size: 1 - Node: 7, size: 1, gain: -1 >
 < Node: 2, size: 1 - Node: 8, size: 1, gain: -1 >
 < Node: 2, size: 1 - Node: 7, size: 1, gain: -1 >
 < Node: 2, size: 1 - Node: 5, size: 1, gain: -1 >
 < Node: 1, size: 1 - Node: 3, size: 1, gain: -1 >
 < Node: 1, size: 1 - Node: 0, size: 1, gain: -1 >
 < Node: 4, size: 1 - Node: 5, size: 1, gain: -2 >
 < Node: 3, size: 1 - Node: 0, size: 1, gain: -2 >
 < Node: 2, size: 1 - Node: 0, size: 1, gain: -2 >
 < Node: 1, size: 1 - Node: 6, size: 1, gain: -2 >
 < Node: 1, size: 1 - Node: 4, size: 1, gain: -2 >
 < Node: 9, size: 1 - Node: 2, size: 1, gain: -3 >
 < Node: 9, size: 1 - Node: 1, size: 1, gain: -3 >
 < Node: 8, size: 1 - Node: 2, size: 1, gain: -3 >
```

```
136 < Node: 8, size: 1 - Node: 1, size: 1, gain: -3 >
137 < Node: 7, size: 1 - Node: 1, size: 1, gain: -3 >
138 < Node: 5, size: 1 - Node: 2, size: 1, gain: -3 >
139 < Node: 5, size: 1 - Node: 1, size: 1, gain: -3 >
140 < Node: 3, size: 1 - Node: 6, size: 1, gain: -3 >
141 < Node: 3, size: 1 - Node: 4, size: 1, gain: -3 >
142 < Node: 2, size: 1 - Node: 6, size: 1, gain: -3 >
143 < Node: 2, size: 1 - Node: 4, size: 1, gain: -3 >
144 < Node: 1, size: 1 - Node: 2, size: 1, gain: -3 >
145 < Node: 1, size: 1 - Node: 1, size: 1, gain: -3 >
146 < Node: 6, size: 625 - Node: 5, size: 625, gain: -4 >
147 < Node: 4, size: 1 - Node: 8, size: 1, gain: -4 >
148 < Node: 4, size: 1 - Node: 7, size: 1, gain: -4 >
149 < Node: 3, size: 1 - Node: 2, size: 1, gain: -4 >
150 < Node: 3, size: 1 - Node: 1, size: 1, gain: -4 >
151 < Node: 2, size: 1 - Node: 2, size: 1, gain: -4 >
152 < Node: 2, size: 1 - Node: 1, size: 1, gain: -4 >
153 < Node: 0, size: 1 - Node: 3, size: 1, gain: -4 >
154 < Node: 6, size: 1 - Node: 3, size: 1, gain: -5 >
155 < Node: 4, size: 1 - Node: 3, size: 1, gain: -5 >
156 < Node: 4, size: 1 - Node: 0, size: 1, gain: -5 >
157 < Node: 4, size: 1 - Node: 6, size: 1, gain: -6 >
158 < Node: 4, size: 1 - Node: 4, size: 1, gain: -6 >
159 < Node: 4, size: 1 - Node: 2, size: 1, gain: -7 >
160 < Node: 4, size: 1 - Node: 1, size: 1, gain: -7 >
161 < Node: 0, size: 1 - Node: 2, size: 1, gain: -8 >
162 < Node: 0, size: 1 - Node: 1, size: 1, gain: -8 >
163 < Node: 6, size: 1 - Node: 2, size: 1, gain: -9 >
164 < Node: 6, size: 1 - Node: 1, size: 1, gain: -9 >
165 < Node: 9, size: 625 - Node: 0, size: 625, gain: -625 >
166 < Node: 8, size: 625 - Node: 0, size: 625, gain: -625 >
167 < Node: 7, size: 625 - Node: 0, size: 625, gain: -625 >
168 < Node: 9, size: 625 - Node: 6, size: 625, gain: -1250 >
169 < Node: 9, size: 625 - Node: 4, size: 625, gain: -1250 >
170 < Node: 8, size: 625 - Node: 6, size: 625, gain: -1250 >
171 < Node: 8, size: 625 - Node: 4, size: 625, gain: -1250 >
172 < Node: 5, size: 625 - Node: 6, size: 625, gain: -1250 >
173 < Node: 6, size: 625 - Node: 8, size: 625, gain: -1254 >
174 < Node: 6, size: 625 - Node: 7, size: 625, gain: -1254 >
175 < Node: 0, size: 625 - Node: 5, size: 625, gain: -1875 >
176 < Node: 6, size: 625 - Node: 0, size: 625, gain: -1879 >
177 < Node: 6, size: 625 - Node: 6, size: 625, gain: -2504 >
178 < Node: 6, size: 625 - Node: 4, size: 625, gain: -2504 >
179 < Node: 0, size: 625 - Node: 8, size: 625, gain: -3125 >
180 < Node: 0, size: 625 - Node: 7, size: 625, gain: -3125 >
181 < Node: 0, size: 625 - Node: 6, size: 625, gain: -4375 >
182 < Node: 5, size: 2500 - Node: 4, size: 2500, gain: -5000 >
183 < Node: 5, size: 2500 - Node: 0, size: 2500, gain: -5000 >
184 < Node: 0, size: 2500 - Node: 4, size: 2500, gain: -5000 >
185 < Node: 0, size: 2500 - Node: 0, size: 2500, gain: -5000 >
186 }
187 inside the while {[0:2499], [10000:10000], [10002:10002],
      [10006:10006], [10012:10012], [10013:12512],
      [20013:20637], [22513:23137], [25013:25637],
```

```
      [27513:28137]}{[2500:4999], [10003:10003],
      [10007:10007], [10011:10011], [12513:15012],
      [20638:21262], [23138:23762], [25638:26262],
      [28138:28762]}
188
189 The best is < Node: 7, size: 625 - Node: 5, size: 625, gain:
       2500 >
190 < Node: 7, size: 625 - Node: 5, size: 625, gain: 2500 >
191 we remove {[22513:23137]} from {[0:2499], [10000:10000],
      [10002:10002], [10006:10006], [10012:10012],
      [10013:12512], [20013:20637], [22513:23137],
      [25013:25637], [27513:28137]}
192 and we get: {[0:2499], [10000:10000], [10002:10002],
      [10006:10006], [10012:10012], [10013:12512],
      [20013:20637], [25013:25637], [27513:28137]}
193
194 we remove {[20638:21262]} from {[2500:4999], [10003:10003],
      [10007:10007], [10011:10011], [12513:15012],
      [20638:21262], [23138:23762], [25638:26262],
      [28138:28762]}
195 and we get: {[2500:4999], [10003:10003], [10007:10007],
      [10011:10011], [12513:15012], [23138:23762],
      [25638:26262], [28138:28762]}
196
197 {[22513:23137]}, {}
198 {[20638:21262]}, {}
199
200 {[0:2499], [10000:10000], [10002:10002], [10006:10006],
      [10012:10012], [10013:12512], [20013:20637],
      [25013:25637], [27513:28137]}, {[2500:4999],
      [10003:10003], [10007:10007], [10011:10011],
      [12513:15012], [23138:23762], [25638:26262],
      [28138:28762]}, < Node: 7, size: 625 - Node: 5, size:
      625, gain: 2500 >, {
201  < Node: 8, size: 1 - Node: 3, size: 1, gain: 1 >
202  < Node: 7, size: 1 - Node: 3, size: 1, gain: 1 >
203  < Node: 5, size: 1 - Node: 3, size: 1, gain: 1 >
204  < Node: 8, size: 625 - Node: 7, size: 625, gain: 0 >
205  < Node: 8, size: 625 - Node: 6, size: 625, gain: 0 >
206  < Node: 7, size: 625 - Node: 7, size: 625, gain: 0 >
207  < Node: 7, size: 625 - Node: 6, size: 625, gain: 0 >
208  < Node: 5, size: 625 - Node: 7, size: 625, gain: 0 >
209  < Node: 5, size: 625 - Node: 6, size: 625, gain: 0 >
210  < Node: 3, size: 1 - Node: 3, size: 1, gain: 0 >
211  < Node: 1, size: 1 - Node: 7, size: 1, gain: 0 >
212  < Node: 1, size: 1 - Node: 6, size: 1, gain: 0 >
213  < Node: 3, size: 1 - Node: 7, size: 1, gain: -1 >
214  < Node: 3, size: 1 - Node: 6, size: 1, gain: -1 >
215  < Node: 1, size: 1 - Node: 3, size: 1, gain: -1 >
216  < Node: 1, size: 1 - Node: 5, size: 1, gain: -2 >
217  < Node: 1, size: 1 - Node: 4, size: 1, gain: -2 >
218  < Node: 1, size: 1 - Node: 0, size: 1, gain: -2 >
219  < Node: 0, size: 1 - Node: 3, size: 1, gain: -3 >
220  < Node: 8, size: 1 - Node: 2, size: 1, gain: -3 >
```

```
221 < Node: 7, size: 1 - Node: 2, size: 1, gain: -3 >
222 < Node: 5, size: 1 - Node: 2, size: 1, gain: -3 >
223 < Node: 3, size: 1 - Node: 5, size: 1, gain: -3 >
224 < Node: 3, size: 1 - Node: 4, size: 1, gain: -3 >
225 < Node: 1, size: 1 - Node: 2, size: 1, gain: -3 >
226 < Node: 3, size: 1 - Node: 0, size: 1, gain: -3 >
227 < Node: 4, size: 1 - Node: 7, size: 1, gain: -4 >
228 < Node: 4, size: 1 - Node: 6, size: 1, gain: -4 >
229 < Node: 3, size: 1 - Node: 2, size: 1, gain: -4 >
230 < Node: 6, size: 1 - Node: 3, size: 1, gain: -5 >
231 < Node: 4, size: 1 - Node: 3, size: 1, gain: -5 >
232 < Node: 4, size: 1 - Node: 5, size: 1, gain: -6 >
233 < Node: 4, size: 1 - Node: 4, size: 1, gain: -6 >
234 < Node: 4, size: 1 - Node: 0, size: 1, gain: -6 >
235 < Node: 0, size: 1 - Node: 2, size: 1, gain: -7 >
236 < Node: 4, size: 1 - Node: 2, size: 1, gain: -7 >
237 < Node: 6, size: 1 - Node: 2, size: 1, gain: -9 >
238 < Node: 2, size: 1 - Node: 3, size: 1, gain: -625 >
239 < Node: 2, size: 1 - Node: 7, size: 1, gain: -626 >
240 < Node: 2, size: 1 - Node: 6, size: 1, gain: -626 >
241 < Node: 8, size: 1 - Node: 1, size: 1, gain: -628 >
242 < Node: 7, size: 1 - Node: 1, size: 1, gain: -628 >
243 < Node: 5, size: 1 - Node: 1, size: 1, gain: -628 >
244 < Node: 2, size: 1 - Node: 5, size: 1, gain: -628 >
245 < Node: 2, size: 1 - Node: 4, size: 1, gain: -628 >
246 < Node: 1, size: 1 - Node: 1, size: 1, gain: -628 >
247 < Node: 2, size: 1 - Node: 0, size: 1, gain: -628 >
248 < Node: 3, size: 1 - Node: 1, size: 1, gain: -629 >
249 < Node: 2, size: 1 - Node: 2, size: 1, gain: -629 >
250 < Node: 0, size: 1 - Node: 1, size: 1, gain: -632 >
251 < Node: 4, size: 1 - Node: 1, size: 1, gain: -632 >
252 < Node: 6, size: 1 - Node: 1, size: 1, gain: -634 >
253 < Node: 8, size: 625 - Node: 5, size: 625, gain: -1250 >
254 < Node: 8, size: 625 - Node: 4, size: 625, gain: -1250 >
255 < Node: 7, size: 625 - Node: 5, size: 625, gain: -1250 >
256 < Node: 7, size: 625 - Node: 4, size: 625, gain: -1250 >
257 < Node: 5, size: 625 - Node: 5, size: 625, gain: -1250 >
258 < Node: 8, size: 625 - Node: 0, size: 625, gain: -1250 >
259 < Node: 7, size: 625 - Node: 0, size: 625, gain: -1250 >
260 < Node: 6, size: 625 - Node: 7, size: 625, gain: -1254 >
261 < Node: 6, size: 625 - Node: 6, size: 625, gain: -1254 >
262 < Node: 2, size: 1 - Node: 1, size: 1, gain: -1254 >
263 < Node: 6, size: 625 - Node: 5, size: 625, gain: -2504 >
264 < Node: 6, size: 625 - Node: 4, size: 625, gain: -2504 >
265 < Node: 6, size: 625 - Node: 0, size: 625, gain: -2504 >
266 < Node: 0, size: 625 - Node: 7, size: 625, gain: -3750 >
267 < Node: 0, size: 625 - Node: 6, size: 625, gain: -3750 >
268 < Node: 5, size: 2500 - Node: 4, size: 2500, gain: -5000 >
269 < Node: 0, size: 625 - Node: 5, size: 625, gain: -5000 >
270 < Node: 5, size: 2500 - Node: 0, size: 2500, gain: -5625 >
271 < Node: 0, size: 2500 - Node: 4, size: 2500, gain: -5625 >
272 < Node: 0, size: 2500 - Node: 0, size: 2500, gain: -6250 >
273 }
```