Simulación Paralela de Sistemas Continuos a través de Nodos Virtuales

Tesina de Grado

Licenciatura en Ciencias de la Computación

Autor: Manuel Dipré
Legajo: D-3725/7

Director: Ernesto Kofman
Co-Director: Federico Bergero

23 de diciembre de 2016

Dedicado a mi familia y amigos, ya que sin ellos nada de esto hubiera sido posible.

En especial a mis mejores amigos de la facu, Álvaro, Felipe y Mauro que fueron de gran soporte durante toda la carrera.

Muchas gracias a Ernesto, Federico y Joaquín, que me ayudaron muchísimo durante toda la preparación de este trabajo.

Índice general

Resumen

Con el avance de la tecnología y la computación, la simulación se ha vuelto muy popular en los últimos tiempos. Es utilizada en muchos ámbitos y es útil para el estudio exhaustivo y seguro en la experimentación de diversos sistemas que en la vida real son difíciles (o incluso imposibles) de estudiar, debido a costos económicos, poca practicidad, entre otros.

Simular algunos sistemas (debido a su tamaño o complejidad) puede presentar un alto costo computacional y no es deseable que se espere *demasiado tiempo* para obtener los resultados. Para reducir el tiempo de simulación, el modelo puede ser dividido en submodelos que pueden ser simulados cada uno en un núcleo de procesamiento diferente, aprovechando así la capacidad de los actuales procesadores multi-núcleo. Debido a que día a día la tecnología multi-tarea avanza fuertemente, paralelizar resulta ser un tema de gran interés en la actualidad.

El problema es que, cuando un sistema es particionado en subsistemas probablemente éstos no son independientes entre sí, por lo que es necesario algún tipo de sincronización entre las simulaciones de cada uno de ellos para el intercambio de la información necesaria en ciertos períodos de tiempo. En algunas ocasiones, estos particionados hacen que el sistema completo se vuelva *inestable* a medida que la ejecución avanza, es decir, el error introducido conlleva a una solución cualitativamente incorrecta. Este nivel de inestabilidad varía dependiendo de la naturaleza del sistema y de la forma en la que es particionado. El grupo de investigación encontró experimentalmente que realizando cómputos redundantes en cada una de las particiones, esta inestabilidad se reducía.

En el presente trabajo se realiza un estudio en profundidad del impacto de la utilización de cómputos redundantes o *nodos virtuales* en el contexto de simulación de sistemas continuos.

Para ello, se introduce un nuevo algoritmo que aplica la metodología de nodos virtuales a un modelo dado y lo particiona en submodelos para su simulación en paralelo. Este algoritmo se aplica en modelos que son simulados bajo una familia de métodos de cuantificación de estados, denominada QSS.

Se explicará el algoritmo de particionado y se mostrarán ejemplos de aplicación en algunos modelos.

Parte I

Preliminares

Capítulo 1

Introducción

1.1. Motivación

Con el avance de la tecnología y la computación, la simulación se ha vuelto muy popular en los últimos tiempos. Es utilizada en muchos ámbitos y es útil para el estudio exhaustivo y seguro en la experimentación de diversos sistemas que en la vida real son difíciles (o incluso imposibles) de estudiar, debido a costos económicos, poca practicidad, entre otros.

Podemos armar cualquier modelo que represente a un sistema (real o no real) y observar su comportamiento en un período finito de tiempo.

En muchos casos es interesante el estudio de modelos a gran escala[1] ya que, por lo general, son los más imprácticos a la hora de estudiarlos en el mundo real. Otro tipo de modelos son aquellos que interaccionan con alguna entidad externa (un usuario, por ejemplo), que pertenecen a la categoría de **simulaciones en tiempo real**.

Simular algunos sistemas (debido a su tamaño o complejidad) puede ser demasiado costoso y no es deseable que se espere *demasiado tiempo* para obtener los resultados. Para reducir el tiempo de ejecución, el modelo puede ser dividido en submodelos que pueden ser simulados cada uno en un núcleo de procesamiento diferente, aprovechando así el poder de procesamiento de los procesadores multi-núcleo actuales en su totalidad. Debido a que día a día la tecnología multi-tarea avanza fuertemente, paralelizar resulta ser un tema de gran interés en la actualidad.

1.2. Objetivo

Cuando un modelo es particionado en submodelos, existe un problema: probablemente éstos no son independientes entre sí (es decir, no son totalmente desconexos), por lo que es necesario algún tipo de sincronización

[1] Cuando hablemos de modelo a gran escala, nos referiremos a su tamaño a nivel "vectorial", no al tamaño de su descripción.

entre la ejecución de cada uno de ellos para el intercambio de la información necesaria en ciertos períodos de tiempo. En algunas ocasiones, estos particionados hacen que el sistema completo se vuelva *inestable* a medida que la ejecución avanza. El nivel de inestabilidad, según diversas pruebas realizadas, puede variar dependiendo de la definición del modelo y la forma en la que es particionado.

Mediante la experimentación en modelos bajo métodos de simulación basados en la cuantificación de estados (denominados QSS por sus siglas en inglés), encontramos que la inestabilidad antes mencionada puede reducirse mediante el uso de ciertas computaciones redundantes, que denominaremos *nodos virtuales*, constituídas por el procesamiento repetido de algunas variables de estado del modelo. Presentaremos entonces un método de particionado de modelos que utiliza estos nodos virtuales, con la intención de reducir la inestabilidad producida por las simulaciones en paralelo.

1.3. Marco Formal y Trabajo Previo

El trabajo se enmarca en el proyecto de Universidad y Transporte Nro 32-64-007, "Simulador de Entrenamiento para Conductores Ferroviarios". Durante el mismo, desarrollamos en parte el módulo funcional de un simulador de manejo de trenes con el fin de entrenar conductores ferroviarios. Como parte del desarrollo, realizamos experimentos con modelos simulados en paralelo bajo métodos de integración numérica clásicos como DASSL, y además aplicamos las técnicas de computaciones redundantes con *nodos virtuales*, observando que se obtenían así resultados más representativos que los modelos paralelizados que no las utilizaban. Es decir, se obtiene menor error con respecto a las simulaciones realizadas de forma paralela sin nodos virtuales.

1.4. Trabajo Relacionado

En la literatura del área existen algunos trabajos relacionados con la presente tesina.

Dentro del área de simulación paralela con métodos QSS, encontramos el trabajo de Maggio [13] y de Nutaro [15] como primera investigación a la simulación en paralelo con QSS. Los resultados obtenidos no fueron extendidos debido a las limitaciones de la arquitectura GPU utilizada. En [2], se presenta una técnica de paralelización para la simulación en DEVS de sistemas continuos e híbridos, denominada ASRTS, enfocada para una arquitectura de memoria compartida. Fernández [9] desarrolla una nueva metodología de simulación en paralelo bajo métodos QSS.

En [16] se presenta un método de particionado de modelos escritos en lenguaje Modelica para su simulación en paralelo, y se estudia su aplicación

sobre modelos de líneas de transmisión. Este trabajo no se basa en métodos QSS. Ninguno de los trabajos antes mencionados utiliza nodos virtuales.

Finalmente, fuera del área de simulación en paralelo, se introdujo en [6] la noción de *fluído fantasma* o *virtual* (ghost fluid) para mitigar el error de discretización de diferencias finitas.

Por lo tanto, el presente trabajo es el primer estudio de la utilización de nodos virtuales para la simulación en paralelo bajo métodos QSS.

Capítulo 2

Conceptos Previos

2.1. Sistemas y Modelos

Podemos definir a un **sistema** como un conjunto delimitado de entidades interactuantes. Bajo esta definición, podemos pensar como un sistema a prácticamente cualquier cosa. Por ejemplo, un celular, una computadora, un automóvil, un tren, un planeta, etc. Más de una vez vamos a experimentar sobre ellos para realizar ciertas observaciones, ya sea por necesidad o por curiosidad. ¿Pero qué sucede cuando experimentar sobre estos sistemas es impráctico o costoso? En el mundo real esto se puede dar por muchas razones:

- Imposibilidad temporal: queremos estudiar el comportamiento de un sistema a lo largo de mucho tiempo (por ejemplo, 10 años).
- Imposibilidad de recursos: conseguir el sistema o experimentar con él puede ser costoso monetariamente (por ejemplo, experimentar con un tren de carga).
- Riesgo: queremos analizar la explosión de una bomba de 50 megatones.
- El sistema no existe: recordar que podemos pensar como un sistema a cualquier cosa, incluso si no es real.

Cuando es imposible experimentar sobre un sistema real, se recurre a hacerlo sobre un **modelo del sistema**.

Un **modelo** es una representación simplificada de un sistema que intenta comportarse lo más similar posible a este último. De esta manera, uno puede experimentar sobre el modelo y obtener conclusiones sobre el sistema, sin tener que experimentar con él en el mundo real.

Las características de un sistema son descriptas en el modelo mediante **variables de estado** que van variando a medida que el tiempo avanza.

2.1.1. Tipos de modelos

Podemos clasificar a los sistemas mediante diferentes criterios. El que nos interesa a nosotros es **según la evolución temporal**, es decir, cómo las variables evolucionan con el tiempo:

- **Tiempo continuo:** las variables evolucionan continuamente con el tiempo. Por lo general son representadas con **ecuaciones diferenciales**.
- **Tiempo discreto:** las variables sólo cambian en determinados instantes de tiempo. Usualmente se representan con **ecuaciones en diferencias**.
- **Eventos discretos:** si bien las variables pueden cambiar en cualquier momento, sólo lo pueden hacer un número finito de veces en instantes de tiempo finitos.

También existen sistemas que combinan tiempo continuo con eventos discretos. Es decir, el tiempo evoluciona de forma continua, y posee algunas variables que evolucionan de forma continua y otras que cambian en determinados instantes de tiempo finitos. Estos sistemas generalmente se denominan **sistemas híbridos de tiempo continuo**.

El desarrollo de nuestro algoritmo aplica sobre este último tipo de sistemas, aunque no nos interesa la parte de eventos discretos; sólo es de nuestra importancia el estudio sobre las variables continuas.

2.2. Lenguaje Modelica

Las antiguas herramientas de modelado y simulación requerían que los modelos estén directamente escritos en algún lenguaje de programación como C o Fortran, lo cual es muy incómodo y casi imposible de realizar con modelos complejos y de gran escala.

Por lo tanto, se comenzaron a desarrollar lenguajes específicamente de modelado hasta que se surgió un lenguaje de modelado estándar, llamado Modelica [10]. Este es un lenguaje de alto nivel, orientado a objetos utilizado para el modelado de sistemas grandes y complejos. Los modelos son descriptos matemáticamente por ecuaciones diferenciales, algebraicas y discretas. Los submodelos pueden estar interconectados para construir modelos más complejos. Existen muchas herramientas de modelado que ofrecen la posibilidad de componer modelos de manera gráfica.

Por otro lado, tenemos una gran variedad de compiladores que convierten modelos construidos en Modelica a código estándar (como C), listo para ser compilado y ejecutado. Algunas herramientas de simulación basadas en Modelica son Dymola [4], OpenModelica [17], entre otros.

El siguiente es un ejemplo de un modelo escrito en Modelica que representa una pelota que cae y rebota contra una superficie:

```
1  model bball
2    Real y(start = 1);
3    Real v;
4    discrete Integer pique;
5    parameter Real k = 10000,b = 30,g = 9.8,m = 1;
6  equation
7    m * der(v) = if y > 0
8                 then -m * g
9                 else -m * g - k * y - b * v;
10   der(y) = v;
11 algorithm
12   when y < 0 then
13     pique:=1;
14   elsewhen y > 0 then
15     pique:=0;
16   end when;
17 end bball;
```

Código 2.1: Modelado de una pelota que cae y rebota contra una superficie.

Como podemos ver, el modelo consta de:

- Dos *variables* reales, denominadas **variables de estado**, que modelan ciertas características del sistema que queremos observar. Estas son variables *continuas*, es decir, cambian constantemente junto al tiempo.
- *Parámetros* reales, utilizados para configurar las caraterísticas del sistema. Durante la ejecución de una simulación, estos valores no cambian.
- Una variable real *discreta*, que cambia sólo en algunos instantes de tiempo.
- Una sección que contiene *ecuaciones diferenciales* que describen las relaciones entre las variables de estado del sistema.
- Una sección llamada `algorithm`: en ella se especifica cuándo la variable discreta cambiará su valor. Es decir, estamos especificando los *instantes de tiempo* en los cuales la variable discreta puede variar.

En la figura 2.1 podemos ver el resultado de simular el modelo durante 2 segundos[1]. Se muestra durante este intervalo de tiempo los valores de la

[1]Cabe aclarar que son 2 segundos de simulación, no 2 segundos de tiempo real de procesamiento.

variable de estado y, que en nuestro modelo representa a la altura de la pelota con respecto a la superficie.

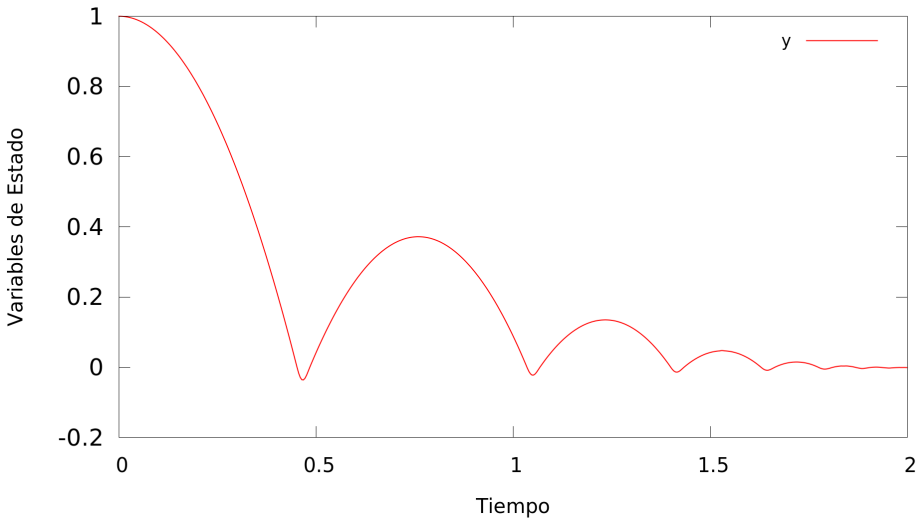

Figura 2.1: Resultado (variable y) de simulación del modelo bball (definido en el código 2.1) durante 2 segundos.

2.3. Simulación de Sistemas Continuos

Los sistemas continuos son unos de los más populares debido a que abarcan, entre otros, sistemas mecánicos, termodinámicos, electromagnéticos e hidráulicos. Las relaciones entre las variables de estos sistemas se describen con Ecuaciones Diferenciales Ordinarias (EDO's)[2]. Lo podemos describir formalmente como:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), t) \tag{2.1}$$

donde $\mathbf{x}$ es el vector de variables de estado y $\dot{\mathbf{x}}$ sus derivadas respecto al tiempo.

Por ejemplo, podemos modelar un sistema masa-resorte-amortiguamiento simple como el de la figura 2.2 con el siguiente sistema de ecuaciones:

$$\begin{aligned} \dot{\mathbf{x_1}}(t) &= \mathbf{x_2}(t) \\ \dot{\mathbf{x_2}}(t) &= \frac{1}{m}[-k\mathbf{x_1}(t) - b\mathbf{x_2}(t) + F(t)] \end{aligned} \tag{2.2}$$

[2]También denominadas **ODE** por sus siglas en inglés: Ordinary Diferential Equation.

Donde:

- $\mathbf{x_1}(t)$ representa la posición.
- $\mathbf{x_2}(t)$ representa la velocidad.
- $F(t)$ es la fuerza de entrada aplicada al sistema.
- Los parámetros son m (masa), b (coeficiente de roce) y k (constante del resorte).

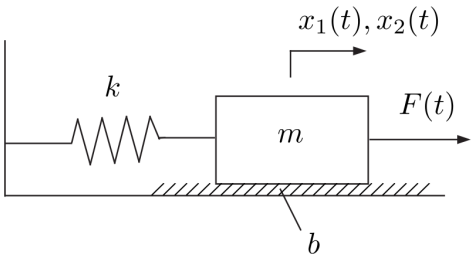

Figura 2.2: Sistema masa-resorte-amortiguación.

Para observar cómo evolucionan las variables a medida que avanza el tiempo, debemos resolver el sistema de ecuaciones 2.2. Si, por ejemplo, igualamos los parámetros m, b, k al valor 1, $F(t) = 1$, y si asumimos $\mathbf{x_1}(0) = \mathbf{x_2}(0) = 0$, la solución analítica es:

$$\begin{aligned} \mathbf{x_1}(t) &= 1 - \frac{\sqrt{3}}{3} e^{-t/2} \sin(\frac{\sqrt{3}}{2} t) - e^{-t/2} \cos(\frac{\sqrt{3}}{2} t) \\ \mathbf{x_2}(t) &= \frac{\sqrt{12}}{3} e^{-t/2} \sin(\frac{\sqrt{3}}{2} t) \end{aligned} \tag{2.3}$$

Para todo $t \geq 0$. Podemos ver la gráfica correspondiente en la figura 2.3.

Sin embargo, para un sistema dado, lo más probable es no podamos obtener la solución analítica de su sistema de ecuaciones diferenciales debido a la complejidad, impracticidad e incluso imposibilidad para resolverlas. Cuando se presentan ecuaciones no lineales, exceptuando algunos pocos casos, no es posible encontrar una solución analítica.

Por ende, se acude a la utilización de *métodos de integración numérica* que bajo ciertas condiciones brindan soluciones *aproximadas*, suficientemente representativas para el sistema en estudio.

Hay varios métodos que brindan mejores y peores soluciones que otros, dependiendo de la naturaleza del sistema. Entre ellos podemos encontrar a los métodos de integración numérica clásicos como Euler [3], DASSL [18] y Runge-Kutta [5], y a los métodos de cuantificación de estado como los QSS [3].

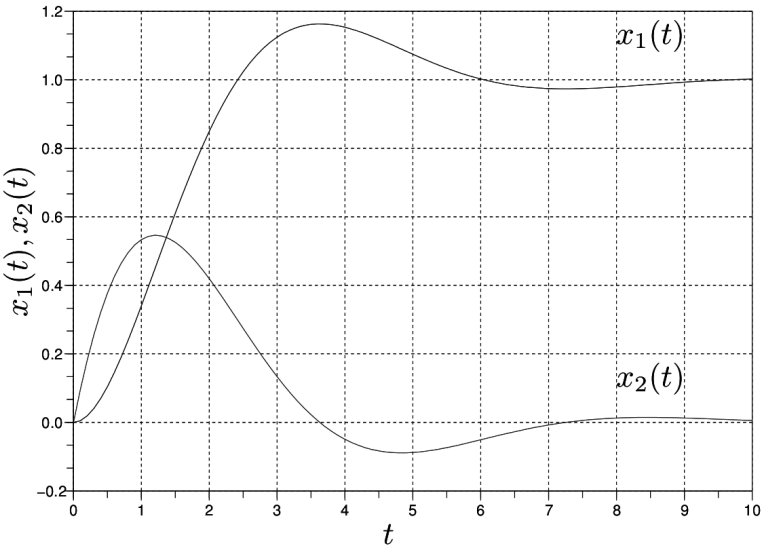

Figura 2.3: Gráfica de la solución de la ecuación (2.3).

2.4. Métodos de Cuantificación de Estados

Los métodos de resolución de EDO's convencionales discretizan el tiempo y calculan $\mathbf{x}(t)$ para cada valor discreto de t, utilizando integración numérica sobre $\dot{\mathbf{x}}$.

Los Métodos de Cuantificación de Estados, denominados métodos **QSS** son una familia de métodos de resolución de EDO's que reemplazan la discretización del tiempo mediante la cuantificación de estados, dando como resultado un modelo de simulación de eventos discretos, en lugar de tiempo discreto.

Mientras los métodos convencionales intentan responder la pregunta:

> Si el estado en el tiempo t_k tiene valor x, ¿cuál es el valor del estado en el instante de tiempo $t_{k+1} = t_k + \Delta t$?

Los Métodos de Cuantificación de Estados responden a la pregunta:

> Si el estado tiene un valor de x_k en el instante de tiempo t, ¿cuál es el instante de tiempo más próximo en el que el estado asume un valor de $x_{k\pm1} = x_k \pm \Delta x$?

Cabe aclarar que estos métodos no convierten sistemas de EDO's en sistemas de ecuaciones equivalentes, sino que convierten un modelo de tiempo continuo a un modelo de eventos discretos equivalente, el cual puede ser simulado con cualquier motor de simulación basado en eventos discretos, como DEVS [1].

De esta forma, la simulación con los métodos QSS en sistemas con ciertas características es mucho más rápida y estable que las realizadas con los métodos convencionales. Algunos de estos sistemas son: sistemas con muchas discontinuidades, sistemas *stiff* y sistemas *marginalmente estables*. Además, se estableció en [12] que un sistema analíticamente estable no puede tornarse numéricamente inestable si es simulado con un método QSS. Puede suceder que el algoritmo produzca oscilaciones de alta frecuencia, lo que equivale a simular utilizando un paso muy pequeño con los algoritmos convencionales.

2.5. Simulador Autónomo de QSS

Como mencionamos anteriormente, los métodos QSS convierten un modelo basado en EDO's a un modelo de eventos discretos equivalente, el cual puede simularse con cualquier motor de simulación basado en eventos discretos, como por ejemplo DEVS.

Las primeras implementaciones de estos métodos, por lo tanto, fueron implementados sobre el formalismo DEVS. El problema es que las implementaciones de las herramientas basadas en este formalismo son ineficientes debido a que se malgasta gran parte de la carga computacional en la transmisión de eventos entre submodelos.

Por tal motivo, en [8] se presenta una implementación de un simulador QSS autónomo apto para simular modelos con los métodos QSS de forma eficiente.

Un problema de estos métodos es que necesitan información estructural del modelo. Cada paso implica un cambio en alguna de las variables de estado y en las derivadas de estado que dependen de ella. Por lo tanto, el modelo debe proveer, además de las expresiones de las derivadas de estado (como en los sistemas EDO), una matriz de incidencia para saber cuáles de las derivadas de estado cambian en cada paso. Sería algo muy incómodo pedir al usuario que brinde esta matriz de incidencia. Además el tamaño de ésta crece a medida que el modelo lo hace, por lo que mientras más grande sea el modelo, más probabilidades hay de ingresar un dato erróneo en ella. Para evitar este problema, la implementación posee un Front-End que automáticamente obtiene la matriz de incidencia desde un modelo definido normalmente. De esta manera el usuario puede describir los modelos utilizando el lenguaje de modelado μ-Modelica [7], el cual es un subconjunto de Modelica. Dado un modelo de entrada, se generará automáticamente su código C incluyendo la estructura.

Además, la implementación ofrece una GUI[3] que integra al solver con el Front-End, incluyendo también algunas herramientas de debug y de impresión de resultados (plotting).

[3]Interfaz Gráfica de Usuario. Llamada así por su sigla en inglés: Graphic User Interface.

Se demostró la superioridad de estos métodos (implementados en el simulador autónomo) sobre algunos métodos numéricos clásicos como DASSL y Runge-Kutta en algunos modelos con ciertas características particulares. En [8] se pueden encontrar algunos experimentos realizados con fines comparativos.

Actualmente la implementación se ha extendido (todavía en forma experimental) para soportar la simulación en paralelo. Se puede indicar la cantidad de procesadores a utilizar, y el particionado del modelo se realiza de forma automática si se desea, aunque se puede realizar manualmente, indicando para cada variable del modelo en qué procesador la queremos computar. Utilizaremos esta característica para el estudio de nuestro método de paralelización.

2.6. μ-Modelica

Como mencionamos anteriormente, el lenguaje de modelado μ-Modelica [7] es un subconjunto del lenguaje Modelica (presentado en la sección 2.2). Su especificación define lo mínimo indispensable para poder modelar sistemas híbridos basados en EDO's.

Presenta las siguientes restricciones respecto a Modelica:

- Sólo se puede definir una clase (no se permite herencia de clases). Por lo tanto el modelo que se construye es *plano*.
- Las variables son de tipo `Real`. Pueden ser de estado o algebraicas por un lado, y continuas o discretas por el otro.
- Los parámetros son solamente de tipo `Real`.
- Las variables de tipo arreglo son sólo unidimensionales[4].
- Los índices en los arreglos dentro de cláusulas `for` deben ser de la forma $\alpha \cdot i + \beta$, donde α y β son expresiones enteras e `i` es el índice de la iteración.
- La sección de ecuaciones se compone de:
  - Definición de variables de estado: cada una de ellas se define brindando su EDO correspondiente, de la forma `der(x) = EXP`. Además, todo lo que aparezca en `EXP` debe estar definido anteriormente.
  - Definición de variables algebraicas de la forma `x = EXP`. Cada variable algebraica sólo puede depender de variables de estado y de variables algebraicas que ya fueron definidas.

[4] Aunque en un futuro se planea extender el formalismo para aceptar arreglos de cualquier dimensión.

- Las discontinuidades se definen sólo con las cláusulas `when` y `elsewhen` dentro de la sección `algorithm`. Las condiciones dentro de las dos cláusulas sólo pueden ser relaciones $<$, $\leq$, $>$ y $\geq$. Dentro de estas cláusulas se permiten solamente asignaciones de variables discretas y sentencias `reinit`'s de variables de estado (continuas).

Estas restricciones son necesarias ya que el simulador autónomo QSS necesita conocer la estructura del sistema para, cada vez que un estado cambia, evaluar sólo las variables que dependen de ella.

2.7. Simulación en Paralelo

Anteriormente se describió la forma de modelar sistemas continuos mediante EDO's, planteada en la ecuación (2.1).

Muchas veces, la complejidad computacional de evaluar la función $\mathbf{f}$ y de ejecutar el método de integración numérica puede ser muy elevada para sistemas de gran escala o complejos. Esto puede mitigarse aplicando técnicas de paralelización a los algoritmos de simulación. Esto se conoce como *paralelización funcional*. Para ello, el modelo de la ecuación (2.1) debe ser particionado en submodelos, los cuales deben simularse concurrentemente en distintos procesadores.

Generalmente este particionado no puede realizarse en partes totalmente desconexas dado que los distintos submodelos tendrán dependencias entre ellos. Por ejemplo, el cálculo de una variable en un procesador requiere del valor de una variable que es computada en otro procesador diferente. Por lo tanto, las simulaciones que se ejecutan en cada procesador no pueden ser totalmente independientes sino que deben sincronizarse de alguna forma. Existen distintos algoritmos para coordinar estas sub-simulaciones, como CMB [14], TimeWarp [11] y NoTIME [19].

2.7.1. Método de paralelización SRTS - ASRTS

En [2], se introduce un nuevo método de paralelización para la simulación en DEVS de sistemas continuos e híbridos, denominada **SRTS**[5]. El modelo es dividido en submodelos que son simulados concurrentemente en diferentes procesadores. Para evitar el costo de la sincronización global de todos los procesos, el tiempo de simulación de cada submodelo se sincroniza localmente con el tiempo real del sistema. De esta manera, todos los procesos se sincronizan de manera implícita.

Si bien en la implementación no se asegura una sincronización perfecta, bajo ciertas condiciones el *error de sincronización* introducido sólo provoca errores numéricos acotados en los resultados de simulación.

[5]Debido a su significado en inglés, Scaled Real-Time Synchronization (Sincronización de Tiempo Real Escalado).

El método SRTS utiliza el mismo parámetro de "escalado" de tiempo real físico durante toda la simulación. Por lo tanto, se deriva de él una versión adaptativa, denominada **ASRTS**[6], donde este parámetro cambia automáticamente durante la simulación dependiendo de la carga de cómputo del sistema.

Aunque en [2] los métodos SRTS y ASRTS son implementados y probados sobre las implementaciones en DEVS de los métodos QSS, actualmente se está concluyendo su desarrollo en la implementación del simulador autónomo QSS presentada en [8]. Nuestros experimentos se realizán entonces en esta última plataforma.

[6]Llamado también Adaptive-SRTS (SRTS Adaptativo).

Parte II

Contribuciones

Capítulo 3

Paralelización con Nodos Virtuales

Durante el desarrollo de ASRTS, se estudió superficialmente la idea de reducir el error introducido durante la paralelización por esta técnica a través de computaciones redundantes o *nodos virtuales*. La idea se basa en particionar el modelo de forma que algunas de sus variables sean computadas repetidas veces por varios procesadores. Se observó experimentalmente en algunos modelos que esta redundancia suaviza el error introducido por la falta de sincronización.

En este capítulo, introducimos formalmente la idea de nodos virtuales para simulación en paralelo. Luego, presentamos un algoritmo que se encarga de insertar los nodos virtuales en un modelo dado, para luego particionarlo. Finalmente se presenta un fundamento sobre cómo los nodos virtuales reducen el error introducido por la comunicación entre los submodelos.

La técnica de utilización de nodos virtuales está enfocada a modelos escritos en μ-Modelica, el cual es, como dijimos anteriormente, un subconjunto del lenguaje Modelica. Además, en la implementación del simulador autónomo QSS que soporta simulaciones en paralelo, el modelo puede ser particionado de forma manual, indicando en cuántas partes se lo desea particionar, y en qué procesador debe ser computada cada variable. De esta forma, podemos alojar los nodos virtuales en los extremos de comunicación de cada submodelo.

3.1. Cómputos Redundantes con Nodos Virtuales

Como mencionamos anteriormente, cuando un modelo es particionado en submodelos, es muy probable que existan ciertas dependencias entre ellos. Es decir, algunas variables de algún submodelo necesitarán los valores de variables alojadas en otro submodelo diferente para realizar sus correspondientes cómputos (esto es, calcular sus ecuaciones). Por lo tanto, estos submodelos

deben estar sincronizados y entre ellos deben comunicarse los valores de las variables que cada submodelo necesita.

Entonces, el particionado junto a la sincronización de los submodelos introducen errores en el cálculo de las variables.

Explicaremos el algoritmo sobre un ejemplo. El siguiente es un modelo de una línea de transmisión formada por `N` secciones de circuitos RLC:

```
1  model rlc_line
2    constant Integer N = 1000;
3    parameter Real r = 0.001;
4    Real x[N];
5    discrete Real d (start = 1);
6  equation
7    der(x[1]) = d - x[2];
8
9    for i in 2:N-1 loop
10     der(x[i]) = x[i-1] - x[i+1] - r * x[i];
11   end for;
12
13   der(x[N]) = x[N-1] - 10000 * x[N];
14 algorithm
15   when time > 1 then
16     d := 0;
17     elsewhen time <= 1 then
18     d := 1;
19   end when;
20 end rlc_line;
```

Código 3.1: Modelo de línea de transmisión.

De esta forma representamos un cable por el que se introduce, desde un extremo, un pulso de voltaje. Éste, a medida que evoluciona el tiempo, va avanzando por el cable disminuyendo su amplitud progresivamente. Cuando llega al otro extremo, "rebota" y vuelve hacia atrás. Este proceso se realiza reiteradamente hasta que desaparece a causa de la reducción constante en su amplitud de onda. Como podemos ver, con la variable de tipo arreglo `x` de longitud `N` (declarada en la línea 4 del código) representamos al cable, mientras que la variable real discreta `d` (declarada en la línea 5) representa al pulso de voltaje que ingresa por un extremo del cable (`x[1]` en este caso). De esta forma obtenemos un modelo *discretizado* de una línea de transmisión, ya que se piensa al cable como una secuencia finita de secciones conectadas como muestra la figura 3.1.

Analicemos las ecuaciones de la variable `x`. Dejando de lado las ecuaciones de `x[1]` y `x[N]` (líneas 7 y 13 respectivamente), prestamos atención en las ecuaciones de `x[2]` ... `x[N-1]` definidas por la cláusula `for` en las líneas

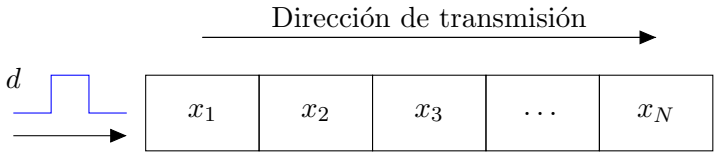

Figura 3.1: Esquema del modelo 3.1 de línea de transmisión.

9–11 del código. Podemos ver que, $\forall i \in 2 \ldots N-1$ la ecuación de `x[i]` *depende* de `x[i-1]` y `x[i+1]`. Es decir, el cálculo del valor de la variable `x[i]` necesita los valores de sus variables "contiguas", además de sí misma.

Si queremos simular este modelo en paralelo, debemos particionarlo. Debido a las restricciones del lenguaje μ-Modelica, la única manera simple de construir modelos a gran escala es mediante la utilización de variables de tipo arreglo de tamaño considerable (como lo es la variable `x` en el modelo de línea de transmisión). Por lo tanto, nuestro algoritmo de particionado se enfocará en las variables de este tipo que definen sus ecuaciones mediante cláusulas `for`.

En este ejemplo particionaremos el modelo en dos partes, con la intención de poder simularlo paralelamente en dos procesadores. Tomando un criterio simple (y para alojar la misma cantidad de variables por procesador), el *punto de corte* será en el valor `N/2`: las variables `x[1]`, ..., `x[N/2-1]` se alojarán en el procesador 1, mientras que las variables `x[N/2]`, ..., `x[N]` lo harán en el procesador 2. Decimos "alojar" para indicar en qué procesador se realizarán los cálculos correspondientes de una variable determinada. Para simplificar la lectura, llamemos `C=N/2`. De esta manera, las variables `x[1]`, ..., `x[C-1]` se alojan en el procesador 1, y las variables `x[C]`, ..., `x[N]` se alojan en el procesador 2.

En la figura 3.2 esquematizamos la distribución de las variables en los dos procesadores. Los datos enviados entre procesadores son valores de variables *ya computados* en su procesador correspondiente. El intercambio de datos entre procesadores sucede durante la sincronización entre ellos, la cual se realiza en determinados períodos –finitos– de tiempo. Al utilizar métodos de cuantificación de estados, la transmisión de mensajes se produce sólo cuando las variables en cuestión cambian de estado.

Entonces, en cada sincronización entre procesos ocurre lo siguiente (entre otras cosas):

- El procesador 1 recibe el *valor* de la variable `x[C]` (computada en el procesador 2), para poder calcular el valor de la variable *local* `x[C-1]`.
- El procesador 2 recibe el *valor* de la variable `x[C-1]` (computada en el procesador 1), para poder calcular el valor de la variable *local* `x[C]`.

Como la sincronización entre procesos se realiza un número finito de veces, puede ocurrir que las variables que –para su cálculo– dependen de otras

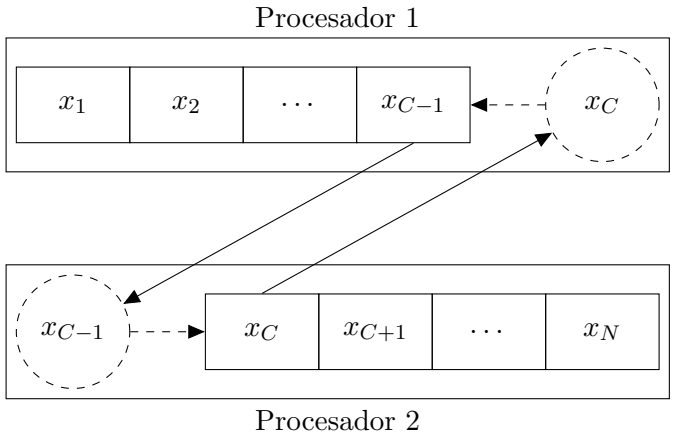

Figura 3.2: Particionado en dos procesadores del modelo 3.1 de línea de transmisión. Con un rectángulo se representan las variables del modelo, con un círculo de borde punteado, los datos recibidos desde otro procesador, con una flecha punteada indicamos dónde es utilizado el dato y con una flecha sólida se refleja el intercambio de datos entre procesos.

que se computan en procesadores diferentes, podrían estar utilizando valores *desactualizados* de ellas en el procesador *local*. Esto se debe a que los valores de las variables utilizados son los obtenidos de la última sincronización; si estas cambian de estado, los nuevos valores no llegarán hasta la próxima sincronización. En este punto, las computaciones de las variables adquieren cierto error con respecto a la simulación realizada de forma secuencial, sin paralelizar.

En otras palabras: si una variable computada en un procesador determinado produce un cambio en su estado, este no se verá reflejado en los demás procesadores hasta que la próxima sincronización ocurra.

En la figura 3.3 podemos ver un ejemplo del problema anteriormente mencionado.

La idea del método de paralelización presentado es, no sólo obtener los valores de las variables `x[C]` y `x[C-1]` para los procesadores 1 y 2 respectivamente, sino también obtener sus *ecuaciones* de modo de poder actualizar sus valores en caso de que sea necesario sin tener que esperar hasta la próxima sincronización para saber si hubo un cambio en ellos, y reducir de esta forma la diferencia al valor actual de los transmitidos durante las sincronizaciones entre procesos.

Para ello, introducimos en el modelo dos variables: `virt1` en el procesador 1 y `virt2` en el procesador 2, representando a las variables `x[C]` y `x[C-1]` respectivamente. Estas nuevas variables las llamaremos *nodos virtuales*.

Necesitamos entonces obtener las ecuaciones de `x[C]` y `x[C-1]`. Las obtenemos instanciando la ecuación `for` en los valores `C` y `C-1`:

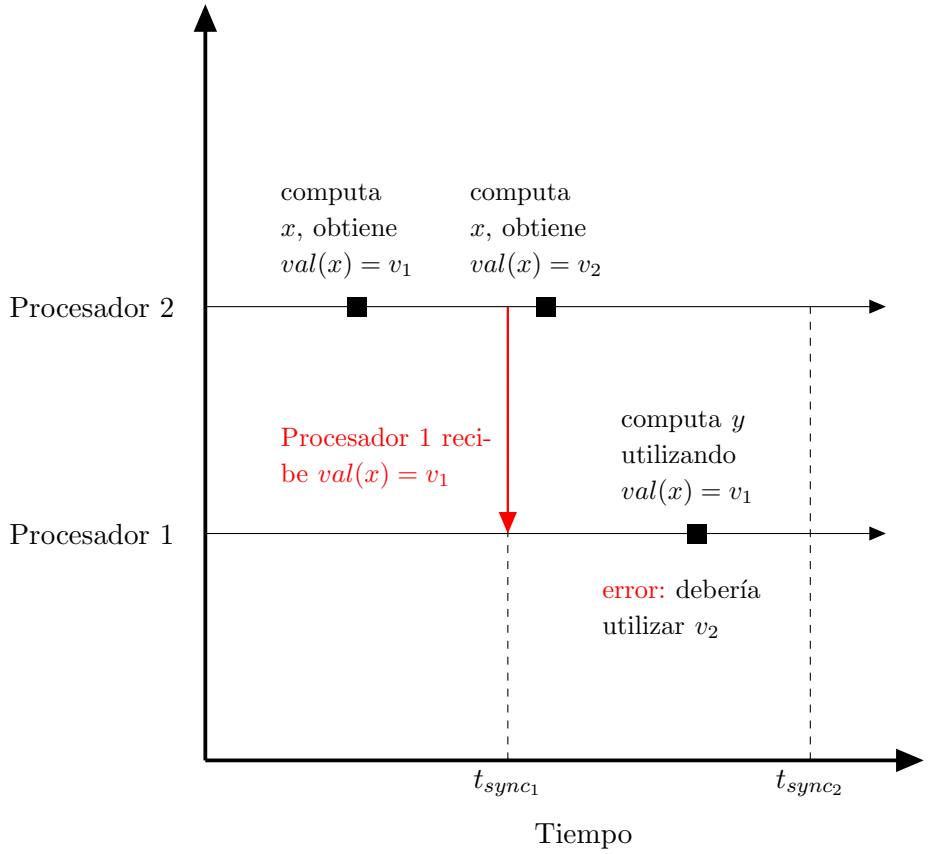

Figura 3.3: Ejemplo de pérdida de datos cuando se simula en paralelo. Suponemos que la variable y se aloja en el procesador 1, mientras que la variable x se aloja en el procesador 2. Además, y *depende* de x. En este escenario, el procesador 1 realiza el cómputo de la variable y utilizando v_1 como el valor actual de la variable x, cuando en realidad en ese instante de tiempo el valor de x es igual a v_2. El procesador 1 obtendrá el valor v_2 recién en el instante de tiempo t_{sync_2}, pero será demasiado tarde puesto que el cálculo de y ya se realizó. Por lo tanto, el procesador 1 descartará el valor v_2 cuando lo recibe en el instante de tiempo en t_{sync_2}. En este punto el cálculo de la variable y posee un error en comparación al modelo simulado de forma secuencial.

```
der(x[C]) = x[C-1] - x[C+1] - r * x[C];
der(x[C-1]) = x[(C-1)-1] - x[(C-1)+1] - r * x[C-1];
```

Lo cual equivale, reduciendo los índices:

```
der(x[C]) = x[C-1] - x[C+1] - r * x[C];
der(x[C-1]) = x[C-2] - x[C] - r * x[C-1];
```

Luego debemos reemplazar cada ocurrencia de `x[C]` por `virt1` en la primera ecuación, y `x[C-1]` por `virt2` en la segunda ecuación:

```
der(virt1) = x[C-1] - x[C+1] - r * virt1;
der(virt2) = x[C-2] - x[C] - r * virt2;
```

Finalmente introducimos un último término que asegura que cada variable virtual converja a la variable a la cual representa:

```
der(virt1) = x[C-1] - x[C+1] - r * virt1
                             + K * (x[C] - virt1);
der(virt2) = x[C-2] - x[C] - r * virt2
                           + K * (x[C-1] - virt2);
```

Donde `K` es un parámetro de tipo `Real`, al cual llamaremos *parámetro de convergencia de nodos virtuales* y su valor puede ser configurado según preferencia y necesidad. A medida que `K` crece, se espera una convergencia más rápida (aunque no necesariamente mejores resultados).

El nuevo esquema se muestra en la figura 3.4 y el código del nuevo modelo se presenta a continuación:

```
1  model rlc_line
2    constant Integer N = 1000;
3    parameter Real r = 0.001;
4    Real x[N];
5    discrete Real d (start = 1);
6    constant Integer C = N/2;
7    parameter Real K = 1;
8    Real virt1, virt2;
9  initial algorithm
10   virt1 := x[C];
11   virt2 := x[C-1];
12 equation
13   der(x[1]) = d - x[2];
14
15   for i in 2:C-2 loop
16     der(x[i]) = x[i-1] - x[i+1] - r * x[i];
17   end for;
18
19   der(virt1) = x[C-1] - x[C+1] - r * virt1;
20                                + K * (x[C] - virt1);
21
22   der(x[C-1]) = x[C-2] - virt1 - r * x[C-1];
```

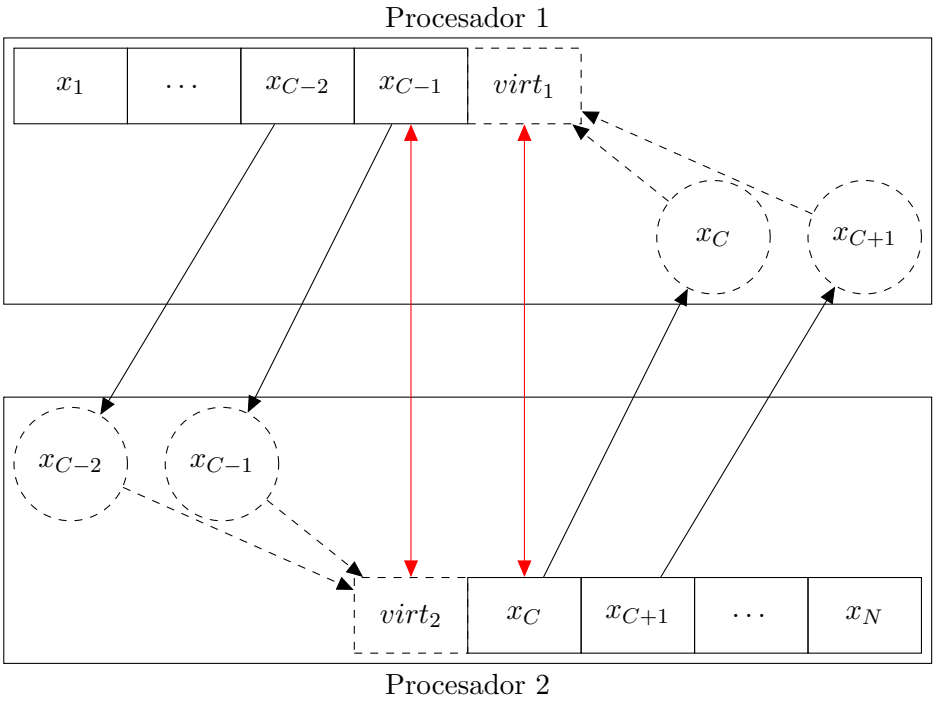

Figura 3.4: Particionado en dos partes del modelo de línea de transmisión 3.1, utilizando *nodos virtuales*. En adición a la figura 3.2, representamos con rectángulo de bordes punteados a los *nodos virtuales*, y con una flecha roja a la correspondencia entre el nodo virtual y la variable a la cual representa.

```
23
24    der(virt2) = x[C-2] - x[C] - r * virt2
25                                + K * (x[C-1] - virt2);
26
27    der(x[C]) = virt2 - x[C+1] - r * x[C];
28
29    for i in C+1:N-1 loop
30      der(x[i]) = x[i-1] - x[i+1] - r * x[i];
31    end for;
32
33    der(x[N]) = x[N-1] - 10000 * x[N];
34  algorithm
35    when time > 1 then
36      d := 0;
37      elsewhen time <= 1 then
38      d := 1;
39    end when;
40  end rlc_line;
```

Código 3.2: Modelo resultante del particionado en dos partes del modelo de línea de transmisión 3.1 utilizando nodos virtuales.

Si prestamos atención a las ecuaciones de `x[C-1]` y `x[C]`, veremos que ahora utilizan a los *nodos virtuales* `virt1` (que representa a `x[C]`) y `virt2` (que representa a `x[C-1]`) respectivamente.

Por otro lado, se puede ver que también se agregó el siguiente fragmento de código:

```
initial algorithm
  virt1 := x[C];
  virt2 := x[C-1];
```

el cual se encarga de asignar los valores iniciales de los nodos virtuales a los valores de las variables que cada uno representa. Tanto en Modelica como en μ-Modelica, el fragmento `initial algorithm` tiene como propósito realizar ciertos ajustes antes de comenzar la simulación. Las asignaciones de los valores iniciales de los nodos virtuales recién destacadas son un ejemplo de ello.

Analicemos el esquema de la figura 3.4. Si lo comparamos con el esquema de la figura 3.2, veremos que cada procesador tiene una variable más: `virt1` en el procesador 1, `virt2` en el procesador 2. Si bien el modelo resultante es entonces más complejo ya que tenemos dos nuevas variables, su presencia es despreciable antes las `1000` variables del modelo. Por lo tanto, esta cuestión es de poca relevancia en modelos a gran escala. Por otro lado, el intercambio de datos entre procesadores se duplicó en este caso: cada uno envía 2 datos y recibe 2 datos. Esto se debe a que, para calcular el valor de `virt1` (que representa a la variable `x[C]`) en el procesador 1, se necesitan los valores de las variables `x[C-1]`, `x[C+1]` y `x[C]`; mientras que la variable `x[C-1]` ya se encuentra en este procesador, las variables `x[C+1]` y `x[C]` son calculadas en el procesador 2. El razonamiento se aplica análogamente para el nodo virtual `virt2`. Este incremento en la carga de datos intercambiada entre procesadores supone una sincronización ligeramente más lenta.

3.1.1. Nivel de Solapamiento

Anteriormente hemos aplicado el método de particionado con *nodos virtuales* al modelo de línea de transmisión escrito en el código 3.1. Para ello, hemos analizado el `for` que define las ecuaciones de la variable `x`, planteamos una partición para esta variable e ingresamos dos nodos virtuales `virt1` alojado en el procesador 1 y `virt2` alojado en el procesador 2 que representan a las variables `x[C]` y `x[C-1]` respectivamente.

Analicemos la ecuación del nodo virtual `virt1` el cual es computado en el procesador 1. Vemos que para su cálculo depende de, por un lado, las variables `x[C-1]`, `x[C-1]` y `virt1` (ella misma) que son computadas en el mismo procesador. Pero también depende de `x[C+1]`, la cual es computada en el procesador 2. Podemos repetir el mismo procedimiento que aplicamos anteriormente, creando un nuevo nodo virtual, llamémoslo `virt1_b`, que represente a esta variable. Análogamente analizamos la ecuación del nodo virtual `virt2` que se computa en el procesador 2, y creamos un nuevo nodo virtual `virt2_b` para representar a la variable `x[C-2]` que es computada en el procesador 1. La cantidad de veces en la que es aplicado el procedimiento se denomina **nivel de solapamiento**. De esta forma, podemos decir que el modelo 3.2 se obtuvo al aplicar el algoritmo de particionado con nodos virtuales sobre el modelo de línea de transmisión 3.1 con nivel de solapamiento igual a 1.

Puede ocurrir que en cada procesador se puede crear más de un nodo virtual por cada iteración del algoritmo. Esto se da cuando las variables dependen, para su cálculo, de valores de dos o más variables que son computadas en un procesador diferente.

En la figura 3.5 podemos comparar el particionado del modelo de línea de transmisión 3.1 en dos partes, utilizando nodos virtuales con niveles de solapamiento 1 y 2.

A continuación se muestra el código 3.3, resultado de aplicar el algoritmo de particionado en dos partes sobre el modelo de línea de transmisión 3.1, utilizando nodos virtuales con nivel de solapamiento 2.

```
1  model rlc_line
2    constant Integer N = 1000;
3    parameter Real r = 0.001;
4    Real x[N];
5    discrete Real d (start = 1);
6    constant Integer C = N/2;
7    parameter Real K = 1;
8    Real virt1, virt2;
9    Real virt1_b, virt2_b;
10 initial algorithm
11   virt1 := x[C];
12   virt1_b := x[C+1];
13   virt2 := x[C-1];
14   virt2_b := x[C-2];
15 equation
16   der(x[1]) = d - x[2];
17
18   for i in 2:C-2 loop
19     der(x[i]) = x[i-1] - x[i+1] - r * x[i];
```

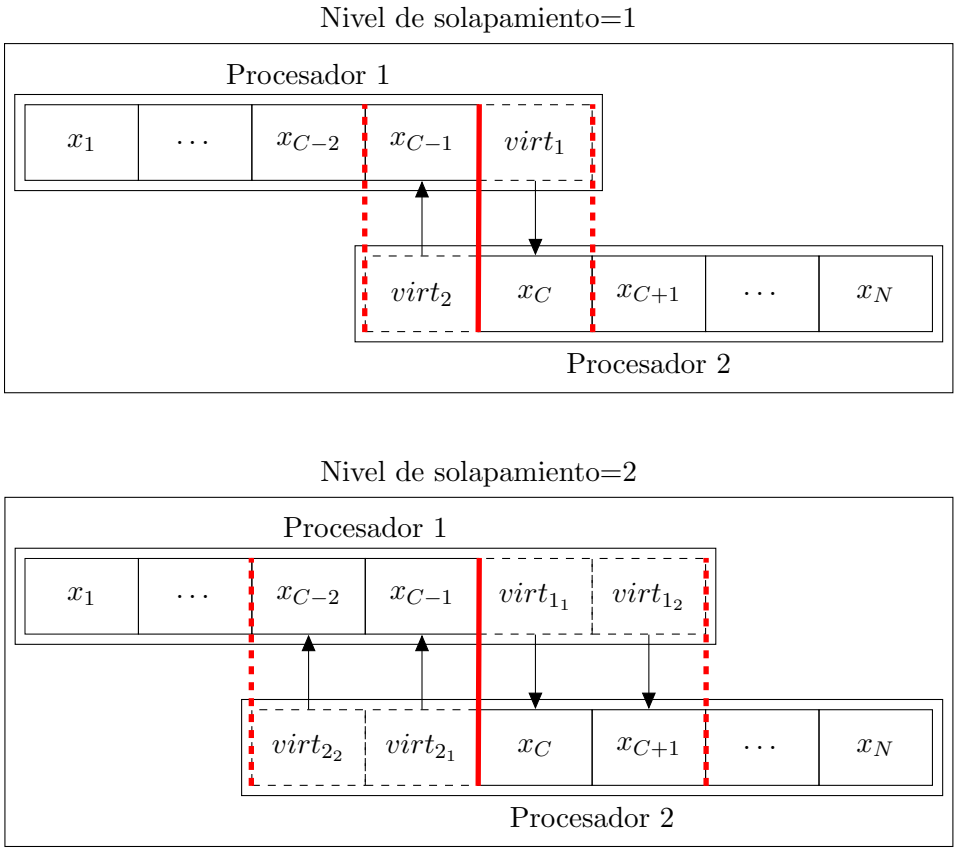

Figura 3.5: Comparación de particionados del modelo de línea de transmisión 3.1 en dos partes, utilizando nodos virtuales con niveles de solapamiento 1 y 2.

```
20    end for;
21
22    der(virt1_b) =
23      virt1 - x[C+2] - r * virt1_b
24                     + K * (x[C+1] - virt1_b);
25
26    der(virt1) =
27      x[C-1] - virt1_b - r * virt1
28                       + K * (x[C] - virt1);
29
30    der(x[C-1]) = x[C-2] - virt1 - r * x[C-1];
31
32    der(virt2_b) =
33      x[C-3] - virt2 - r * virt2_b
34                     + K * (x[C-2] - virt2_b);
```

```
35
36    der(virt2) =
37      virt2_b - x[C] - r * virt2
38                     + K * (x[C-1] - virt2);
39
40    der(x[C]) = virt2 - x[C+1] - r * x[C];
41
42    for i in C+1:N-1 loop
43      der(x[i]) = x[i-1] - x[i+1] - r * x[i];
44    end for;
45
46    der(x[N]) = x[N-1] - 10000 * x[N];
47  algorithm
48    when time > 1 then
49      d := 0;
50      elsewhen time <= 1 then
51      d := 1;
52    end when;
53  end rlc_line;
```

Código 3.3: Modelo resultante del particionado en dos partes del modelo de línea de transmisión 3.1, utilizando nodos virtuales con nivel de solapamiento 2.

3.2. Particionado del Modelo

Previamente hemos particionado el modelo de línea de transmisión 3.1 en dos partes, utilizando *nodos virtuales* con niveles de solapamiento 1 y 2. Cabe aclarar que el particionado de modelos tiene sentido cuando éstos son lo suficientemente grandes, ya que el costo introducido por particionar afecta en mayor proporción a modelos pequeños. Como explicamos anteriormente, este algoritmo de particionado se enfoca en las variables de tipo arreglo que definen sus ecuaciones mediante cláusulas `for`, ya que con ellas se construyen modelos a gran escala en donde probablemente existan dependencias entre sus componentes.

En el lenguaje μ-Modelica, las ecuaciones diferenciales de las variables de tipo arreglo se definen con cláusulas `for`, por cuestiones de practicidad y eficiencia a la hora de cargar el modelo en las herramientas de simulación. Esto lo podemos ver en el modelo 3.1 utilizado como ejemplo anteriormente, en donde definir a mano las ecuaciones diferenciales de cada componente de la variable `x` (sin utilizar `for`) es totalmente impráctico.

A continuación describimos el pseudoalgoritmo de particionado generalizado para cualquier número de particiones y número de solapamiento.

Primero, definimos inicialmente algunas variables globales:

$nProc \leftarrow numero\_particiones$
$K \leftarrow parametro\_K\_convergencia$
$nSolap \leftarrow numero\_de\_solapamiento$
$eqs \leftarrow \emptyset$
$virtualesLocales \leftarrow \emptyset$
$totalVirtuales \leftarrow \emptyset$
$partActual \leftarrow 0$
$rangoPart \leftarrow (0, 0)$

Las variables $nProc$, K y $nSolap$ representan a, como describe el código, el **número de particiones** en las que se quiere particionar el modelo, el **parámetro K de convergencia** y el **número de solapamiento**, respectivamente. Éstos, junto al modelo, son los **parámetros de entrada del algoritmo**. Se debe cumplir que $nProc > 0$ y $nSolap \geq 0$, aunque no tendría mucho sentido utilizar $nProc = 1$. La variable eqs representa a un conjunto que contendrá las nuevas ecuaciones del modelo resultante; $totalVirtuales$ es un conjunto que contendrá los nodos virtuales insertados en todo el modelo; $virtualesLocales$ es un conjunto utilizado auxiliarmente, indicando por cada partición el entorno de nodos virtuales ya creados; $partActual$ es una variable auxiliar utilizada para indicar en qué procesador se ejecutarán las variables (ocasionalmente sus valores cambiarán desde 0 a $nProc - 1$); $rangoPart$ es una variable temporal que indica el rango que define la nueva subecuación `for` que está siendo procesada.

La función principal es NODOSVIRTUALES:

```
1:  function NODOSVIRTUALES(modelo)
2:      eqsModelo ← OBTENEREQS(modelo)
3:      eqs ← eqsModelo
4:      for all ecuacion ∈ eqsModelo do
5:          if TIPOEQ(ecuacion) = FOR then
6:              PROCESARFOR(ecuacion)
7:          end if
8:      end for
9:      nuevoModelo ← CREARNUEVOMODELO(modelo, eqs)
10:     AGREGARDECLARACIONES(nuevoModelo, totalVirtuales)
11:     AGREGARASIGNACIONESINICIALES(nuevoModelo, totalVirtuales)
12:     return nuevoModelo
13: end function
```

Esta función toma un modelo de entrada, obtiene sus ecuaciones y procesa (llamando a la función PROCESARFOR) las ecuaciones de tipo `for`, que en el modelo se escriben como:

```
for i in rango_init:rango_end loop
    ...
end for;
```

Luego de realizar todo el procesamiento, crea un nuevo modelo con las ecuaciones obtenidas y agrega las declaraciones y asignaciones iniciales de los nodos virtuales a las variables que ellos representan (estas asignaciones se realizan en la sección `initial algorithm` como vimos anteriormente). Finalmente se devuelve el nuevo modelo particionado.

La función PROCESARFOR se describe a continuación:

1: **procedure** PROCESARFOR($eqFor$)
2: $(forInit, forEnd) \leftarrow$ OBTENERRANGO($eqFor$)
3: **for** $i \leftarrow 0, nProc - 1$ **do**
4: $partActual \leftarrow i$
5: $virtualesLocales \leftarrow \emptyset$
6: $(init, end) \leftarrow$ CREARRANGOPARTICION($forInit, forEnd$)
7: $rangoPart \leftarrow (init, end)$
8: $newFor \leftarrow$ CREARFOR($eqFor, rangoPart$)
9: SETPART($newFor, partActual$)
10: **if** $partActual > 0$ **then**
11: PROCESAREQSINIT($newFor$)
12: **end if**
13: $eqs \leftarrow eqs \cup \{newFor\}$
14: **if** $partActual < nProc - 1$ **then**
15: PROCESAREQSEND($newFor$)
16: **end if**
17: **for all** $virt \in virtualesLocales$ **do**
18: SETPART($virt, partActual$)
19: **end for**
20: $totalVirtuales \leftarrow totalVirtuales \cup virtualesLocales$
21: **end for**
22: $eqs \leftarrow eqs \setminus \{eqFor\}$
23: **end procedure**

Aquí, cada ecuación de tipo `for` es subdividida en cuantas partes se desee particionar el modelo ($nProc$). Es decir, es convertida a $nProc$ ecuaciones `for` que contienen las mismas ecuaciones, pero sus rangos son el resultado de subdividir en $nProc$ partes al rango de la ecuación que está siendo procesada. La función CREARRANGOPARTICION se encarga de generar el rango adecuado dependiendo de la partición en la que se esté trabajando, y asegura de que se abarque todo el rango de la ecuación original, que los subrangos generados no se solapen entre sí, y que además quede un valor "libre" para

la evaluación en los extremos de las ecuaciones contenidas dentro de ellas. La función SETPART utilizada indica que, tanto la ecuación *newFor* como los nuevos nodos virtuales creados, deben ser procesados en el procesador actual, definido por *partActual*. Luego, dependiendo del número de partición, se evalúan las ecuaciones contenidas en los extremos de los rangos (que definen los límites de las variables en las particiones) para la posible inserción de nodos virtuales.

Describimos la función CREARRANGOPARTICION:

1: **function** CREARRANGOPARTICION(init, end)
2: $init' \leftarrow init + ((end - init)/nProc) * partActual$
3: **if** $partActual > 0$ **then**
4: $init' \leftarrow init' + 2$
5: **end if**
6: **if** $partActual < nProc - 1$ **then**
7: $end' \leftarrow init - 1 + ((end - init)/nProc) * (partActual + 1)$
8: **else**
9: $end' \leftarrow end$
10: **end if**
11: **return** $init', end'$
12: **end function**

La función PROCESAREQSINIT instancia las ecuaciones contenidas en el `for` en el extremo inferior de la variable dentro de la partición, y es procesada en la función PROCESARECUACION. Análogamente la función PROCESAREQSEND realiza el mismo trabajo pero instanciando las ecuaciones en el extremo superior.

1: **procedure** PROCESAREQSINIT($eqFor$)
2: $(init, end) \leftarrow$ OBTENERRANGO($eqFor$)
3: $init' \leftarrow init - 1$
4: **for all** $eq \in$ OBTENEREQS($eqFor$) **do**
5: $eqInst \leftarrow$ INSTANCIAR($eq, init'$)
6: PROCESARECUACION($eq, eqInst, nSolap$)
7: **end for**
8: **end procedure**

La función INSTANCIAR toma una ecuación e instancia el índice de la misma en un valor dado. Por ejemplo,

INSTANCIAR(`der(x[i])= x[i-1] - x[i+1] - r * x[i]`, 4) =
`der(x[4])= x[4-1] - x[4+1] - r * x[4]`

```
1: procedure procesarEqsEnd(eqFor)
2:     (init, end) ← obtenerRango(eqFor)
3:     end′ ← end + 1
4:     for all eq ∈ obtenerEqs(eqFor) do
5:         eqInst ← instanciar(eq, end′)
6:         procesarEcuacion(eq, eqInst, nSolap)
7:     end for
8: end procedure
```

La función procesarEcuacion analiza cada ecuación instanciada (en las funciones procesarEqsInit y procesarEqsEnd) y evalúa la posibilidad de ingresar nodos virtuales. Esto lo realiza analizando cada ocurrencia de la variable en cuestión en la expresión que la define, y revisando si la expresión recae fuera del rango de la ecuación `for` que la contiene (recordemos que si la ecuación es diferencial, entonces en este punto tiene que definir la derivada de una variable de tipo arreglo). Además, realiza este procedimiento de manera recursiva hasta obtener el nivel de solapamiento deseado.

```
1: procedure procesarEcuacion(eq, eqInst, iSolap)
2:     (expIzqInst, expDerInst) = obtenerPartes(eqInst)
3:     (expIzq, expDer) = obtenerPartes(eq)
4:     var ← obtenerVar(expIzq)
5:     varInst ← obtenerVar(expIzqInst)
6:     if iSolap ≤ 0 ∨ tipoEq(eqInst) ≠ DIFERENCIAL then
7:         finalEq ← eqInst
8:         if esVirtual(varInst) then
9:             finalEq ← agregarTermConv(finalEq, varInst)
10:        end if
11:        eqs ← eqs ∪ {finalEq}
12:        return
13:    end if
14:    nuevosVirtuales ← ∅
15:    for all exp ∈ ocurrenciasVar(expDerInst, var) do
16:        if fueraDeRango(exp) then
17:            if ¬(∃virt ∈ virtualesLocales : expr(virt) = exp) then
18:                nuevoVirt ← crearVirtual(exp)
19:                nuevosVirtuales ← nuevosVirtuales ∪ {nuevoVirt}
20:                expDerInst ← reemplazar(expDerInst, exp, nuevoVirt)
21:            else
22:                virt ← buscarVirt(exp)
23:                expDerInst ← reemplazar(expDerInst, exp, virt)
24:            end if
```

```
25:         end if
26:     end for
27:     virtualesLocales ← virtualesLocales ∪ nuevosVirtuales
28:     for all virt ∈ nuevosVirtuales do
29:         virtEq ← armarEcuacionVirtual(eq, virt)
30:         procesarEcuacion(eq, virtEq, iSolap − 1)
31:     end for
32:     newEqInst ← armarEcuacion(expIzqInst, expDerInst)
33:     if esVirtual(varInst) then
34:         newEqInst ← agregarTermConv(newEqInst, varInst)
35:     end if
36:     eqs ← eqs ∪ {newEqInst}
37: end procedure
```

Algunas aclaraciones:

- obtenerPartes supone que la ecuación es de la forma

  `der(x[i])= EXP` o `x[i] = EXP` (ya que dentro de un `for` son las únicas ecuaciones válidas). Devuelve de forma separada la parte izquierda de la igualdad por un lado, y la parte derecha por el otro.

- obtenerVar obtiene la variable que se aloja en la expresión. Por ejemplo,

  obtenerVar(`der(x[i])`) = `x` obtenerVar(`x[i]`) = `x`

- fueraDeRango evalúa si la variable instanciada recae fuera del rango de la ecuación `for` que la contiene.

- expr toma un nodo virtual y devulve la expresión a la cual representa.

- armarEcuacionVirtual es la encargada de generar la ecuación correspondiente a un nodo virtual dado. Esto se hace instanciando la ecuación "original" en el rango de la variable a la cual reemplaza. En esta función no se agrega el *término de convergencia* –que introduce el parámetro K– a la ecuación, sino que éste es agregado mediante agregarTermConv luego de que la ecuación haya sido procesada –recursivamente– por procesarEcuacion.

3.3. Caso multidimensional

El procedimiento que brindamos anteriormente es para particionar ecuaciones de tipo `for` que definen las ecuaciones diferenciales de variables de tipo arreglo de una dimensión.

Si bien la especificación del lenguaje μ-Modelica no admite arreglos multidimensionales por el momento, se tiene pensado admitirlos en un futuro, por lo que nuestro algoritmo debería estar preparado para el tratamiento de arreglos multidimensionales.

Si se trata con arreglos multidimensionales, nos encontraremos con ecuaciones `for` anidadas que los definen. En nuestro algoritmo, por cuestiones de simplicidad, decidimos procesar las ecuaciones `for` más anidadas. De esta forma, de los `for` anidantes nos quedamos con los rangos que definen, ya que los necesitamos para definir nuevos nodos virtuales que ahora serán arreglos, en algunos casos de más de una dimensión. Más precisamente, si hay `D` ecuaciones `for` anidadas, los nodos virtuales creados tendrán dimensión `D` (si `D` = `0`, serán variables simples en lugar de variables de tipo arreglo).

Para ello realizamos ligeros ajustes en el algoritmo. Primero, introducimos una nueva variable *entornoRangos* que es un arreglo encargado de guardar los rangos de las ecuaciones `for` anidantes de niveles superiores:

$entornoRangos \leftarrow [\,]$

Modificamos la función principal NODOSVIRTUALES, la cual ahora llamará a PROCESARFORREC en lugar de PROCESARFOR:

```
1:  function NODOSVIRTUALES(modelo)
2:      eqsModelo ← OBTENEREQS(modelo)
3:      eqs ← eqsModelo
4:      for all ecuacion ∈ eqsModelo do
5:          PROCESARFORREC(ecuacion)
6:      end for
7:      nuevoModelo ← CREARNUEVOMODELO(modelo, eqs)
8:      AGREGARDECLARACIONES(nuevoModelo, totalVirtuales)
9:      AGREGARASIGNACIONESINICIALES(nuevoModelo, totalVirtuales)
10:     return nuevoModelo
11: end function
```

Y la función PROCESARFORREC se define de la siguiente manera:

```
1:  procedure PROCESARFORREC(ecuacion)
2:      if TIPOEQ(ecuacion) = FOR then
3:          if TIENEFORANIDADOS(ecuacion) then
4:              (init, end) ← OBTENERRANGO(ecuacion)
5:              eqsAnterior ← eqs
6:              entornoRangosAnterior ← entornoRangos
7:              eqs ← OBTENEREQS(ecuacion)
8:              entornoRangos ← entornoRangos ⌢ [(init, end)]
9:              for all eq ∈ OBTENEREQS(ecuacion) do
10:                 PROCESARFORREC(eq)
```

```
11:             end for
12:             SETEARECUACIONES(ecuacion, eqs)
13:             entornoRangos ← entornoRangosAnterior
14:             eqs ← eqsAnterior
15:         else
16:             for all eq ∈ OBTENEREQS(ecuacion) do
17:                 (expIzq, expDer) = OBTENERPARTES(eq)
18:                 var ← OBTENERVAR(expIzq)
19:                 if LARGO(entornoRangos) ≠ DIM(var) − 1 then
20:                     return ;
21:                 end if
22:             end for
23:             PROCESARFOR(ecuacion)
24:         end if
25:     end if
26: end procedure
```

La idea de esta función es que, cuando encontramos una ecuación `for`, la procesamos si ésta no anida a ninguna ecuación `for`, es decir, si las ecuaciones que contiene son sólo algebraicas y diferenciales. En el caso que anide alguna ecuación `for`, guardamos su rango y llamamos recursivamente a PROCESARFORREC sobre las ecuaciones que contiene. De esta manera, esta función se llamará de forma recursiva "bajando" entre las ecuaciones `for` hasta encontrar las que sólo contienen ecuaciones algebraicas y diferenciales. En este punto, se llama a PROCESARFOR sobre estas ecuaciones, para procesarlas como se ha explicado anteriormente. Además, una ecuación `for` sólo será procesada si el último índice de las variables definidas en las subecuaciones coinciden con el índice definido por ella (de lo contrario, no será procesada y quedará igual que en el modelo original, sin modificaciones).

Por otro lado, las funciones SETPART, FUERADERANGO, ARMARECUACIONVIRTUAL, AGREGARDECLARACIONES y AGREGARASIGNACIONESINICIALES fueron cambiadas para considerar ahora las multidimensionalidades de las variables y los nodos virtuales creados, utilizando el arreglo global *entornoRangos*.

Mostramos como ejemplo el siguiente código de un modelo de advección-reacción bi-dimensional:

```
1 model advection_2d
2   parameter Real ax = 1, ay = 1, r = 1000;
3   constant Integer N = 10, M = 10;
4   parameter Real dx = 10/N;
5   parameter Real dy = 10/M;
6   Real u[N,M](each fixed = false);
```

```
7  initial equation
8    for i in 1:5 loop
9      u[1,i] = 1;
10   end for;
11 equation
12   der(u[1,1]) = (-u[1,1]*ax/dx)
13                 + (-u[1,1]*ay/dy)
14                 + r * (u[1,1]^2)
15                     * (1- u[1,1]);
16   for i in 2:N loop
17       der(u[i,1]) = (-u[i,1]*ax/dx)
18                     + (-u[i,1] + u[i-1,1])
19                       * ay/dy
20                     + r * (u[i,1]^2)
21                         * (1 - u[i,1]);
22   end for;
23
24   for i in 2:M loop
25       der(u[1,i]) = (-u[1,i] + u[1,i-1])
26                       * ax/dx
27                     + (-u[1,i] * ay/dy)
28                     + r * (u[1,i]^2)
29                         * (1 - u[1,i]);
30   end for;
31
32   for i in 2:N loop
33     for j in 2:M loop
34         der(u[i,j]) = (-u[i,j] + u[i,j-1])
35                       * ax/dx
36                     + (-u[i,j] + u[i-1,j])
37                       * ay/dy
38                     + r * (u[i,j]^2)
39                         * (1 - u[i,j]);
40     end for;
41   end for;
42 end advection_2d;
```

Código 3.4: Modelo de advección-reacción bi-dimensional, que define una variable de dimensión 2 a través de un doble `for` anidado.

Si queremos particionar al modelo 3.4 en dos partes, el modelo que obtenemos es el siguiente:

```
1 model advection_2d
2   parameter Real r = 1000,ay = 1,ax = 1;
3   constant Integer M = 10,N = 10;
```

```
4    parameter Real dx = 10/N;
5    parameter Real dy = 10/M;
6    Real u[N,M](each fixed = false);
7    parameter Real K_Param = 1;
8    Real node_virt_1;
9    Real node_virt_2[9];
10 initial equation
11   for i in 1:5 loop
12     u[1,i] = 1;
13   end for;
14
15 equation
16   der(u[1,1]) = (-u[1,1] * ax/dx)
17                   + (-u[1,1] * ay/dy)
18                   + r * (u[1,1]^2)
19                       * (1-u[1,1]);
20
21   for i in 2:N loop
22     der(u[i,1]) = (-u[i,1] * ax/dx)
23                     + ((-u[i,1]) + u[i-1,1])
24                       *ay/dy
25                     + r * (u[i,1]^2)
26                         * (1-u[i,1]);
27   end for;
28
29   for i in 2:5 loop
30     der(u[1,i]) = ((-u[1,i]) + u[1,i-1]) * ax/dx
31                     + ((-u[1,i] * ay/dy))
32                     + r * (u[1,i]^2) * (1-u[1,i]);
33   end for;
34
35   der(u[1,6]) = ((-u[1,6]) + u[1,5]) * ax/dx
36                   + ((-u[1,6] * ay/dy))
37                   + r * (u[1,6]^2) * (1-u[1,6]);
38
39   der(node_virt_1) =
40     ((-node_virt_1) + u[1,5]) * ax/dx
41       +((-node_virt_1 * ay/dy))
42       + r * (node_virt_1^2) * (1-node_virt_1)
43       + K_Param * (u[1,6] - node_virt_1);
44
45   der(u[1,7]) = ((-u[1,7]) + node_virt_1) * ax/dx
46                   + ((-u[1,7] * ay/dy))
47                   + r * (u[1,7]^2) * (1-u[1,7]);
48
```

```
  for i in 8:10 loop
    der(u[1,i]) = ((-u[1,i]) + u[1,i-1]) * ax/dx
                    +((-u[1,i] * ay/dy))
                    + r * (u[1,i]^2) * (1-u[1,i]);
  end for;

  for i in 2:N loop
    for j in 2:5 loop
      der(u[i,j]) = ((-u[i,j])+u[i,j-1])
                        * ax/dx
                      + ((-u[i,j]) + u[i-1,j])
                        * ay/dy
                      + r * (u[i,j]^2)
                          * (1-u[i,j]);
    end for;
    der(u[i,6]) = ((-u[i,6]) + u[i,5]) * ax/dx
                    + ((-u[i,6])+u[i-1,6])
                      * ay/dy
                    + r * (u[i,6]^2)
                          * (1-u[i,6]);
    der(node_virt_2[i-1]) =
      ((-node_virt_2[i-1]) + u[i,5]) * ax/dx
        + ((-node_virt_2[i-1]) + u[i-1,6])
          * ay/dy
        + r * (node_virt_2[i-1]^2)
            * (1-node_virt_2[i-1])
        + K_Param * (u[i,6] - node_virt_2[i-1]);
    der(u[i,7]) = ((-u[i,7])
                    + node_virt_2[i-1]) * ax/dx
                    + ((-u[i,7]) + u[i-1,7])
                      * ay/dy
                    + r * (u[i,7]^2) * (1-u[i,7]);
    for j in 8:10 loop
      der(u[i,j]) = ((-u[i,j]) + u[i,j-1])
                        * ax/dx
                      + ((-u[i,j]) + u[i-1,j])
                        * ay/dy
                      + r * (u[i,j]^2) * (1-u[i,j]);
    end for;
  end for;

initial algorithm
  node_virt_1:=u[1,6];

  for i in 2:10 loop
```

```
94      node_virt_2[i-1]:=u[i,6];
95    end for;
96
97  end advection_2d;
```

Código 3.5: Paralelización con *nodos virtuales* aplicado al modelo 3.4 con nivel de solapamiento 1.

3.4. Análisis de Error y Nodos Virtuales

Como ya hemos desarrollado anteriormente, el particionado de un modelo para su ejecución en paralelo implica una sincronización entre procesos para la comunicación de datos necesarios, entre ellos, los valores de ciertas variables computadas localmente en un procesador dado[1]. Esto se debe a que los submodelos obtenidos tienen ciertas dependencias entre ellos para el cálculo de sus variables locales.

Dado que no podemos realizar infinitas sincronizaciones entre procesos, y que además estamos simulando modelos de sistemas continuos en donde el tiempo y las variables avanzan de forma continua, evidentemente hay un problema. Si el cómputo de una variable x_1 en un procesador P_1 utiliza el valor de una variable x_2 que es computada en otro procesador P_2, puede que este sea un valor *desactualizado*. Esto sucede si el valor de x_2, digamos v_2, se recibió en el tiempo de sincronización t_{sync_1}, y antes de la próxima sincronización en t_{sync_2} la variable en cuestión cambió su valor a v_2' en el instante de tiempo t_c, donde $t_{sync_1} < t_c$ y $t_c < t_{sync_2}$. En este punto, cada cómputo de la variable x_1 (es decir, cada cambio en el valor de ésta) en el intervalo de tiempo (t_c, t_{sync_2}) será erróneo ya que realizará sus cálculos utilizando v_2 cuando en realidad debería utilizar v_2'. En el ejemplo gráfico 3.3 de la sección 3.1 vemos este problema.

Por otro lado, como los cómputos de la variable x_1 son erróneos, el cálculo de las variables en procesadores distintos a P_1 que dependan de su valor también lo serán, además de sufrir el error producido por la utilización de valores desactualizados como se mencionó anteriormente.

Esta propagación de errores en algunos casos puede provocar que el modelo en su totalidad se torne inestable, y esto sucede cuando los errores en las variables son demasiado grandes (en este punto llamamos error a la diferencia del valor de una variable cuando se computa en una simulación no paralela (secuencial) y cuando lo hace en una simulación paralela).

Lo que se intenta hacer al utilizar nodos virtuales es amortiguar estos errores. Cuando ingresamos en P_1 un nodo virtual $virt_{x_2}$ que representa a x_2 como se explicó en las secciones anteriores, la finalidad es mantener en este

[1] Llamamos *variable local* a una variable de un submodelo que es computada en un procesador en particular.

procesador el valor de x_2 lo más actualizado posible. La ecuación diferencial de $virt_{x_2}$ será muy similar a la de x_2 junto a un término que forzará su convergencia a la variable representada. De esta manera se intenta realizar los cálculos de x_1 lo más correctamente posible, especialmente en el intervalo de tiempo (t_c, t_{sync_2}), ya que mediante $virt_{x_2}$ podremos actualizar, mediante su ecuación, el valor utilizado para el cómputo de x_1.

Es muy probable que, para realizar el cómputo de la variable $virt_{x_2}$, necesitemos valores de otras variables computadas en un procesador diferente. Estamos en una situación similar a la anterior, aunque aquí esperamos que el error total del modelo sea menor debido a, primero, al cómputo más preciso de x_1 en el intervalo (t_c, t_{sync_2}) gracias a la utilización del nodo virtual $virt_{x_2}$, y segundo, al término en la ecuación diferencial de $virt_{x_2}$ que la obliga a converger a x_2. Si se desea, podemos agregar otro nodo virtual para mitigar el error sobre $virt_{x_2}$ con el propósito de amortiguar más aun el error del modelo en su totalidad. Esto se realiza aumentando el *nivel de solapamiento* como se explicó en la sección 3.1.1, aunque de todas maneras un nivel muy alto implica una gran cantidad de nodos virtuales y de comunicación entre procesos, lo que puede concluir en un modelo demasiado complejo y su simulación podría llevar demasiado tiempo respecto a la simulación del modelo sin nodos virtuales, o más aún, respecto a la simulación del modelo de manera secuencial. Por lo tanto, se busca un nivel de solapamiento de manera que reduzca el error introducido por el paralelismo, pero que no afecte demasiado la performance de la simulación.

En la figura 3.6 podemos ver el resultado de simulación de la variable x_1 si consideramos que es la variable `x[499]` del modelo 3.1. Utilizamos esta variable debido a que es la "última" que es computada en el procesador P_1 (`x[1]`,...,`x[499]` se alojan en el procesador P_1, mientras que `x[500]`,...,`x[1000]` lo hacen en el procesador P_2). Para realizar su cómputo, `x[499]` necesita, además de las variables `x[498]` y `x[499]` —computadas en el mismo procesador— la variable `x[500]` la cual es la "primera" que es computada en el procesador P_2, por lo que veremos instantáneamente el resultado de utilizar un nodo virtual aquí. En la gráfica, comparando con la simulación de forma secuencial, podemos apreciar la estabilidad que agrega la utilización de nodos virtuales con nivel de solapamiento 1 (equivalente al modelo 3.2), respecto a la simulación paralela sin la utilización de ellos.

Además, en la figura 3.7 podemos ver cómo el nodo virtual "sigue" los valores de la variable a la cual representa. En este caso, $virt_{x_2}$ equivale al nodo virtual `virt1` del modelo 3.2, y la variable que representa, x_2, equivale a la variable `x[C]` (`x[500]`) del modelo anteriormente mencionado.

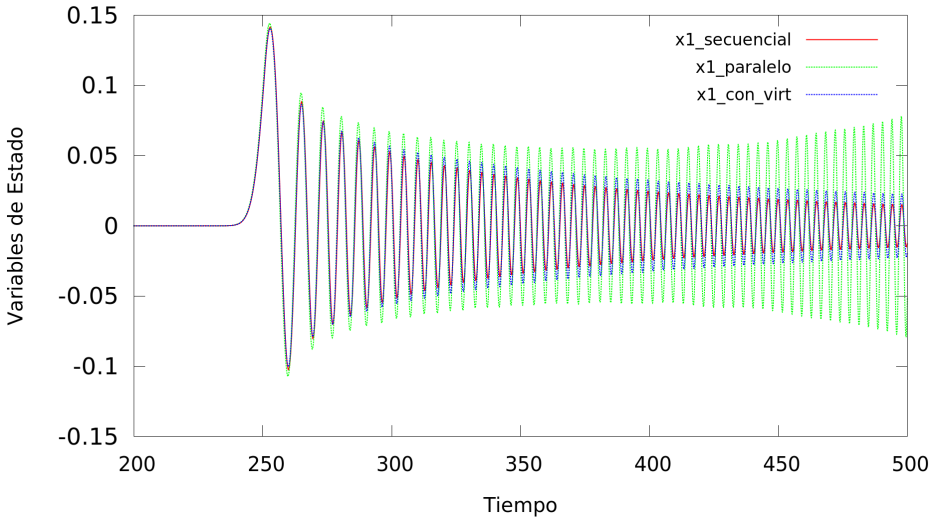

Figura 3.6: Comparación de variable x_1 simulada en secuencial, paralelo y paralelo utilizando *nodos virtuales*.

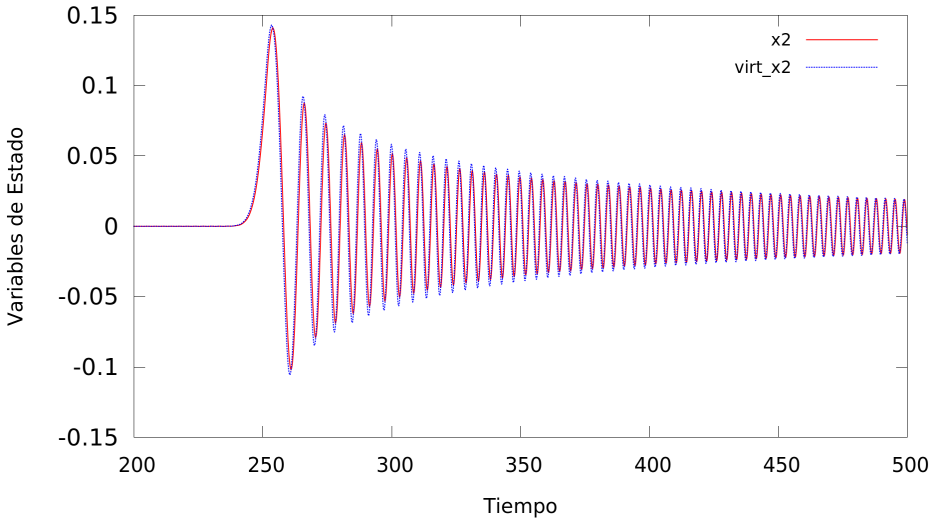

Figura 3.7: Comparación de nodo virtual $virt_{x_2}$ con respecto a la variable x_2 que representa.

Capítulo 4

Implementación

Se realizó una implementación prototipo en C++ de la herramienta que aplica el algoritmo de paralelización con *nodos virtuales* a un modelo de entrada. Fue ideada como un complemento del compilador μ-Modelica, el cual es la parte del simulador autónomo QSS presentado en [8] que se encarga de parsear el archivo **.mo** de entrada para construír, a partir del código del modelo escrito en μ-Modelica, un **AST**[1], para luego formar la matriz de dependencias y generar el código C necesario para simular el modelo. La herramienta es de software libre y está disponible online[2].

4.1. Generación del Modelo

Al igual que el compilador μ-Modelica, la herramienta utiliza el mismo parser para formar el AST. Una vez que lo construye, obtiene cierta información sobre él y lo atraviesa buscando ecuaciones `for`, modificándolas en caso de que cumplan las condiciones mencionadas en las secciones 3.2 y 3.3, aplicando así el algoritmo de particionado con *nodos virtuales*. Una vez que se obtiene el nuevo modelo, este es guardado en un nuevo archivo **.mo**. Por ejemplo, si el archivo de entrada es **modelo.mo**, entonces el nuevo archivo guardado será **modelo_virt.mo**.

Se aplicó el patrón de diseño *visitor* para recorrer el AST obtenido por el parser. Las variables y funciones se implementaron de manera de corresponderse lo mejor posible al pseudoalgoritmo descripto en la sección 3.2. Los conjuntos, por ejemplo, fueron representados por listas y sus correspondientes iteradores.

En los códigos 4.1, 4.2 y 4.3 se pueden ver, respectivamente, las implementaciones de las funciones procesarFor, procesarEcuacion y armarEcuacionVirtual que se describen en el pseudoalgoritmo de la sección 3.2. Podemos ver que allí se utilizan variables como `_entornoRangos`,

[1] Por su sigla: Abstract Syntax Tree (Árbol de Sintaxis Abstracta).

[2] `https://sourceforge.net/projects/micromodelicavirtualnodes/`.

`_virtualesLocales`, `_eqs`, `_partActual`, `_rangoPart`, `_totalVirtuales` y `_nProc` que se corresponden con las variables del pseudoalgoritmo (sin el guión bajo al principio); `_eqs_it` es un iterador utilizado para iterar sobre la lista `_eqs`.

El objeto de tipo `MMO_MicroModelicaNodosVirtuales_` alberga las variables antes mencionadas.

El tipo `MMO_MicroModelicaOcurrences` (utilizado en el código 4.2) representa al fragmento entre las líneas 15–26 de la función PROCESARECUACION del pseudoalgoritmo. Se encarga de encontrar las ocurrencias de la variable *var* en la expresión *expDerInst* y analiza la posibilidad de introducir nuevos nodos virtuales.

```
1  void MMO_MicroModelicaNodosVirtuales_::
2      procesarFor(AST_Equation_For eqFor)
3  {
4    AST_ForIndex forIndex = eqFor->
5                            forIndexList()->
6                            back();
7
8    for (int i = 0; i < _nProc; i++)
9    {
10     _partActual = i;
11     _virtualesLocales = newVirtInfoList();
12
13     AST_Expression_Range nuevoRango =
14         crearRangoParticion(forIndex);
15
16     AST_ForIndex newForIndex =
17         newAST_ForIndex(forIndex->variable(),
18                         nuevoRango);
19
20     _rangoPart = newForIndex;
21
22     AST_ForIndexList newForIndexList =
23         newAST_ForIndexList();
24
25     newForIndexList->push_back(_rangoPart);
26
27     AST_Equation newFor =
28         newAST_Equation_For(newForIndexList,
29                             eqFor->equationList());
30
31     newFor->setLineNum(eqFor->lineNum());
32
```

```
33      setPart(newFor->getAsFor());
34
35      if (_partActual > 0)
36        procesarEqsInit(newFor->getAsFor());
37
38      _eqs->insert(_eqs_it, newFor);
39
40      if (_partActual < _nProc - 1)
41        procesarEqsEnd(newFor->getAsFor());
42
43      agregarParticionesVirts();
44
45      _totalVirtuales->
46          insert(_totalVirtuales->end(),
47                 _virtualesLocales->begin(),
48                 _virtualesLocales->end());
49
50      delete(_virtualesLocales);
51
52      _virtualesLocales = NULL;
53    }
54
55    _processedFors++;
56    _eqs_it = _eqs->erase(_eqs_it);
57    _eqs_it--;
58  }
```

Código 4.1: Implementación de la función PROCESARFOR.

```
1  void MMO_MicroModelicaNodosVirtuales_::
2      procesarEcuacion(AST_Equation eq,
3                       AST_Equation eqInst,
4                       int iSolap)
5  {
6    AST_Expression expDer = eq->
7                            getAsEquality()->
8                            right();
9
10   AST_Expression expIzq = eq->
11                           getAsEquality()->
12                           left();
13
14   AST_Expression expIzqInst = eqInst->
15                               getAsEquality()->
16                               left();
```

```
17
18    AST_Expression expDerInst = eqInst->
19                                getAsEquality()->
20                                right();
21
22    VarName var = obtenerVariable(expIzq);
23    VarName varInst = obtenerVariable(expIzqInst);
24
25    VirtInfo virt = findVirt(varInst);
26
27    if (iSolap <= 0 || !esDiferencial(eqInst))
28    {
29      AST_Equation finalEq = eqInst;
30      if (virt != NULL)
31      {
32          finalEq = agregarTermConv(finalEq, virt);
33      }
34
35      _eqs->insert(_eqs_it, finalEq);
36      return;
37    }
38
39    if (eqInst->equationType() != EQEQUALITY)
40    {
41      Error::getInstance ()->
42          add (eqInst->lineNum (),
43               EM_AST,
44               ER_Error, "Internal error...");
45      return;
46    }
47
48    MMO_MicroModelicaOcurrences varOcurrences =
49      newMMO_MicroModelicaOcurrences(expDer,
50                                     expDerInst,
51                                     var,
52                                     this);
53
54    varOcurrences->execute();
55
56    VirtInfoList nuevosVirtuales =
57      varOcurrences->getNuevosVirtuales();
58
59    _virtualesLocales->
60      insert(_virtualesLocales->end(),
61             nuevosVirtuales->begin(),
```

```
62              nuevosVirtuales->end());
63
64    VirtInfoListIterator it;
65
66    foreach (it, nuevosVirtuales)
67    {
68      VirtInfo virt = current_element(it);
69
70      AST_Equation ecuacionVirt =
71          armarEcuacionVirtual(eq, virt);
72
73      procesarEcuacion(eq, ecuacionVirt, iSolap - 1);
74    }
75
76    expDerInst = varOcurrences->getResult();
77
78    AST_Equation newEqInst =
79      newAST_Equation_Equality(expIzqInst,
80                               expDerInst);
81
82    if (virt != NULL)
83      newEqInst = agregarTermConv(newEqInst, virt);
84
85    _eqs->insert(_eqs_it, newEqInst);
86  }
```

Código 4.2: Implementación de la función PROCESARECUACION.

```
1  AST_Equation MMO_MicroModelicaNodosVirtuales_::
2      armarEcuacionVirtual(AST_Equation eq,
3                           VirtInfo virt)
4  {
5    AST_Expression_ComponentReference reempExp =
6      virt->getReempExp()->getAsComponentReference();
7
8    AST_Expression expDer = eq->
9                            getAsEquality()->
10                           right();
11
12   // Instanciamos en TODOS los indices
13   AST_ExpressionList indices = reempExp->
14                                firstIndex();
15
16   AST_ExpressionListIterator expIt;
17   AST_ForIndexListIterator indIt;
```

```
18
19    int diff = indices->size() - 1
20                - _entornoRangos->size();
21
22    expIt = indices->begin();
23    // Ajustamos indice
24    for (int i = 0; i < diff; i++)
25      expIt++;
26
27    for (indIt = _entornoRangos->begin();
28         expIt != indices->end() &&
29           indIt != _entornoRangos->end();
30         expIt++, indIt++)
31    {
32      AST_Expression indInst = current_element(expIt);
33      AST_ForIndex ind = current_element(indIt);
34
35      AST_Expression indComp =
36        newAST_Expression_ComponentReferenceExp(
37          ind->variable());
38
39      expDer = instanciarExpresion(expDer,
40                                   indComp,
41                                   indInst);
42    }
43
44    // Instanciamos el ultimo indice que esta a parte
45    expDer = instanciarExpresion(
46      expDer,
47      newAST_Expression_ComponentReferenceExp(
48          _rangoPart->variable()),
49      indices->back());
50
51    // Reemplazamos cada ocurrencia de la expresion
52    //   por el nodo que la representa.
53    // Esto se realiza para todos los nodos virtuales
54
55    VirtInfoListIterator viIt;
56    foreach (viIt, _virtualesLocales)
57    {
58      VirtInfo vi = current_element(viIt);
59
60      expDer = instanciarExpresion(expDer,
61                                   vi->getReempExp(),
62                                   vi->getNodeExp());
```

```
63    }
64
65    AST_Expression deriv =
66      newAST_Expression_Derivative(
67        newAST_ExpressionList(virt->getNodeExp()));
68
69    AST_Equation ecuacionNodoVirtual =
70      newAST_Equation_Equality(deriv, expDer);
71
72    return (ecuacionNodoVirtual);
73  }
```

Código 4.3: Implementación de la función armarEcuacionVirtual.

4.2. Generación del Archivo de Partición

Como dijimos anteriormente, existe una extensión de la implementación del simulador autónomo QSS presentado en [8] que soporta simulación en paralelo. Recibe como parámetro el número de procesadores en los que se quiere simular, y ofrece como opciones el particionado automático o manual del modelo. Si se quiere realizar un particionado manual, se debe adjuntar un **archivo de partición** que indica en qué procesador se debe computar cada variable. Para las variables de tipo arreglo, se indica para cada índice del mismo en qué procesador es computado.

Por lo tanto, como queremos tener total control en el particionado del modelo, generaremos un archivo de partición. Siguiendo el ejemplo anterior, si el archivo del modelo es **modelo.mo**, además de generarse un archivo **modelo_virt.mo** se generará un archivo de partición **modelo_virt.part**:

- Si existe un archivo de partición del modelo original, **modelo.part**, se aprovecha esta información para formar el archivo de partición del nuevo modelo. Sin embargo no se utilizará la información sobre las variables de tipo arreglo involucradas en las ecuaciones `for` modificadas por el algoritmo de particionado, ya que éstas deben particionarse exactamente en los mismos rangos que las ecuaciones.

- Si no existe tal archivo de partición, de todas maneras se generará uno para el nuevo modelo indicando con números de particiones aleatorias para las variables en las que no se tiene información sobre la partición en la que se desea computar. De todas maneras, se recomienda acompañar al archivo **.mo** del modelo con su correspondiente archivo de partición **.part**.

4.3. Módulos

Entre otros módulos, los principales que implementan el algoritmo de particionado con *nodos virtuales* se describen a continuación:

- **ast_info.cpp**: Reúne información del modelo de entrada. En este caso, lee el archivo de partición del modelo si es que existe, y otra información, como por ejemplo declaraciones de constantes y cálculo de dimensiones y tamaños de variables.
- **ast_printer.cpp**: Se encarga de imprimir el modelo generado en un nuevo archivo **.mo**.
- **expression.cpp**: Implementación de algunas utilidades sobre expresiones: incluye funciones para la reducción de expresiones y deducción de igualdad entre ellas.
- **ocurrences.cpp**: Este módulo analiza la necesidad de crear nuevos nodos virtuales en una expresión instanciada.
- **part_file.cpp**: Módulo de lectura y escritura del archivo de partición. Utilizado por los módulos **ast_info.cpp** y **virt_nodes.cpp**.
- **replace.cpp**: Módulo utilizado para el reemplazo de expresiones.
- **var_part_table.cpp**: Representación de una tabla de la forma (variable, partición) en donde a cada variable se le corresponde la información sobre las particiones en donde se aloja.
- **virt_generator.cpp**: Generador de nombres para nuevos nodos virtuales. Además, lleva la cuenta de los nuevos nodos virtuales creados.
- **virt_info.cpp**: Guarda la información necesaria de un nodo virtual. Por ejemplo, expresión que reemplaza, índices utilizados, dimensiones y tamaños, etc.
- **virt_nodes.cpp**: Implementación del algoritmo de particionado con *nodos virtuales*.

Capítulo 5

Ejemplos y Comparaciones

Aplicaremos el algoritmo de particionado con *nodos virtuales* en los modelos de tres sistemas diferentes, y analizaremos los resultados obtenidos en comparación al modelo particionado de forma convencional (estándar), que no utiliza nodos virtuales.

Por cada modelo simulado en paralelo, compararemos tiempos totales de simulación y error obtenido en las variables con respecto al modelo simulado de forma secuencial. Para calcular este error, se calculará el *error cuadrático medio*, en función del tiempo, de algunas variables de estado del modelo y se obtendrá el promedio de ellas. Para obtener resultados más precisos, se simulará cada modelo diez veces y se obtendrá el promedio de cada resultado.

El *nivel de solapamiento* elegido ($nSolap$) para todos los casos es 1 debido a que, si bien podemos bajar el error al aumentar el nivel de solapamiento, se penaliza mucho el tiempo de simulación (simula más lento). El valor elegido para el *parámetro de convergencia* (K) es 1 ya que no se observaron diferencias importantes al variar su valor[1].

Además, las simulaciones en paralelo se realizarán variando el parámetro dT, el cual indica cada cuánto tiempo se sincronizarán los procesos para el intercambio de datos: para una simulación dada, valores más pequeños para dT implican más sincronizaciones entre procesos, mayor tiempo de simulación y menor error en el cálculo de variables. De manera inversa, valores más grandes para dT implican menos sincronizaciones entre procesos –lo que permite que estos se ejecuten más asincrónicamente–, por lo que se espera menor tiempo de simulación, pero mayor error en el cálculo de variables.

Todos los resultados presentados aquí son de simulaciones realizadas en una computadora-servidor con un procesador AMD Opteron 6272, el cual posee 64 núcleos de procesamiento. Además, utilizando una computadora

[1] Aunque aquí el parámetro K parece no tener importancia, en otros modelos podría ser diferente. En principio se espera que, a mayor valor en la elección de K, mayor velocidad de convergencia en los nodos virtuales y por ende menor error en los resultados de simulación.

doméstica que posee un procesador AMD FX-4100 de 4 núcleos de procesamiento, se simularon los modelos divididos hasta 4 particiones, observando que los resultados obtenidos se corresponden con los realizados en la computadora-servidor.

5.1. Línea de Transmisión

El sistema de línea de transmisión es el que ya utilizamos en la sección 3.1 para explicar el funcionamiento del algoritmo de particionado de cómputos redundantes con *nodos virtuales*. El modelo que utilizaremos es el descripto por el código 3.1 de dicha sección.

En la figura 5.1 podemos ver el resultado de simular el modelo durante 1000 segundos de forma secuencial, en donde se visualizan los valores de las variables `x[1]`, `x[201]`, `x[401]`, `x[601]` y `x[801]` a medida que evoluciona el tiempo. Estas variables son las que utilizaremos para el cálculo de errores cuadráticos medios. El tiempo de simulación total realizado en forma secuencial fue de 22,5 segundos.

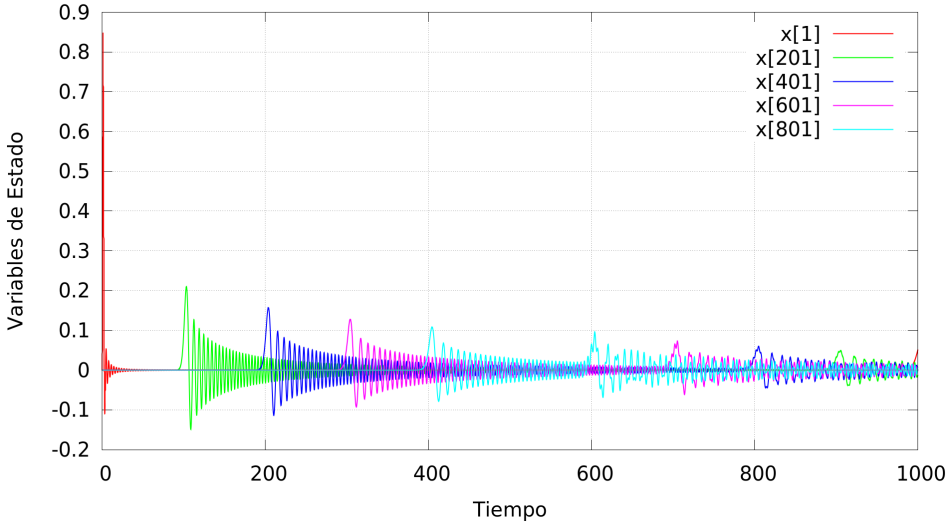

Figura 5.1: Resultado de simulación de 1000 segundos, de forma secuencial, del modelo de línea de transmisión (código 3.1).

En la figura 5.2 podemos ver los gráficos de las simulaciones del modelo particionado en 4 partes, comparando el modelo que no utiliza nodos virtuales, y el modelo que sí lo hace. A medida que dT aumenta hasta 1,3, el modelo sin nodos virtuales obtiene un error de hasta 3,01e−5, con tiempo de simulación de 14,002 segundos, mientras que el modelo que los implementa obtiene un error de hasta 1,96e−5, con un tiempo de simulación de 15,392

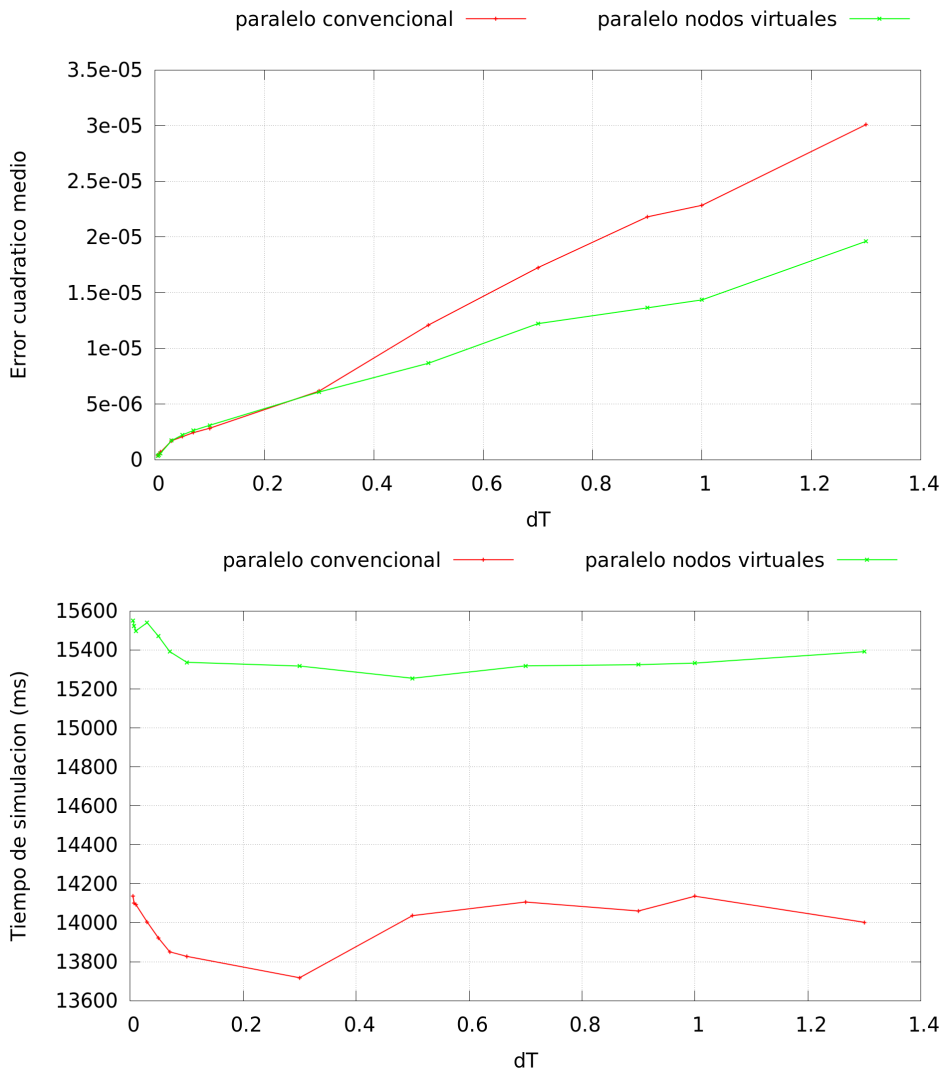

Figura 5.2: Promedio de errores cuadráticos medios (con respecto al modelo simulado en forma secuencial), y tiempos de simulación del modelo de línea de transmisión particionado en 4 núcleos, con y sin utilización de *nodos virtuales*.

segundos. Como podemos ver, en este caso la utilización de nodos virtuales redujo el error hasta un 34,9 %, pero el tiempo de simulación fue 9,9 % más lento.

Si llevamos dT hasta 4,0, podemos apreciar mucho mejor la estabilidad que se introduce al utilizar nodos virtuales. Esto lo podemos ver en la figura 5.3: con $dT = 4{,}0$, el modelo que no utiliza nodos virtuales obtiene un error

de 2,80e−3 con un tiempo de simulación de 14,291 segundos, y el modelo que los utiliza obtiene un error de 4,65e−5 (98,3 % menos) con tiempo de simulación de 15,317 segundos (7,2 % más lento). En la figura 5.4 se puede apreciar cómo la estabilidad introducida por los nodos virtuales ayuda a obtener resultados más precisos, representando mejor a los obtenidos en la simulación realizada de forma secuencial (ver figura 5.1).

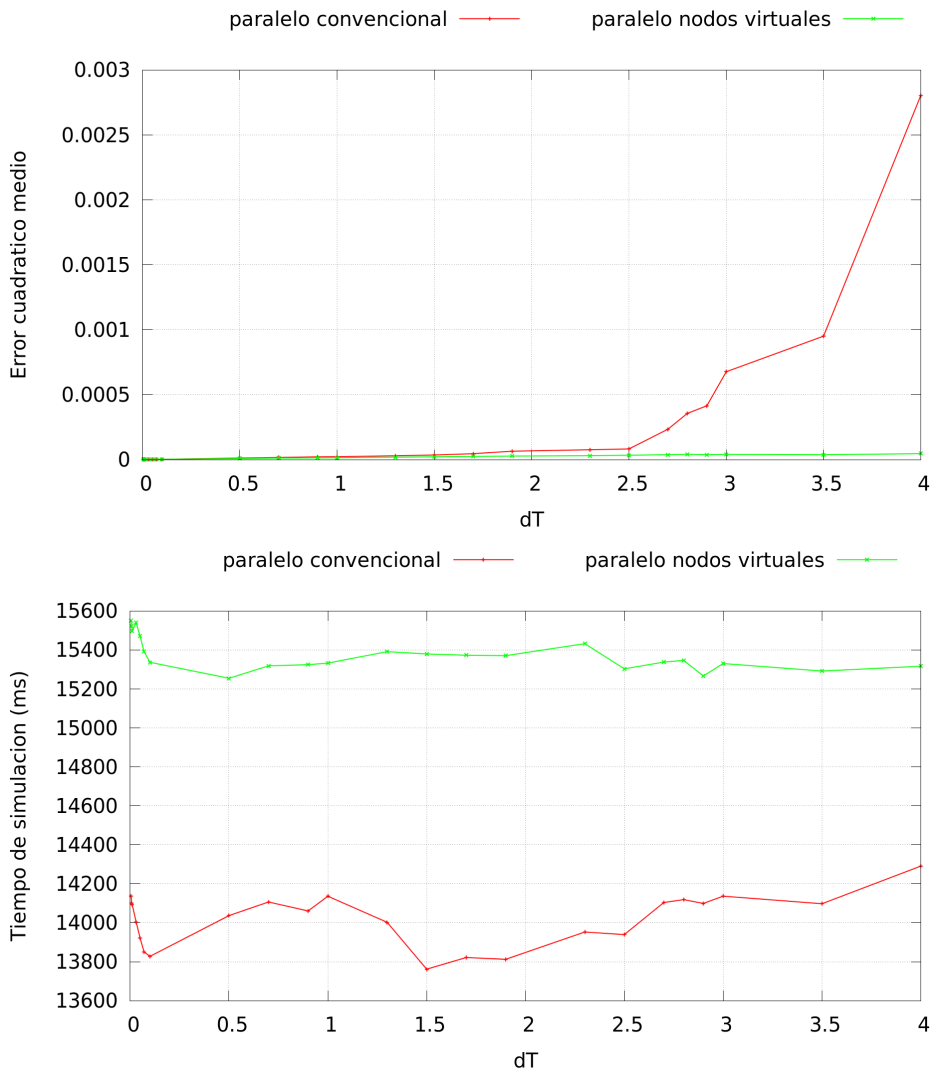

Figura 5.3: Comparación de estabilidad y tiempos de simulación del modelo de línea de transmisión particionado en 4 núcleos, con y sin utilización de *nodos virtuales*.

En la figura 5.5 hacemos el mismo análisis, pero particionando el modelo

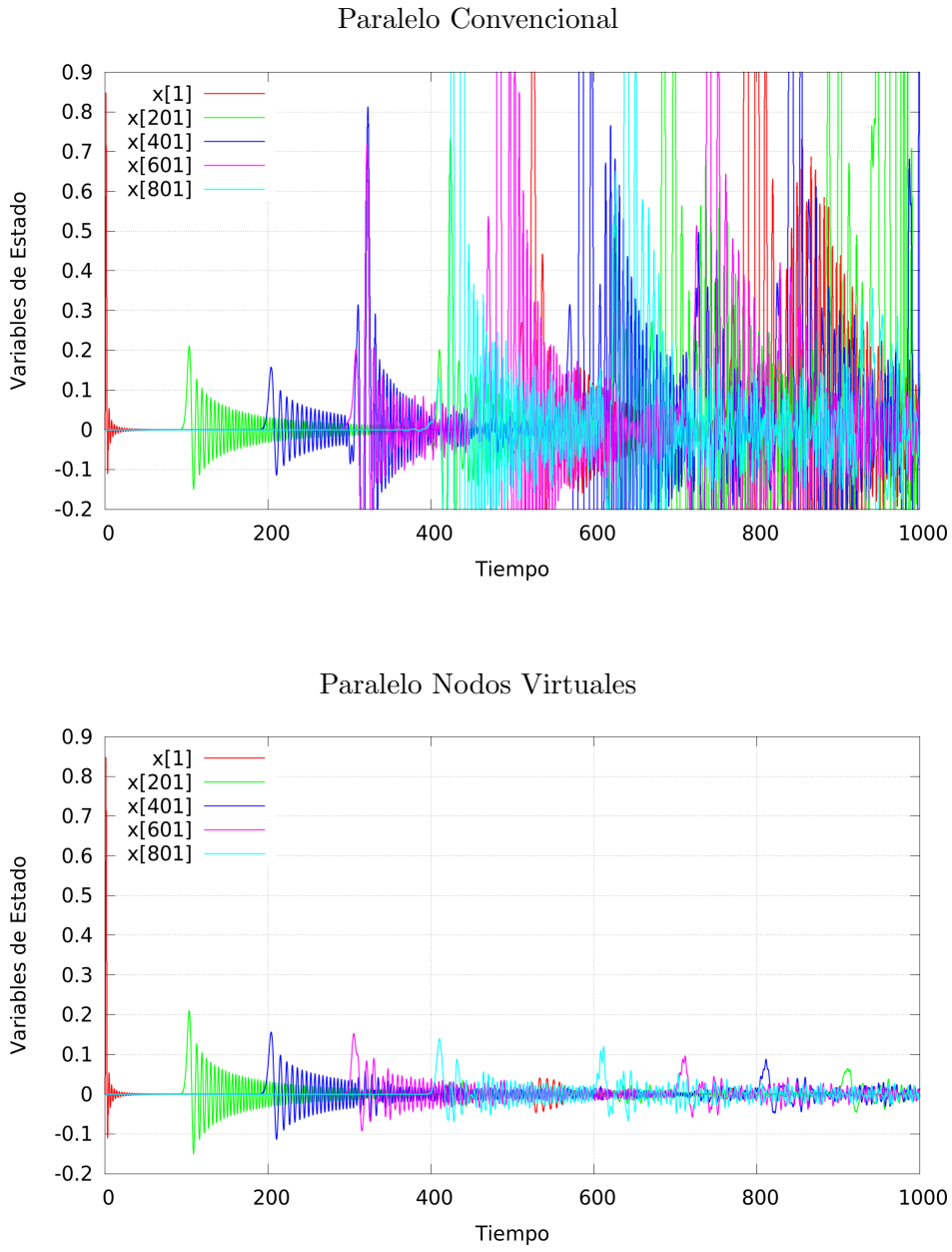

Figura 5.4: Resultados comparativos de simulaciones, utilizando $dT = 4{,}0$, del modelo de línea de transmisión particionado en 4 núcleos, con y sin utilización de *nodos virtuales*.

en 60 partes y utilizando un valor de dT hasta 0,05. Se puede observar que con valores muy pequeños para dT, los tiempos de simulación en ambos casos son incluso mayores que el modelo simulado de manera secuencial. Esto se

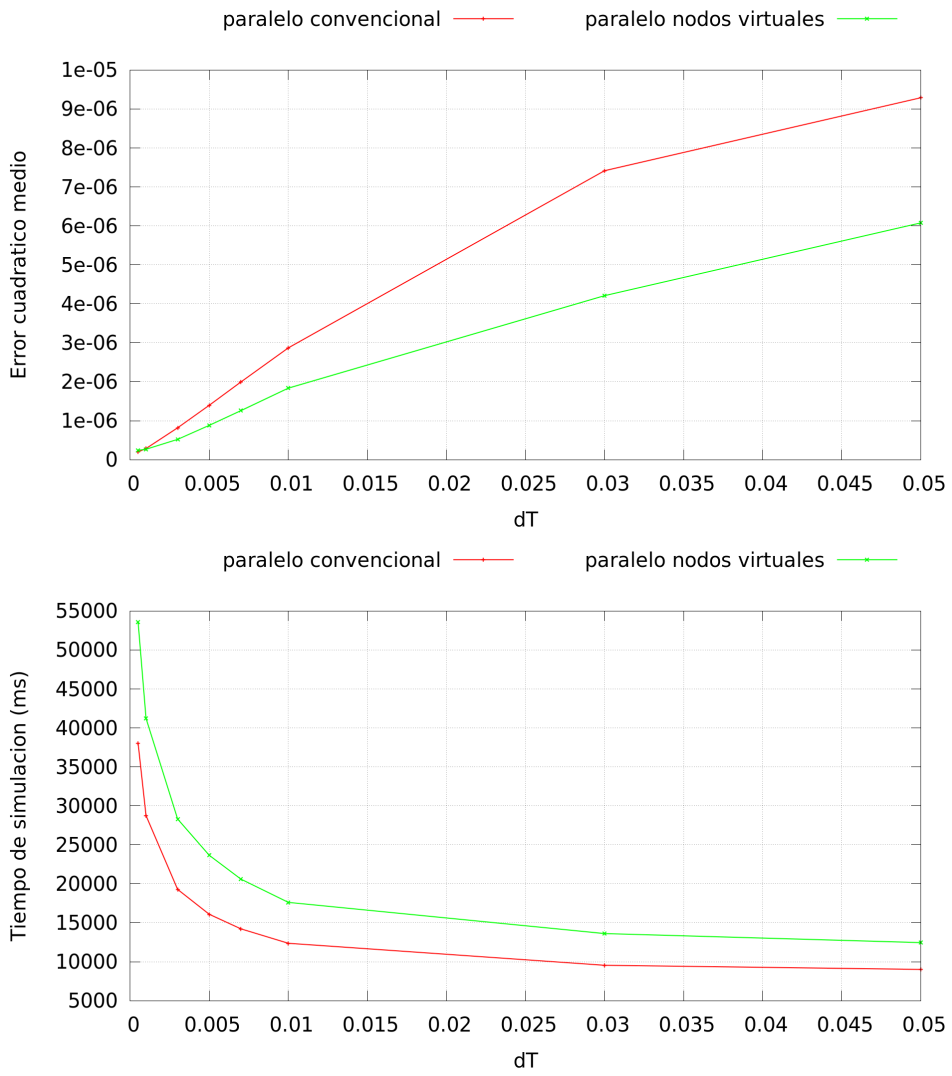

Figura 5.5: Promedio de errores cuadráticos medios (con respecto al modelo simulado en forma secuencial), y tiempos de simulación del modelo de línea de transmisión particionado en 60 núcleos, con y sin utilización de *nodos virtuales*.

debe a que la cantidad de sincronizaciones es demasiada y el costo de simular en paralelo no obtiene mejoras. A medida que aumentan los valores para dT, comenzamos a obtener mejores tiempos de simulación respecto a la realizada secuencialmente.

En el modelo particionado en 60 partes, entonces, el modelo que utiliza nodos virtuales redujo el error hasta un 34,6 % con tiempo de simulación

38,2 % más lento. Obviamente, el porcentaje de mejora en el error puede ser mayor eligiendo valores más grandes para dT.

Podemos ver que, a medida que aumenta la cantidad de particiones del modelo, la tolerancia al error que brinda el modelo que utiliza nodos virtuales crece, pero a su vez los tiempos de simulación empeoran más rápido.

En este problema en particular, se hicieron varias pruebas variando la cantidad de particiones. En todos los casos, el modelo que utiliza nodos virtuales siempre obtuvo mejor error que el modelo que no los tiene.

5.2. Advección-Reacción

Un sistema de advección-reacción describe el fenómeno físico donde las partículas o la energía son transferidas dentro de un sistema físico. Por ejemplo, se puede modelar un sistema compuesto por un tubo de agua, y se puede observar el avance del líquido a traves de éste, en función del tiempo. También, se puede modelar un sistema compuesto por una barra de metal (por ejemplo, acero), y se puede observar la variación de la temperatura en cada sección de la barra ante la presencia de una fuente de calor, por ejemplo, en uno de los extremos de la misma.

El siguiente modelo representa un sistema de advección-reacción:

```
1  model advection_reaction
2    parameter Real alpha = 0.2;
3    parameter Real mu = 10;
4    constant Integer N = 10000;
5    Real u[N];
6  initial algorithm
7    for i in 1:N/3 loop
8    u[i] := 1;
9    end for;
10 equation
11   der(u[1]) = (-u[1]+1)*N
12                 - mu * u[1]
13                     * (u[1]-alpha) * (u[1] - 1);
14   for j in 2:N loop
15     der(u[j]) = (-u[j]+u[j-1])*N
16                   -mu * u[j]
17                       * (u[j]-alpha) * (u[j]-1);
18   end for;
19 end advection_reaction;
```

Código 5.1: Modelo de advección-reacción.

Este modelo será simulado por 1,5 segundos y las variables elegidas para el cálculo de errores cuadráticos medios son `x[2000]`, `x[3000]`, `x[4000]`,

`x[5000]`, `x[6000]`, `x[7000]`, `x[8000]`, `x[9000]` y `x[10000]`. Los resultados de las mismas al simular el modelo de forma secuencial se pueden visualizar en la figura 5.6. El tiempo de simulación total realizado en forma secuencial fue de 3,6 segundos.

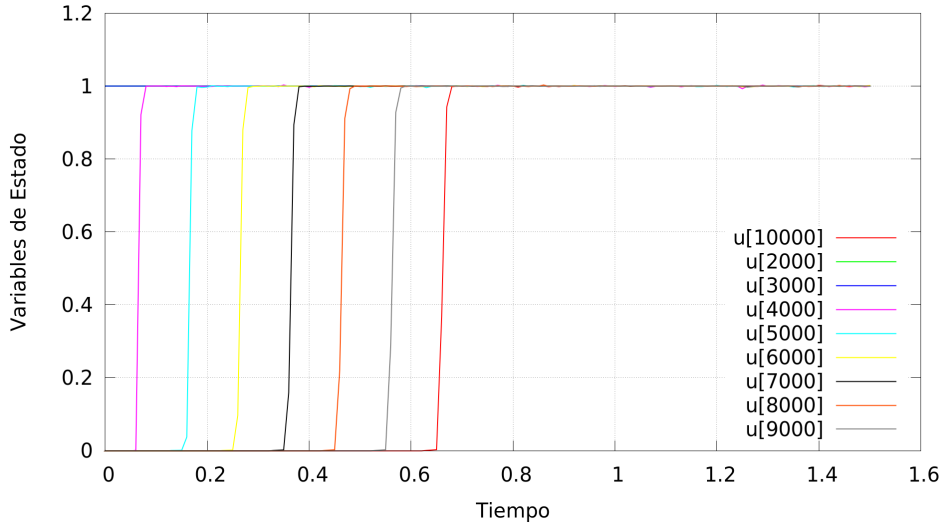

Figura 5.6: Resultado de simulación de 1,5 segundos, de forma secuencial, del modelo de advección-reacción.

En este problema en particular, la aplicación de nodos virtuales parece no tener una mejora total. Variando la cantidad de particiones, el modelo que los utiliza obtuvo mejores errores en algunos casos, pero en otros, peores. Esto se puede ver en la figura 5.7, donde las simulaciones se realizaron en el modelo particionado en 5 partes, y el modelo con nodos virtuales mejoró el error hasta un 27,4 % y simuló hasta 17,1 % más lento. Pero en el modelo particionado en 60 núcleos (ver gráficos de la figura 5.8), el modelo que utiliza nodos virtuales *empeoró* el error, en promedio, en 1,4 % (aunque los tiempos de simulación mejoran un 4,8 %). Esto se debe a que, en este modelo, $\forall i \in 2 \dots N$ la ecuación `u[i]` sólo depende de ella misma y de `u[i-1]`, pero no de `u[i+1]`. Es decir, lo que ocurre en `u[i]` no afecta a `u[1]`,...,`u[i-1]`. Como consecuencia, el aumento de dT sólo provoca una especie de retraso en la señal que avanza a través de las variables, lo que es inevitable incluso con la inserción de nodos virtuales. Al parecer, la utilización de éstos tiene sentido en sistemas como el de línea de transmisión (visto anteriormente) y el de advección-difusión-reacción (que veremos a continuación) en donde se presenta retroalimentación entre las variables, generada por dependencias mutuas entre ellas (como se puede apreciar en la figura 3.2), que introducen inestabilidad en el sistema cuando dT aumenta.

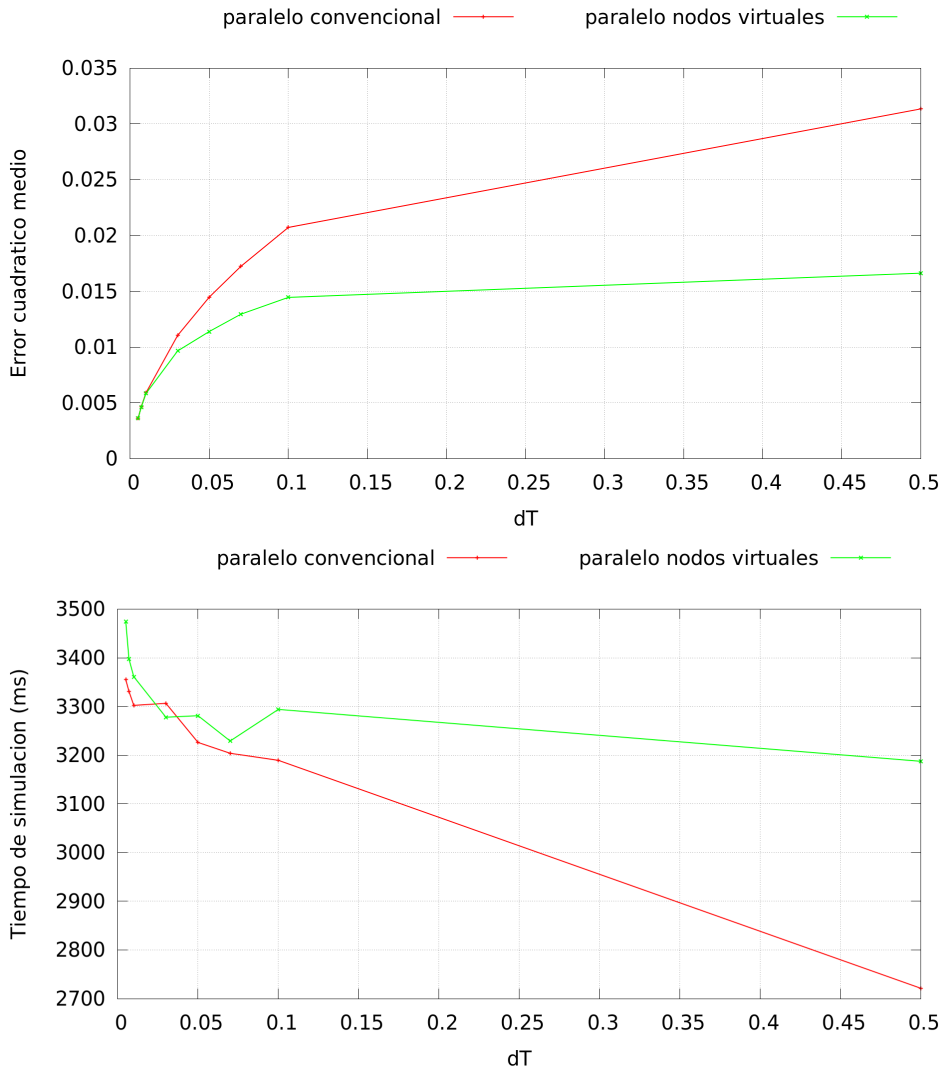

Figura 5.7: Promedio de errores cuadráticos medios (con respecto al modelo simulado en forma secuencial), y tiempos de simulación del modelo de advección-reacción particionado en 5 núcleos, con y sin utilización de *nodos virtuales*.

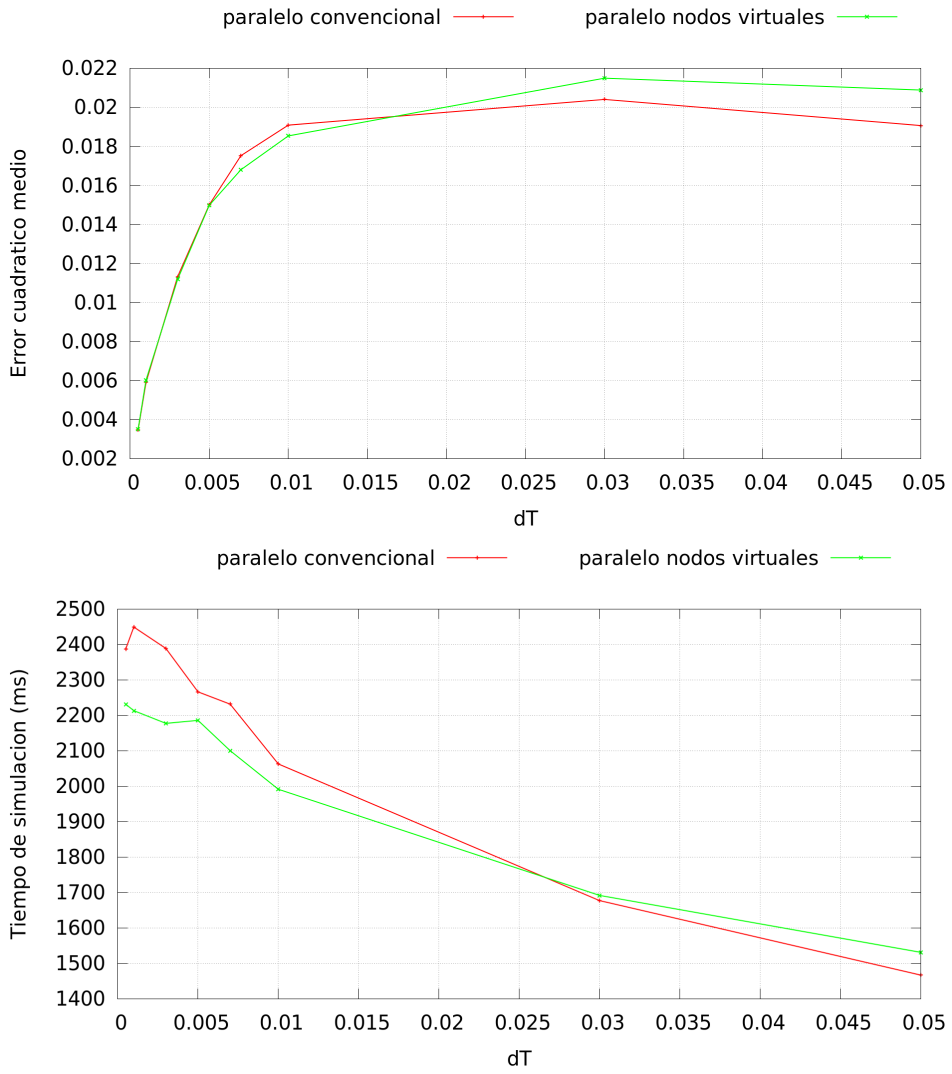

Figura 5.8: Promedio de errores cuadráticos medios (con respecto al modelo simulado en forma secuencial), y tiempos de simulación del modelo de advección-reacción particionado en 60 núcleos, con y sin utilización de *nodos virtuales*.

5.3. Advección-Difusión-Reacción

Un sistema de advección-difusión-reacción es similar a un sistema de advección-reacción pero agrega el término de *difusión* en las ecuaciones derivadas.

```
model advection_difussion_reaction
  constant Integer N = 1000;
  parameter Real a = 1;
  parameter Real d = 0.1;
  parameter Real r = 10;
  parameter Real L = 10;
  parameter Real dx = L/N;
  parameter Real dx2 = dx^2;
  Real u[N];
initial algorithm
  for i in 1:N/5 loop
    u[i] := 1;
  end for;
equation
  der(u[1]) = -a*(u[1]-1)/dx
                + d*(u[2]-2*u[1]+1)/dx2
                + r*(u[1]^2)*(1-u[1]);
  der(u[N])= -a*(u[N]-u[N-1])/dx+d
                *(u[N]-2*u[N]+u[N-1])/dx2
               +r*(u[N]^2)*(1-u[N]);

  for i in 2:N-1 loop
    der(u[i]) = -a*(u[i]-u[i-1])/dx
                  + d*(u[i+1]-2*u[i]+u[i-1])/dx2
                  + r*(u[i]^2)*(1-u[i]);
  end for;
end advection_difussion_reaction;
```

Código 5.2: Modelo de advección-difusión-reacción.

Simularemos este modelo por 5 segundos. De forma similar al modelo de advección-reacción, elegimos a las variables `x[100]`, `x[200]`, `x[300]`, `x[400]`, `x[500]`, `x[600]`, `x[700]`, `x[800]`, `x[900]` y `x[1000]` para el cálculo de errores cuadráticos medios. El resultado de la simulación se aprecia en la figura 5.9. El tiempo de simulación total realizado en forma secuencial fue de 3,2 segundos.

La aplicación de nodos virtuales en este tipo de problema parece funcionar muy bien. En todas las pruebas realizadas (variando la cantidad de particiones), el modelo con nodos virtuales mejoró notablemente el error de

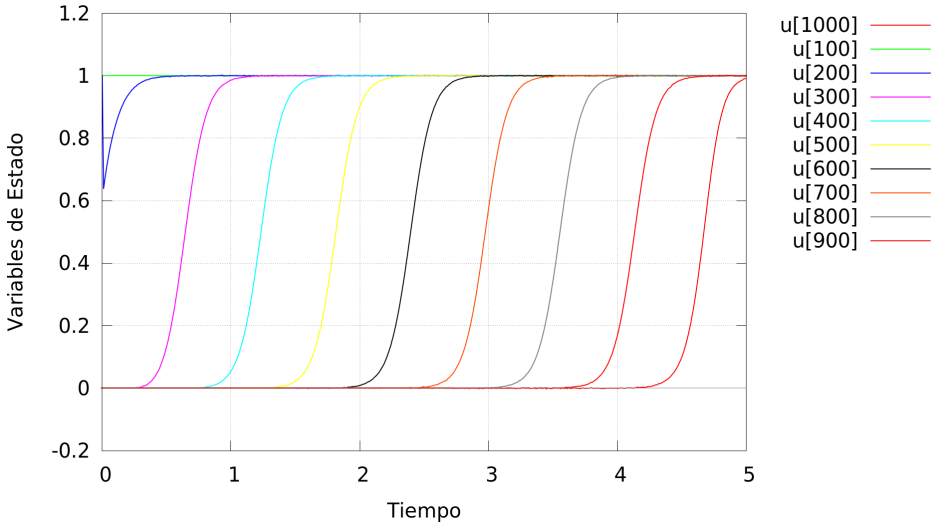

Figura 5.9: Resultado de simulación de 5 segundos, de forma secuencial, del modelo de advección-difusión-reacción.

las variables. En la figura 5.10 las simulaciones se realizaron en el modelo dividido en 4 partes. El modelo con nodos virtuales mejoró el error hasta 55,5 % y el tiempo de simulación fue 12,1 % más lento. En el modelo dividido en 24 partes[2] (ver gráficos de la figura 5.11), el modelo con nodos virtuales redujo el error en un 60,8 % aunque el tiempo de simulación fue 61,7 % más lento. Por ende, podemos ver que en este problema, la introducción de nodos virtuales en el particionado del modelo mejora considerablemente el error aunque la penalización en el tiempo de simulación aumenta rápidamente a medida que se incrementa la cantidad de particiones. En todos los casos de prueba, utilizar valores más grandes para dT no cambió los resultados.

[2]En este problema, no pudimos realizar las simulaciones de modelos particionados en más de 24 partes, debido a que el modelo que no utiliza nodos virtuales no finalizaba las simulaciones correctamente.

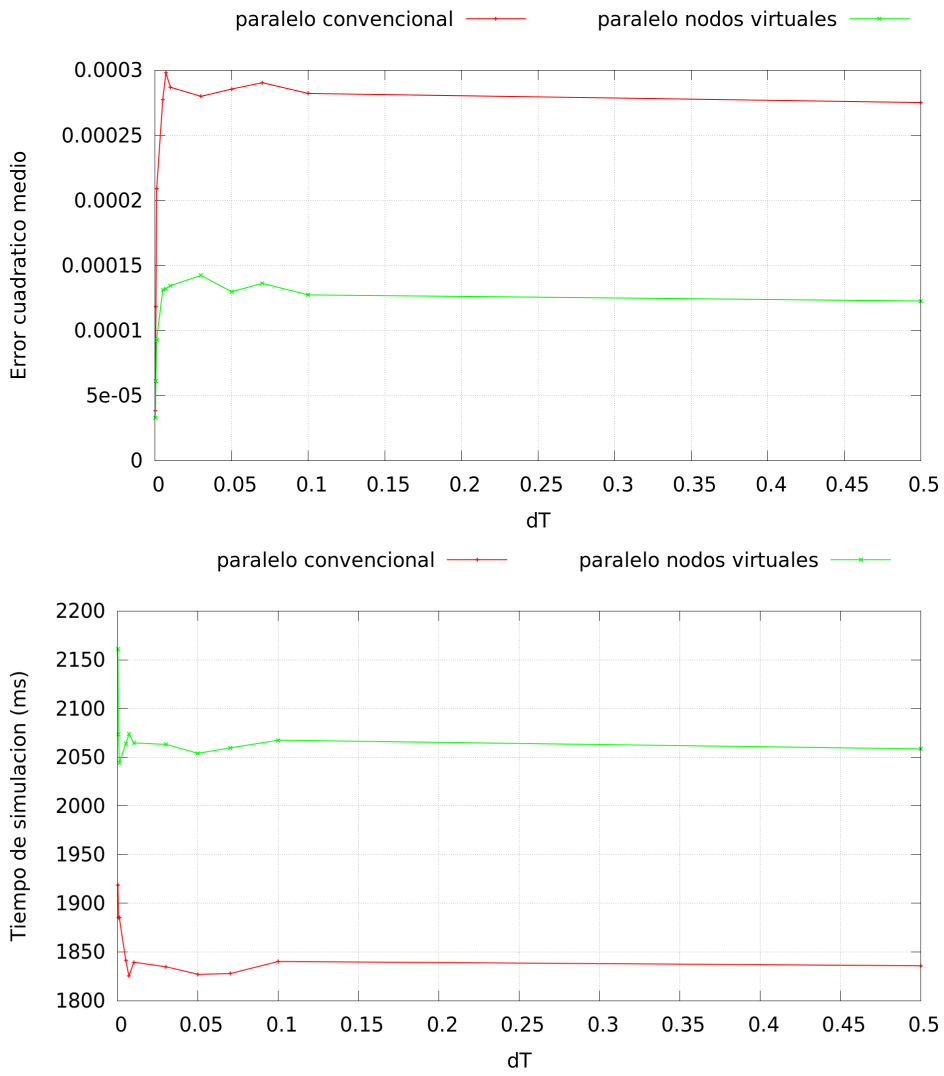

Figura 5.10: Promedio de errores cuadráticos medios (con respecto al modelo simulado en forma secuencial), y tiempos de simulación del modelo de advección-difusión-reacción particionado en 4 núcleos, con y sin utilización de *nodos virtuales*.

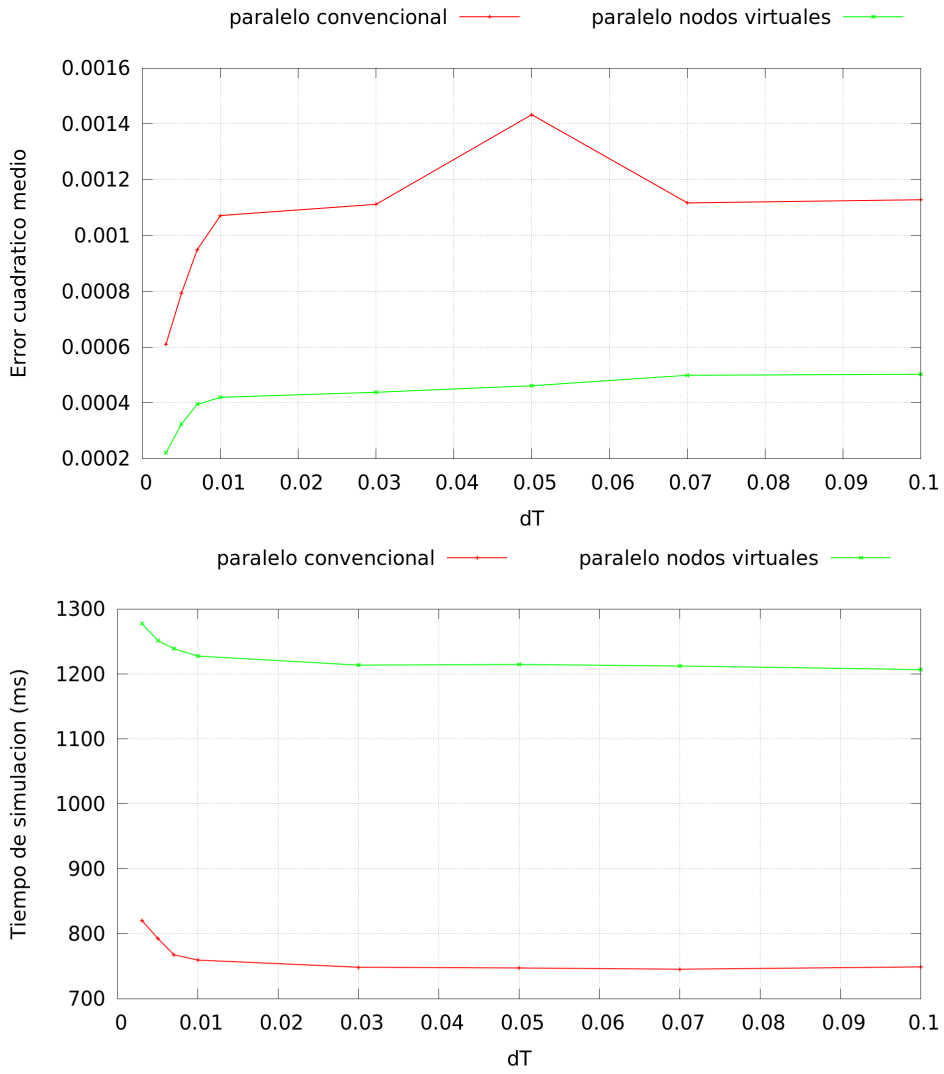

Figura 5.11: Promedio de errores cuadráticos medios (con respecto al modelo simulado en forma secuencial), y tiempos de simulación del modelo de advección-difusión-reacción particionado en 24 núcleos, con y sin utilización de *nodos virtuales*.

Capítulo 6

Conclusiones y Trabajo a Futuro

Se ha realizado un estudio del impacto de la utilización de cómputos redundantes o nodos virtuales en la simulación paralela bajo métodos QSS.

Para ello, se ha presentado un algoritmo que introduce nodos virtuales y particiona el modelo, ubicándolos en las conexiones entre los submodelos.

Su aplicación puede tener buenos resultados en algunos sistemas, pero puede puede tener pocas mejoras en otros. Esto depende del sistema que se esté modelando. El modelo obtenido se vuelve un poco más complejo que el original debido a la inserción de nuevas variables que implican más cómputo y más transmisión de datos entre procesadores, lo que puede ocasionar simulaciones con tiempos de ejecución un poco mayores.

Se implementó el algoritmo de particionado con *nodos virtuales* en una herramienta de software libre, desarrollada en C++. Ésta está disponible online y se puede encontrar en `https://sourceforge.net/projects/micromodelicavirtualnodes/`.

Se analizaron tres casos de modelos a gran escala donde se ha estudiado la aplicación del método de particionado, utilizando el simulador autónomo QSS. En cada uno de ellos se obtuvieron los siguientes resultados:

- **Línea de Transmisión:** Mejora del error en un 34,9 % con un costo de performance de 9,9 % (con valores de dT mayores, el error se reduce hasta 98,3 % con costo de performance de 7,2 %). Al aumentar el número de particiones, la mejora es de 34,6 %, pero el costo de performance es de 38,2 % (la reducción del error puede ser mayor al aumentar el valor de dT).
- **Advección-Reacción:** En algunos casos se obtienen mejoras (27,4 % menos de error con un costo de performance de 17,1 %), y en otros no (aumento del error en 1,4 %, con tiempos de simulación 4,8 % menores, en promedio).

- **Advección-Difusión-Reacción:** Mejora del error en 55,5 % con un costo de performance de 12,1 %. Cuando el número de particiones es mayor, la mejora del error alcanza hasta un 60,8 % pero el costo de performance se eleva hasta 61,7 %.

Por lo tanto, la utilización de este método de particionado se ve más justificada cuando se desea minimizar el error de las variables a costa de un tiempo de simulación mayor. Los modelos simulados en paralelo bajo esta técnica son más robustos con respecto al parámetro dT, ya que su estabilidad se ve afectada en menor proporción cuando se realiza una mala elección de éste.

Por otro lado, el estudio realizado indica que la utilización de esta técnica resulta más beneficiosa en casos donde se presenta retroalimentación entre los submodelos, como el sistema de línea de transmisión y el de advección-difusión-reacción. En caso de que ésta no se presente (como en el sistema de advección-reacción), no hemos llegado a resultados concluyentes.

Algunas cuestiones para considerar en trabajo a futuro:

- Mejorar el tratamiento de las ecuaciones de tipo `for`: En estas ecuaciones, nuestro algoritmo sólo revisa las subecuaciones en los extremos del rango definido por la ecuación `for`, para evaluar la posibilidad de introducir nuevos nodos virtuales. Pueden surgir dependencias a variables computadas en otro procesador en valores más internos del rango, que necesitarían nodos virtuales que las representen.

- Mejorar el tratamiento de ecuaciones `for` anidadas: Cuando este tipo de ecuaciones se presenta (definiendo variables multidimensionales), decidimos por simplicidad particionar las ecuaciones `for` más anidadas (nivel máximo de anidamiento). Se podría brindar como opción particionar las ecuaciones `for` más externas (de mínimo nivel de anidamiento), o poder elegir múltiples niveles de anidamiento sobre los que se desea aplicar el algoritmo.

- Particionado de una ecuación `for` en rangos con longitudes diferentes: Suponiendo que $nProc$ es el número de particiones elegido, nuestra implementación del algoritmo particiona una ecuación `for` en $nProc$ partes, con rangos de longitudes lo más iguales posible. El usuario podría elegir particionar el modelo en rangos de longitudes diferentes para modificar la forma en la que el modelo es particionado, y poder analizar cómo esto afecta a los errores de las variables y a los tiempos de simulación.

Bibliografía

[1] T.G. Kim B. Zeigler and H. Praehofer. *Theory of Modeling and Simulation. Second Edition.* Academic Press, New York, 2000.

[2] 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.

[3] François Cellier and Ernesto Kofman. *Continuous System Simulation.* Springer, New York, 2006.

[4] D. Brück, H. Elmqvist, S.E. Mattsson and H. Olsson. Dymola for multi-engineering modeling and simulation. In *Proceeding of the Second International Modelica Conference*, pages 55.1–55.8. 2002.

[5] J.R. Dormand and P.J. Prince. A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19 – 26, 1980.

[6] Ronald P Fedkiw, Tariq Aslam, Barry Merriman, and Stanley Osher. A non-oscillatory Eulerian approach to interfaces in multimaterial flows (the ghost fluid method). *Journal of computational physics*, 152(2):457–492, 1999.

[7] Joaquín Fernández and Ernesto Kofman. μ-Modelica Language Specification. URL: `http://www.fceia.unr.edu.ar/control/modelica/micromodelicaspec.pdf`. CIFASIS-CONICET, Rosario, Argentina.

[8] Joaquín Fernández and Ernesto Kofman. A stand-alone quantized state system solver for continuous system simulation. URL: `http://sim.sagepub.com/content/90/7/782`. CIFASIS–CONICET, FCEIA, UNR, Rosario, Argentina.

[9] Joaquín Fernández, Ernesto Kofman, and Federico. Bergero. A Parallel Quantized State System Solver for ODEs. *PARALLEL AND DISTRIBUTED COMPUTING*, 2016. Enviado.

[10] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* New York, 2004.

[11] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[12] Ernesto Kofman, Gustavo Migoni, Mario Bortolotto, and François E. Cellier. Quantized State System Simulation. page 6.

[13] Martina Maggio, Kristian Stavåker, Filippo Donida, Francesco Casella, and Peter Fritzson. Parallel simulation of equation-based object-oriented models with quantized state systems on a GPU. In *Proceedings of the 7th International Modelica Conference*, 2009.

[14] Jayadev Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.

[15] James Joseph Nutaro. *Parallel discrete event simulation with application to continuous systems*. PhD thesis, UNIVERSITY OF ARIZONA, 2003.

[16] Kaj Nyström and Peter Fritzson. Parallel simulation with transmission lines in Modelica. In *Proceedings of the 5th International Modelica Conference (Modelica'2006)*, pages 4–5, 2006.

[17] Peter Fritzson, Peter Aronsson, Hakan Lundvall, Kaj Nystrom, Adrian Pop, Levon Saldamli and David Broman. The OpenModelica Modeling, Simulation, and Development Environment. In *Proceedings of the 46th Conference on Simulation and Modeling (SIMS'05)*, pages 83–90. 2005.

[18] L. R. Petzold. A description of DASSL: a differential/algebraic system solver. In *Scientific computing (Montreal, Quebec, 1982)*, pages 65–68. IMACS, New Brunswick, NJ, 1983.

[19] D.M. Rao, N.V. Thondugulam, R. Radhakrishnan, and P.A. Wilsey. Unsynchronized parallel discrete event simulation. In *Simulation Conference Proceedings, 1998. Winter*, volume 2, pages 1563–1570, 1998.