Universidad de Sevilla
Dpto. de Ciencias de la
Computación
e Inteligencia Artificial

Aceleración de Simuladores de Sistemas de Membranas Mediante Computación de Altas Prestaciones con GPU

Resumen extendido presentado por
Miguel Ángel Martínez del Amor
para optar al grado de Doctor
por la Universidad de Sevilla

Miguel Ángel Martínez del Amor

V.º B.º Los Directores de la Tesis

Dr. D. Mario de J. Pérez Jiménez

Dr. D. Ignacio Pérez Hurtado de Mendoza

A Chary,
por ser el mejor regalo de mi vida.

A mis padres,
por darme todo lo que tengo y soy.

Y a mis hermanos y amigos,
siempre por su apoyo incondicional.

Agradecimientos

Esta tesis representa la capacidad de trabajo y sacrificio que he ido adquiriendo desde donde alcanzan mis recuerdos. Nada de esto habría sido posible sin la calidad humana de los que me rodean, que siempre me han alentado a continuar, han conseguido moldear mi personalidad para ser tal y como soy, y han conseguido que me encuentre ahora mismo escribiendo estas palabras: la mejor oportunidad para manifestar mi gratitud.

Me gustaría comenzar por dar las gracias a Chary, el regalo de mi vida, que me ha enseñado a vivir y a amar, y que sin sus ánimos, consejos y apoyo, no habría llegado tan lejos. A mis padres, Lorenzo y Loli, por ofrecerme toda una vida, por su educación y por transmitirme su capacidad de sacrificio, trabajo, y superación. A mis hermanos, José Luis (junto a María y el pequeño Luis) y Mariloli, por siempre estar ahí y por enseñarme a luchar por lo que uno quiere. También a éste primero, mi hermano mayor, por haber sido siempre mi guía y un ejemplo a seguir. A mi abuela, por su cariño, transmitirme toda su experiencia y sabiduría, y ser una segunda madre para mí. Al resto de mi familia, tíos y primos, y a todos mis amigos de infancia y de la peña, por llenar mis recuerdos con muy buenos momentos, de los mejores de mi vida, y que hacen que tenga unas fuertes raíces en mi tierra (Calasparra). Por supuesto, agradecer a todos aquellos que han conseguido hacer de Sevilla un nuevo hogar para mí. Especialmente a mi segunda familia: Carmen y Paco, por abrirme las puertas de su casa y cuidarme como a un hijo más; a Cisco y Virginia, por aceptarme como a un hermano; a Chico y Taor por su fiel amistad; y al resto de familia política que me ha apoyado.

Quiero expresar mi infinita gratitud a todos mis compañeros durante mi etapa de investigador primerizo en el Grupo de Computación Natural de la Universidad de Sevilla. En especial, a mi director Mario de Jesús Pérez Jiménez, por acogerme en su grupo, y por brindarme esta oportunidad única de iniciarme en hacer ciencia bajo su guía. Su capacidad de trabajo y ayuda siempre desinteresada hacia los demás me ha causado un gran efecto positivo. También a mi co-director, Ignacio Pérez Hurtado, por su gran compañerismo, y ser siempre esa mano amiga. Gracias a él mi trayecto en el grupo ha sido mucho más sencillo. A Francisco José Romero, por su gran apoyo e ingenio (agradecimientos especiales por ser el que dio la idea que culminó en el trabajo presentado en mi tesis); a Agustín Riscos, por tener siempre una lección diaria para mí; a Miguel Ángel Gutiérrez por su capacidad de apoyo y de humor; a

Luis, Manu y Luis Felipe por ofrecerme siempre su ayuda, apoyo, y fomentar la cooperación y buen ambiente en el grupo; a Ana, Carmen, Álvaro, Fernando y Andrés también por su ayuda incondicional. Asimismo, deseo hacer extensivo este sentimiento de gratitud al resto de los compañeros incluido el personal de administración y servicios, del Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla. *I would like also to thank all the collaborators (now friends) that passed by our group: Niall, Enrico, Henry, Xiangxiang, and Daniel; because they do not only infected me their passion for science, but also their ability to learn and be open mind with other cultures. Special thanks to Henry Adorna, who started a really interesting and active collaboration with us together with Francis and the Membrane Computing Research Group. Thanks for give us this opportunity to work together, and be also part of your group. Furthermore, I really acknowledge the support of Anne C. Elster during my short stay at Trondheim (Norway), what gave to me the best experience of my life. I would like to extend this to the rest of the HPC-LAB at NTNU, specially for Ian Karlin and Rune E. Jensen, who helped me a lot developing part of my thesis and taught me really interesting and useful things.* Con respecto a la estancia en Trondheim, he de dar las gracias a Alfredo Pérez, quien me cuidó como a un hermano. También debo agradecer la colaboración inicial con Chema y Ginés de la Universidad de Murcia, con quienes comenzamos a desarrollar el trabajo de manera conjunta. Gracias a ellos todo evolucionó rápidamente.

Deseo continuar expresando mis agradecimientos a todos mis formadores y compañeros de formación. A todos los profesores que tuve, desde primaria (EP Nª Sª de la Esperanza) a la universidad (Universidad de Murcia), pasando por secundaria y bachiller (IES Emilio Pérez Piñero), que me motivaron, y sembraron la semilla de la pasión por la ciencia. A todos mis compañeros de fatiga durante aquellos años, en especial en los últimos a José Luis, por su gran apoyo y compañerismo que acabó de forma natural en una gran amistad, al igual que con Fernando, Ana, Alfredo, Juan, Andrés y Fran. *To all the (Portuguese, Brazilian, Italian, German, Belgium and French) people that supported me at my Erasmus experience in Trondheim. Thanks to them all, it was the most personally enriching experience in my life.*

Finalmente, agradezco profundamente todo el apoyo económico ofrecido por la beca de personal investigador en formación asociado al proyecto de excelencia con investigador de reconocida valía P08-TIC04200, de la Junta de Andalucía, y por los proyectos de investigación nacionales I+D+i del Ministerio de Economía y Competitividad TIN2006-13425, TIN2009-13192 y TIN2012-37434; todos ellos confinanciados por los fondos FEDER de la Unión Europea.

Índice general

Página

Índice de figuras

Página

Índice de cuadros

Página

Introducción

La *Computación Natural* es una disciplina cuyo objetivo principal es el estudio y simulación de los procesos dinámicos que ocurren en la naturaleza, y que son susceptibles de ser interpretados como procedimientos de cálculo. En ella se investigan modelos y técnicas computacionales inspiradas en la naturaleza con el objetivo de entender más y mejor el mundo que nos rodea, en términos de procesamiento de información.

Dentro de este área, la *Computación Celular con Membranas* [91] es una rama emergente iniciada por Gheorghe Păun en 1998 [90]. Este nuevo modelo de computación parte del supuesto de que los procesos que tienen lugar dentro de la estructura compartimental de una célula viva pueden ser interpretados como computaciones. Los dispositivos de este modelo se denominan *sistemas P*. Estos dispositivos tienen varios ingredientes *sintácticos*: una *estructura de membranas* que consiste en una distribución jerárquica en forma de árbol enraizado, cuya raíz se denomina *piel*, delimitando *regiones* o compartimentos donde se ubican multiconjuntos de objetos y conjuntos de reglas de evolución. Los sistemas P también tienen dos ingredientes *semánticos*: su *paralelismo* inherente y *no determinismo*. Los objetos dentro de las membranas pueden evolucionar de acuerdo a unas reglas dadas, de una forma sincronizada (en el sentido de que se admite la existencia de un reloj global), paralela y no determinista.

Cabe destacar que existe un *doble paralelismo*: uno al nivel de cada región (las reglas son aplicadas de forma paralela), y otro al nivel del sistema (todas las regiones evolucionan de forma concurrente). Este paralelismo y no determinismo puede ser usado para resolver problemas computacionalmente duros en un tiempo "factible" [86]; sin embargo, debemos destacar dos consideraciones. Por un lado, tenemos que tratar con el no determinismo, de tal manera que las soluciones obtenidas con estos dispositivos sean soluciones algorítmicas en el sentido clásico [48]. Y por otro lado, el decrecimiento drástico del tiempo de ejecución, desde un orden exponencial a uno polinomial, se puede alcanzar mediante la creación de un espacio de trabajo exponencial (en forma de

membranas y/o objetos), en un tiempo polinomial (a veces lineal).

Aunque muchas investigaciones en el marco de los sistemas P se concentran en la potencia y eficiencia computacional de tales dispositivos, últimamente estos esfuerzos han ido dirigidos, también, hacia la modelización de fenómenos biológicos dentro del marco de la *Biología de Sistemas Computacional* y de la *Dinámica de Poblaciones*. Los sistemas P ofrecen una aproximación al desarrollo de modelos de sistemas biológicos cumpliendo con los requisitos de un buen marco de modelización computacional: relevancia, comprensibilidad, extensibilidad y tratabilidad computacional / matemática. En este caso, los sistemas P no son usados como un paradigma de computación, sino como un formalismo para describir el comportamiento del sistema que se modeliza. En los últimos años se han propuesto varios modelos de sistemas P para describir sistemas oscilatorios [44], traducción de señales [33, 85], control de regulación de genomas [93], quorum sensing [98, 92] y metapoblaciones [16], *algoritmos metabólicos* [17], *sistemas P con dinámica probabilística* [16] o el *algoritmo (multi-compartimental) de Gillespie* [92]. Estos modelos difieren uno del otro en el tipo de reglas de reescritura, estructura de membranas así como en la estrategia aplicada a la ejecución de reglas en los compartimentos definidos por las membranas. Además, los *sistemas P probabilístico*s han sido aplicados con éxito como una herramienta para procesos de nivel macroscópico, como el modelizado computacional de ecosistemas reales [21, 35, 22, 37].

Con el fin de *validar experimentalmente* los modelos basados en sistemas P es necesario desarrollar simuladores capaces de ser ejecutados en ordenadores electrónicos, lo que ayudaría a los investigadores a computar, analizar y extraer resultados de un modelo [91]. Estos simuladores tienen que ser lo más *eficiente* posible para poder manejar instancias de gran tamaño, lo que hoy en día es uno de los principales retos con los que se enfrentan los simuladores actuales. Las aplicaciones software para la Computación con Membranas normalmente implementan algoritmos de simulación *secuenciales* (o con un paralelismo muy limitado) adaptados a las arquitecturas de CPUs convencionales [91], por lo que carecen de capacidad para explotar la naturaleza masivamente paralela que los sistemas P presentan por definición, con el fin de obtener una simulación más cercana al modelo teórico.

Este modelo de computación paralela nos conduce a la búsqueda de tecnologías de computación altamente paralelas, donde ejecutar un simulador de forma más eficiente. En este sentido, las nuevas generaciones de *procesadores gráficos (GPUs)* implementan *un paralelismo masivo* que soportan la ejecución de miles de *hilos* de forma concurrente. Hasta hoy, muchas aplicaciones de propósito general han sido adaptadas a estas plataformas, obteniendo gran-

des *aceleraciones* comparada con las versiones secuenciales [94, 95, 5, 8]. Las GPUs actuales de NVIDIA, por ejemplo, contienen miles de elementos de procesamiento escalares por chip [60], y son programados usando extensiones del lenguaje de programación *C* [56], llamado CUDA (Compute Unified Device Architecture) [57, 5, 60, 78, 42].

El objetivo del trabajo descrito en esta memoria es *desarrollar* mejoras de *eficiencia* en los simuladores de sistemas P usando *arquitecturas paralelas*, lo cual es de vital importancia para la creación de nuevas herramientas que permitan la interactividad de los científicos con los modelos. Así, se estudia el uso de las *GPUs* de NVIDIA con *CUDA* para el desarrollo de simuladores paralelos de sistemas P. Resultados anteriores, relativos a algoritmos de simulación estocásticos en GPU [59], pusieron la semilla que alentó a considerar esta tecnología a cara de obtener resultados positivos. Para este propósito, en este trabajo se comenzó con la creación de un simulador paralelo implementado en CUDA para *sistemas P reconocedores con membranas activas* [27], ya que en este modelo teórico la capacidad de construir en tiempo polinomial, una cantidad exponencial de recursos, expresados en el número de membranas y/o de objetos, se produce de una manera natural. En este contexto, se realiza un estudio del comportamiento de la tecnología de las nuevas GPUs de NVIDIA en la simulación de sistemas P. Dado que esta simulación implica abordar un problema con naturaleza de datos dinámicos (los objetos y membranas de un sistema P se crean y eliminan a lo largo de la computación) y con una cantidad muy grande de datos (la cantidad de membranas creadas con reglas de división es exponencial), su simulación está limitado por memoria.

Después de la creación del simulador paralelo para membranas activas, se desarrolló otro simulador paralelo específico para la familia de sistemas P reconocedores con membranas activas que resuelven el problema `SAT` en tiempo lineal [86, 26]. De esta forma, diseñando un simulador *ad hoc*, se consigue un mejor rendimiento que cuando se utiliza simuladores más genéricos (flexibles) para estos sistemas P en concreto. Además, el trabajo precedente se extendió a una familia de sistemas P a modo de tejidos que resuelven el mismo problema. Estudiando ambos simuladores se puede deducir qué características de los sistemas P son mejores de simular en GPUs. Se han realizado también esfuerzos en la simulación con GPUs de sistemas P que sirven como marco de modelización en la *Dinámica de Poblaciones (sistemas PDP).* En este sentido se han propuesto dos nuevos *algoritmos de simulación* con el fin de reproducir mejor la semántica de los modelos. Estos se denominan DNDP (Direct Non-Deterministic distribution with Probabilities) [65] y DCBA (Direct distribution based on Consistent Blocks Algorithm) [63]. Después de su validación

experimental dentro de la librería estándar de simulación *pLinguaCore* [66, 64], se implementó una versión paralela del algoritmo DCBA, que funciona tanto en tecnologías de *procesadores multinúcleo* [68] como en *GPUs* [69], mejorando así el rendimiento de la simulación.

El código fuente de todos los simuladores implementados en GPU con CUDA están incluidos en el proyecto software *PMCGPU (Parallel simulators for Membrane Computing on the GPU)* [12], bajo licencia GNU GPL [2]

Contenido de la memoria

La presente memoria está estructurada en cuatro partes que conforman un total de ocho capítulos. Seguidamente, describimos sucintamente su contenido.

Parte I: Preliminares

En el **primer capítulo** se realiza una introducción a las disciplinas de la Computación Natural y la Computación Celular con Membranas. Además se muestra una descripción detallada de los componentes sintácticos y semánticos de los sistemas de membranas, o sistemas P. El capítulo finaliza con algunas aplicaciones en el marco de la complejidad computacional.

La parte introductoria de la memoria continúa en el **Capítulo 2**, donde se discute las partes comunes de un simulador de sistemas P, y se hace un sumario de las herramientas de simulación de sistemas P existentes. Además, se realiza un resumen cronológico de las versiones del marco de simulación P-Lingua [46, 11, 83] que es, propiamente, el punto de partida del trabajo aquí descrito. Finalmente, se analiza la necesidad de simuladores eficientes en la Computación Celular con Membranas, y se hace una breve descripción de los simuladores paralelos ya existentes en el área.

Esta primera parte finaliza con el **Capítulo 3**, en donde se introduce la Computación de Alto Rendimiento como fuente de soluciones para mejorar la eficiencia de los simuladores, haciendo especial hincapié en la Computación Paralela y en la computación con GPUs. Así, se presenta la tecnología CUDA y Tesla de NVIDIA que será usada para los experimentos a lo largo del trabajo.

Parte II: Simulación paralela aplicada a soluciones de problemas computacionalmente duros

En el **Capítulo 4** se describe el diseño y desarrollo de un simulador para sistemas P con membranas activas basado en CUDA y, a la vez, se presentan una serie de pruebas realizadas con la GPU Tesla C1060. Más concretamente, se analizan dos casos de prueba: un caso simple de testeo y una solución de la literatura para el problema `SAT`. De aquí se concluyen qué características deben de cumplir los sistemas P para que su simulación sea acelerada con éxito.

La segunda parte de la memoria finaliza con el **capítulo 5**, en donde se describe el desarrollo de un simulador más específico para estas soluciones, indicando la diferencia de tiempo conseguido en cada uno. Además, al tratarse de variantes distintas, se analiza cuáles son las características de los sistemas P simulados que propician una mejora del rendimiento en la GPU.

Parte III: Simulación paralela aplicada a modelos computacionales en biología

En el **Capítulo 6** se presentan los dos algoritmos de simulación diseñados para capturar mejor la semántica de los sistemas P que modelizan dinámica de poblaciones (denominados sistemas PDP, Population Dynamics P systems). Para ello, se introducen los elementos sintácticos y semánticos de los sistemas PDPs y, acto seguido, se describen los algoritmos DNDP y DCBA, que harán las funciones de motores de inferencia.

La tercera parte del documento finaliza con el **Capítulo 7**, donde se detallan los simuladores desarrollados para los algoritmos introducidos en el capítulo anterior. Se analizan tanto los simuladores creados dentro de la librería de simulación pLinguaCore, como los independientes creados en C++ con OpenMP y CUDA.

Parte IV: Conclusiones

La memoria finaliza con un capítulo dedicado a la presentación de conclusiones y planteamientos para trabajo futuro. Además, se provee una guía de buenas prácticas para desarrolladores de simuladores paralelos de sistemas P, definidos a partir de la experiencia acumulada durante el desarrollo de esta memoria. Finalmente se aportan dos capítulos apéndices, de carácter técnico. El primero muestra cómo usar los simuladores aquí descritos, y el segundo describe el servidor de GPU implementado para los experimentos realizados.

Aportaciones

Cabe destacar las siguientes aportaciones originales del trabajo realizado que se detallarán en este documento:

- *Desarrollo de un simulador paralelo basado en CUDA, para la variante de sistemas P reconocedores con membranas activas.* Este simulador recibe como entrada un sistema P descrito en un fichero binario (generado por el compilador de pLinguaCore a partir de un fichero descrito en P-Lingua). El simulador está basado en el implementado en pLinguaCore y se ayuda de la GPU para la aceleración de la simulación. Tras realizar pruebas con un ejemplo de juguete, se observa el buen comportamiento de la GPU para simular sistemas P, dado que ambos comparten una naturaleza con doble paralelismo (membranas y reglas frente a bloques de hilos e hilos). Cabe mencionar que este simulador ha sido desarrollado en colaboración con investigadores del Grupo de Arquitectura y Computación Paralela de la Universidad de Murcia. Además, este trabajo ha sido presentado en diversos congresos de diferentes áreas (a destacar el *First International Workshop on High Performance Computational Systems Biology*, Trento, Italia, 2009; el *Tenth Workshop on Membrane Computing*, Curtea de Arges, Rumanía, 2009; y el *Symposium on Application Accelerators in High-Performance Computing*, Urbana-Champaign, Illinois, EEUU, 2009), y publicado en revistas científicas, la primera de ellas incluida en el ranking JCR (ISI Journal Citation Reports), de gran impacto en la comunidad científica:

  - J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Simulation of P systems with active membranes on CUDA. *Briefings in Bioinformatics*, 11, 3 (2010), 313-322. **JCR 9.283**
  - J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Implementing P systems parallelism by means of GPUs. *Lecture Notes in Computer Science*, 5957 (2010), 227-241.

- *Desarrollo de un simulador paralelo basado en CUDA para una familia de sistemas P que resuelve* `SAT` *en tiempo lineal.* Tras la experimentación con el simulador paralelo anterior, la simulación de soluciones de sistemas P para problemas **NP**-completos no obtiene ventaja de tiempo con respecto al simulador secuencial equivalente. El rendimiento alcanzado

en el simulador flexible anterior depende directamente del sistema P a simular, y de que siga el comportamiento del simulador en la GPU. Este resultado nos lleva a estudiar el desarrollo de simuladores *ad hoc* para soluciones específicas de sistemas P. Con esto, se consigue optimizar el algoritmo de simulación y sus estructuras de datos, obteniendo una mejora de tiempo del orden de hasta 90 veces más rápido. Así, se demuestra que la GPU es una buena opción para la simulación de sistemas P, en el caso en que se consiga adaptar correctamente la simulación del sistema P concreto a la arquitectura de la GPU. Este simulador también se ha desarrollado en el marco de colaboración con investigadores de la Universidad de Murcia y de la Universidad de Málaga. Además, este trabajo ha dado lugar a nuevas aportaciones en el área de la Computación Paralela, con nuevas implementaciones más eficientes para la arquitectura GPU y supercomputador, dando lugar a una serie de contribuciones en diversos congresos de la rama de arquitectura de computadores y Computación Paralela (a destacar el *Third International Workshop on Parallel Architectures and Bioinspired Algorithms*, Viena, Austria, 2010; y el *Symposium on Application Accelerators in High Performance Computing*, Knoxville, Tennessee, EEUU, 2010), junto con las publicaciones en las siguientes revistas del ISI:

- J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Simulating a P system based efficient solution to SAT by using GPUs. *Journal of Logic and Algebraic Programming*, 79 (2010), 317-325. **JCR 0.552**
- J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, M. Ujaldón. The GPU on the simulation of cellular computing models. *Soft Computing*, 16, 2 (2012), 231-246. **JCR 1.880**

- *Desarrollo de algoritmos y simuladores para el marco de modelización basado en sistemas P probabilísticos, sistemas PDP.* Tras los éxitos conseguidos con los anteriores simuladores, el paso natural fue aplicar los conocimientos adquiridos al área de la modelización computacional en la dinámica de poblaciones. En primer lugar, se diseñaron dos nuevos algoritmos de simulación para sistemas PDP, de tal manera que la semántica probabilística se captura con más fidelidad. El algoritmo DNDP introdujo una nueva manera de simular sistemas PDP, mediante tres fases que realizaban una distribución aleatoria de las reglas sobre los objetos

disponibles en los multiconjuntos de la configuración de un tal sistema. Si bien se mejoró el comportamiento, aún no se habían satisfecho todas las expectativas; por ejemplo, se detectaron diversos modelos de ecosistemas reales donde había una gran dispersión en los resultados. Por ello, se diseñó un nuevo algoritmo denominado DCBA, que realiza una distribución proporcional de los objetos a las reglas, de tal manera que se evitaba dicha dispersión. Cabe mencionar que el desarrollo de los algoritmo se hizo en estrecha colaboración con investigadores de la Universidad de Lleida. Los trabajos elaborados fueron presentados en diversos congresos internacionales (a destacar el *IEEE Fifth International Conference on Bio-inpired Computing: Theories and Applications*, Changsha, China, 2010; y el *Thirteenth International Conference on Membrane Computing*, Budapest, Hungary, 2012), y dieron lugar las siguientes publicaciones en las siguientes revistas científicas, el primero indexado en el *ranking* del ISI:

- M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, F. Sancho-Caparrini. A simulation algorithm for multienvironment probabilistic P systems: A formal verification. *International Journal of Foundations of Computer Science*, 22, 1 (2011), 107-118. **JCR 0.379**
- M.A. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Romero-Jiménez, C. Graciani, A. Riscos-Núñez, M.A. Colomer, M.J. Pérez-Jiménez. DCBA: Simulating Population Dynamics P systems with proportional object distribution, *Lecture Notes in Computer Science*, 7762 (2013), 257-276.

- *Desarrollo de simuladores de alto rendimiento para el marco de modelización de los sistemas PDP.* Tras validar los algoritmos citados (DNDP y DCBA) con la librería de simulación pLinguaCore y comprobar el problema de tiempo y espacio consumido por ésta, el siguiente paso natural consistió en implementar simuladores eficientes para el DCBA. Primero, se creó una primera aproximación en el lenguaje C++ que consiguió mejorar el rendimiento sustancialmente y, posteriormente, se implementaron versiones paralelas en tecnologías de procesadores multinúcleo con OpenMP (obteniendo una aceleración de 2.5x) y en GPUs con CUDA (obteniendo una aceleración de 7x). En la implementación CUDA se requería un nuevo generador de números aleatorios con distribución bi-

nomial, por lo que se desarrolló la librería cuRNG_BINOMIAL, la cual está basada en la librería cuRAND. La solución está basada en la aproximación de la distribución binomial con la normal para valores altos, y un algoritmo para binomiales, BINV, para valores pequeños. Este último paso se llevó a cabo en una colaboración con el grupo HPC-Lab de la universidad noruega NTNU (Norwegian University of Science and Technology), implicando una estancia de 3 meses durante el verano de 2011. Los trabajos elaborados fueron presentados en diversos congresos internacionales (a destacar el *Tenth Conference on Computational Methods in Systems Biology*, Londres, Reino Unido, 2012; y el *First International Conference on Developments in Membrane Computing*, Sevilla, España, 2012), dando lugar a la siguiente publicación:

- M.A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A.C. Elster, M.J. Pérez-Jiménez. Population Dynamics P Systems on CUDA. *Lecture Notes in Bioinformatics*, 7605 (2012), 247-266.

Parte I

Preliminares

"Mientras me quedaba allí admirando aquella asombrosa enzima, me llamó la atención por su parecido a algo descrito en 1936 por Alan Turing, el famoso matemático británico."

Leonard Adleman

1

Computación Natural y Bioinspirada

El hombre ha tenido siempre una sublime aspiración: la constante mejora de la calidad de vida. La necesidad de resolver problemas de la vida real le ha conducido, de manera natural, a la búsqueda de procedimientos sistemáticos que permitieran obtener soluciones a través de la realización de una serie de tareas elementales debidamente secuenciadas.

La irrupción de los ordenadores electrónicos a mediados del siglo pasado, significó un avance cualitativo en la resolución de problemas concretos que, hasta ese momento, habían sido inabordables. En 1983, R. Churchouse estableció las limitaciones físicas de la velocidad de cálculo de un procesador convencional, demostrando la existencia de una cota para la velocidad y el tamaño que los microprocesadores pueden alcanzar. Esa cota, además, impediría resolver con las técnicas actuales problemas relevantes de la vida real.

Ante esa tesitura, el hombre se plantea la posibilidad de diseñar nuevas máquinas cuyo soporte físico sea distinto del electrónico. En ese escenario aparece la Naturaleza viva como fuente de inspiración dando origen a la disciplina de la *Computación Natural*, cuya finalidad es el análisis de modelos y técnicas computacionales inspiradas por la Naturaleza, tratando de comprender el mundo que nos rodea en términos de procesamiento de la información, partiendo de la base de que desde hace billones de años, en la Naturaleza se lleva produciendo una serie de procesos dinámicos que son susceptibles de ser interpretados como procedimientos de cálculo.

En este capítulo se analiza la problemática descrita, contextualizándola históricamente e introduciendo los conceptos básicos, debidamente ilustrados.

1.1. Computación Celular con Membranas

La *Computación Celular con Membranas* es una rama emergente iniciada por Gheorghe Păun a finales de 1998 [90] que está inspirada en la estructura y el funcionamiento de las células de los seres vivos.

La célula es la unidad fundamental de todo organismo vivo. Posee una estructura compleja y, a la vez, muy organizada que permite la ejecución simultánea de un gran número de reacciones químicas. La célula es la unidad más simple con la capacidad de implementar de forma autónoma los cuatro procesos biológicos que caracterizan la *Vida*: (a) replicación del ADN; (b) producción de energía; (c) síntesis de proteínas; y (d) realización de procesos metabólicos.

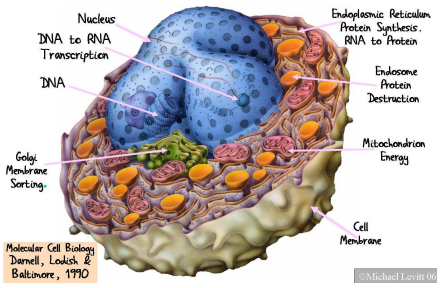


Figura 1.1: La célula eucariota

El comportamiento de una célula puede ser considerado como el de una máquina, en tanto que realiza una serie de procesos dinámicos que, en cierto sentido pueden ser interpretados como procedimientos de cálculos. Se trata, pues, de una máquina que es muy compleja desde el punto de vista estrictamente biológico.

En la presente memoria usaremos el término *simulador* de un modelo formal a una aplicación de ordenador software y/o hardware, que describe la especificación a través de un lenguaje de programación y captura la semántica a través de un algoritmo de simulación. Este algoritmo de simulación debe reproducir fielmente la dinámica, es decir, cada paso de computación del modelo formal es reproducido en el simulador mediante un número finito de pasos

(mayor que 1), por lo que el simulador es capaz de determinar los elementos básicos del modelo que intervienen de manera relevante en ese paso.

1.1.1. Sistemas P

La Computación Celular con Membranas es un paradigma de computación inspirado en la estructura y el funcionamiento de la célula de los organismos vivos en tanto en cuanto son elementos capaces de procesar y generar información. Los dispositivos computacionales de los modelos celulares se denominan genéricamente *Sistemas P* y constituyen un marco teórico de computación inspirado en la realidad de las células vivas. Los ingredientes sintácticos de estos modelos son: una *estructura de membranas* formalizado a través de un árbol enraizado, cuya raíz se denomina *piel*. Los nodos del árbol determinan regiones o compartimentos donde se ubican multiconjuntos de objetos (abstracciones de las sustancias químicas) y conjuntos de reglas (abstracciones de las reacciones químicas).

Los dispositivos P también poseen unos ingredientes semánticos fundamentales: su paralelismo inherente y su no determinismo. En estos sistemas existe un doble paralelismo: uno se encuentra a nivel de regiones o compartimentos, ya que las reglas son aplicadas de forma paralela en una región, y el otro se encuentra a nivel del sistema en general, ya que todas las regiones a la vez evolucionan de forma concurrente en cada uno de los compartimentos que conforman el sistema.

En su trabajo fundacional, Gh, Păun definió los *sistemas P de transición*. Para hablar de ellos, introduciremos algunos conceptos y notaciones que serán usados a lo largo de esta memoria.

Un *alfabeto*, Γ, es un conjunto no vacío cuyo elementos se denominan *símbolos*. Una secuencia finita de símbolos es una *cadena* o *palabra*. El conjunto de todas las cadenas sobre el alfabeto Γ se denota por Γ^*. Un *lenguaje* sobre Σ es un subconjunto de Γ^*.

Un *multiconjunto* m sobre un alfabeto Γ es un par (Γ, f) donde f es una función de Γ en $\mathbb{N}$. Si $m = (\Gamma, f)$ es un multiconjunto, entonces su *soporte* está definido por $supp(m) = \{x \in \Gamma \mid f(x) > 0\}$. Un multiconjunto es finito si su soporte es un conjunto finito. Si $m = (\Gamma, f)$ es un multiconjunto finito sobre Γ, y $supp(m) = \{a_1, \ldots, a_k\}$, entonces se notará $m = a_1^{f(a_1)} \ldots a_k^{f(a_k)}$ (el orden aquí es irrelevante), y diremos que $f(a_1) + \cdots + f(a_k)$ es el cardinal de m, denotado por $|m|$. El multiconjunto vacío se denota por $\emptyset$. Notaremos por $M_f(\Gamma)$ el conjunto de todos los multiconjuntos finitos sobre Γ.

Si $m_1 = (A, f_1)$, $m_2 = (A, f_2)$ son multiconjuntos sobre A, entonces defini-

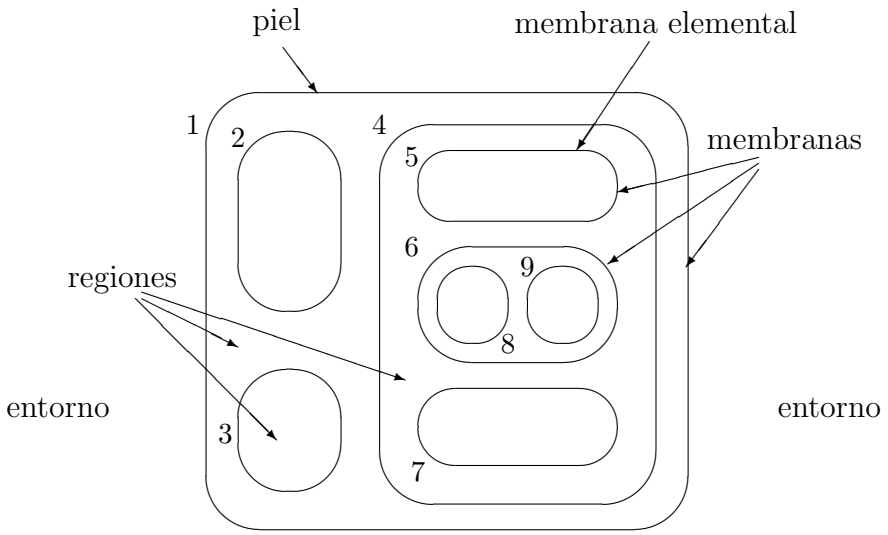


Figura 1.2: Una estructura de membranas

mos la unión de m_1 y m_2 como $m_1 + m_2 = (A, g)$, donde $g = f_1 + f_2$.

En su trabajo fundacional, Gh, Păun definió los *sistemas P de transición* de grado $q \geq 1$ como una tupla $\Pi = \big(\Gamma, \mu_{\Pi}, \mathcal{M}_1, \ldots, \mathcal{M}_q, (R_1, \rho_1), \ldots, (R_q, \rho_q), i_0\big)$ en donde: (a) Γ es un alfabeto finito denominado *alfabeto de trabajo*; (b) μ_{Π} es una *estructura de membranas* (un árbol enraizado) que consta de q membranas etiquetadas de forma unívoca desde 1 hasta q; (c) para cada i, $(1 \leq i \leq q)$, $\mathcal{M}_i$ es una cadena sobre Γ asociada a la membrana i, representando el contenido inicial y R_i es un conjunto finito de *reglas* asociado a la membrana i. Una regla es un par (u, v), que se nota por $u \to v$, en donde $u \in \Gamma$ y $v = v'$ o $v = v'\delta$, siendo $v' \in (\Gamma \times \big(\{here, out\} \cup \{in_i : i = 1, \ldots, q\}\big))^*$ y $\delta \notin \Gamma$ es un símbolo especial; (d) para cada i, $(1 \leq i \leq q)$, ρ_i es un *orden parcial estricto* sobre R_i; y (e) i_0 $(1 \leq i_0 \leq q)$ representa la *membrana de salida* del sistema.

Una *configuración* de un sistema P consta de una estructura de membranas y una familia de multiconjuntos de objetos asociados a cada membrana existente en ese instante. En la descripción de Π, existe una configuración denominada configuración inicial: $(\mu_{\Pi}, \mathcal{M}_1, \ldots, \mathcal{M}_p)$.

Una regla $u \to v$ asociada a una membrana i es *aplicable* a una configuración $\mathcal{C}$ de Π en un instante determinado, si: (a) cada objeto de u aparece en la membrana i de $\mathcal{C}$; (b) Si $(v_1, in_j) \in v$, entonces j debe ser hija de i; (c) si $\delta \in v$, entonces la membrana i no puede ser la raíz del árbol; y (d) no existe ninguna

regla de R_i que sea aplicable a esa configuración y tenga *mayor* prioridad. La ejecución de tal regla se realiza como sigue: se elimina los objetos u de la membrana i; para cada $(a, out) \in v$, un objeto $a \in \Gamma$ se incluye en la membrana padre de i (si i es la raíz, entonces el objeto a se envía al entorno). Para cada $(a, here) \in v$, un objeto $a \in \Gamma$ se añade a i. Para cada $(a, in_j) \in v$, un objeto $a \in \Gamma$ se introduce en la membrana hija etiquetada por j. Finalmente, si $\delta \in v$, entonces la membrana i se disuelve y su contenido pasará a la membrana que sea el primer antecesor no disuelto.

Las reglas del sistema serán aplicadas de manera *no determinista, paralela y maximal*. Es decir, las reglas que se van a disparar y los objetos involucrados en dichas reglas, serán elegidos de tal manera que en cada paso de computación, no pueden quedar objetos por evolucionar que *puedan* hacerlo mediante la aplicación de alguna regla. Las *relaciones de prioridad* dadas sobre las reglas, se interpretan en sentido fuerte, como sigue: si la regla r_1 tiene mayor prioridad que la regla r_2 y se puede aplicar r_1 a una configuración, entonces no se aplicará r_2 a esa configuración, aunque fuera aplicable.

Diremos que una configuración $\mathcal{C}$ es de *parada* si no existe ninguna regla del sistema que es aplicable a dicha configuración. Dadas dos configuraciones $\mathcal{C}$ y $\mathcal{C}'$ del sistema Π, diremos que $\mathcal{C}'$ se obtiene de $\mathcal{C}$ tras la ejecución de un *paso de computación* , y notaremos $\mathcal{C} \Rightarrow_{\Pi} \mathcal{C}'$, si la configuración $\mathcal{C}'$ se obtiene a partir de la configuración $\mathcal{C}$ al aplicar un multiconjunto maximal de reglas. Por tanto, el modelo de computación resultante es no determinista, en tanto que una configuración puede poseer más de una configuración siguiente.

Una *computación* en un sistema P es una sucesión (finita o infinita) de configuraciones tal que la primera de ellas es una configuración inicial y, a partir del segundo término de la sucesión, éste se obtiene de la anterior configuración mediante un paso de computación. Diremos que una *computación* es de *parada* si la computación es una sucesión finita y, además, el último término es una configuración de parada.

En este modelo, las computaciones se implementan de forma sincronizada; es decir, se supone la existencia de una especie de reloj universal que controla la evolución del sistema celular.

1.2. Sistemas P con membranas activas

A continuación se introduce una variante de sistemas celulares con membranas que permite la construcción de una cantidad de espacio exponencial en tiempo polinomial, lo que posibilitará proporcionar soluciones eficientes de

problemas **NP**–completos en este marco. [101].

La *mitosis* es un proceso de división celular que da como resultado la producción de dos células hijas a partir de una sola célula original. Este proceso ha sido introducido en el marco de los sistemas celulares con membranas a través de una regla de división de membranas dando lugar a una variante: los *sistemas P con membranas activas.*

Definición 1.1. *Un sistema P con membranas activas de grado* $q \geq 1$ *es una tupla* $\Pi = (\Gamma, H, \mu_\Pi, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$*, en donde:*

- Γ *es un alfabeto finito, denominado alfabeto de trabajo.*
- H *es un conjunto finito disjunto con* Γ*.*
- μ_Π *es un árbol enraizado etiquetado (de manera inyectiva) con elementos de* $H \times \{0, +, -\}$*.*
- $\mathcal{M}_1, \ldots, \mathcal{M}_q$ *son cadenas sobre el alfabeto de trabajo* Γ*.*
- $\mathcal{R}$ *es un conjunto finito de reglas de los tipos siguientes:*
  - (*a*) $[a \to v]_h^\alpha$*, donde* $a \in \Gamma$*,* $v \in \Gamma^*$*,* $\alpha \in \{0, +, -\}, h \in H$ *(regla de evolución).*
  - (*b*) $[a]_h^\alpha \to b[\ ]_h^\beta$*, donde* $a, b \in \Gamma$*,* $\alpha, \beta \in \{0, +, -\}, h \in H$ *(regla de comunicación hacia fuera).*
  - (*c*) $a[\ ]_h^\alpha \to [b]_h^\beta$*, donde* $a, b \in \Gamma$*,* $\alpha, \beta \in \{0, +, -\}, h \in H$*, h no forma parte de la etiqueta de la raíz del árbol (regla de comunicación hacia dentro).*
  - (*d*) $[a]_h^\alpha \to b$*, donde* $a, b \in \Gamma$*,* $\alpha \in \{0, +, -\}$*,* $h \in H$*, h no forma parte de la etiqueta de la raíz del árbol (regla de disolución).*
  - (*e*) $[a]_h^\alpha \to [b]_h^\beta [c]_h^\gamma$*, donde* $a, b, c \in \Gamma$*,* $\alpha, \beta, \gamma \in \{0, +, -\}$*,* $h \in H$*, h no forma parte de la etiqueta de la raíz del árbol (regla de división).*
- $i_{out} \in H \cup \{env\}$*, donde* $env \notin H \cup \Gamma$*.*

Un sistema P con membranas activas $\Pi = (\Gamma, H, \mu_\Pi, \mathcal{M}_1, \ldots, \mathcal{M}_p, \mathcal{R}, i_{out})$ de grado $q \geq 1$ puede ser considerado como un conjunto de q membranas dispuestos según una estructura de árbol enraizado, de tal manera que cada nodo (membrana) está etiquetado por un elemento de H y posee una carga eléctrica positiva (+), negativa (-), o neutra (0). En adelante, cuando hablemos de la etiqueta de una membrana nos referiremos al elemento de H que aparece,

propiamente, en esa etiqueta. Las reglas del sistema están asociadas a etiquetas no a membranas y las reglas del tipo (e) permiten la existencia de distintas membranas en el sistema con la misma etiqueta.

Seguidamente se definen cómo se aplican las reglas del sistema.

(a) *Reglas de evolución*: la regla $[a \to v]_h^\alpha$, será aplicable a toda membrana h con carga α y contenga el objeto a. Su aplicación no altera ni la etiqueta ni la carga de la membrana en la que se aplica. Su ejecución produce la sustitución de un objeto a por un multiconjunto v de objetos.

(b) *Reglas de comunicación hacia fuera*: la regla $[a]_h^\alpha \to b[\ ]_h^\beta$ será aplicable a toda membrana h con carga α y contenga el objeto a. Su aplicación no altera la etiqueta de la membrana pero, en cambio, la carga pasa a ser β. Además, su ejecución produce la eliminación de un objeto a de la membrana h y la aparición de un objeto b en la membrana padre de h (si h es la membrana piel, el objeto b pasaría al entorno del sistema).

(c) *Reglas de comunicación hacia dentro*: la regla $a[\ ]_h^\alpha \to [b]_h^\beta$ será aplicable a toda membrana h, distinta de la membrana piel, con carga α y tal que el objeto a aparece en la membrana padre de h. Su aplicación no altera la etiqueta de la membrana h pero, en cambio, la carga pasa a ser β. Además, su ejecución produce la eliminación de un objeto a de la membrana padre de h y la aparición de un objeto b en la membrana h.

(d) *Reglas de disolución*: la regla $[a]_h^\alpha \to b$, será aplicable a toda membrana h, distinta de la piel, con carga α y contenga al objeto a. Su aplicación provoca la eliminación de laa membrana h del sistema y su contenido pasará a la membrana que sea el primer antecesor de h no disuelto.

(e) *Reglas de división*: la regla $[a]_h^\alpha \to [b]_h^\beta[c]_l^\gamma$ será aplicable a toda membrana h que sea elemental y distinta de la piel, que posea carga α y contenga el objeto a. Su aplicación produce la creación de dos membranas con la misma etiqueta, la primera de ellas con carga β y la segunda con carga γ. El objeto a que dispara la regla desaparece y, en su lugar, aparece b en la primera de las membranas creadas y c en la segunda de esas membranas. Los restantes objetos que aparecían en la membrana donde se ha aplicado la regla, son replicados en las dos membranas creadas.

Las reglas del sistema se aplicarán de manera paralela y maximal, de tal forma que: (1) una regla de división se puede ejecutar sobre una membrana simultáneamente con reglas de evolución; y (2) en cada paso de computación, a una

membrana sólo se le puede aplicar, *a lo sumo*, una regla de entre las que no son de tipo (a).

Los conceptos de configuración, paso de computación de una configuración a otra y computación, se definen en el marco de los sistemas P con membranas activas de forma similar al de los sistemas P de transición.

Todas las computaciones comienzan con una configuración inicial y evoluciona de acuerdo con lo indicado anteriormente. Únicamente las *computaciones de parada* proporcionan un resultado que estará codificado por los objetos presentes en una cierta región de salida en la *configuración de parada*. Esa región puede ser una membrana ordinaria o bien el entorno.
Notación: Si $\mathcal{C} = \{\mathcal{C}_t\}_{t<r+1}$ $(r \in \mathbf{N})$ es una computación de parada de Π, entonces diremos que la longitud de $\mathcal{C}$ es r y la notaremos por $|\mathcal{C}|$. Así mismo, notaremos por $\mathcal{C}_t(h)$ el contenido de la membrana h en la configuración $\mathcal{C}_t$.

1.3. Sistemas P a modo de tejidos

En esta sección vamos a presentar la segunda variante de sistema celular con membranas en cuyo marco vamos a proporcionar una solución eficiente del problema `SAT` de la satisfactibilidad de la Lógica Proposicional.

El marco computacional de los sistemas P con membranas activas puede ser extendido a sistemas celulares que están organizados a modo de tejidos. En el nuevo contexto van a existir unidades fundamentales que denominaremos *células* y un elemento singular que denominaremos *entorno* del sistema. Las células del sistema estarán identificadas a través de una etiqueta y se pueden comunicar entre sí o con el entorno, a través de ciertas reglas. Así mismo, admitiremos la existencia de un tipo de reglas que implementa la división celular. Estos modelos se denominarán *sistemas P de tejidos con división celular*.

Definición 1.2. *Un sistema de tejidos con división celular de grado $q \geq 1$ es una tupla $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, tal que:*

1. *Γ es un* alfabeto *finito.*
2. *$\mathcal{E} \subseteq \Gamma$.*
3. *$\mathcal{M}_1, \ldots, \mathcal{M}_q$ son cadenas sobre Γ.*
4. *$\mathcal{R}$ es un conjunto finito de reglas del siguiente tipo:*
   (a) Reglas de Comunicación*: $(i, u/v, j)$, para $i, j \in \{0, 1, 2, \ldots, q\}, i \neq j$, $u, v \in \Gamma^*$, $|u + v| \neq 0$;*

(b) Reglas de División*:* $[a]_i \rightarrow [b]_i[c]_i$*, donde* $i \in \{1, 2, \ldots, q\}$*,* $i \neq i_{out}$ *y* $a, b, c \in \Gamma$.

5. $i_{out} \in \{0, 1, 2, \ldots, q\}$.

Un *sistema de tejidos con división celular* $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$ de grado $q \geq 1$, se puede ver como un conjunto de q células, etiquetadas de 1 a q, con un entorno que será etiquetado por 0, y de tal manera que: (a) $\mathcal{M}_1, \ldots, \mathcal{M}_q$ son cadenas del alfabeto Γ que representan los multiconjuntos finitos de objetos (elementos del alfabeto de trabajo Γ) inicialmente colocados en las q células del sistema; (b) $\mathcal{E}$ es el conjunto de objetos inicialmente localizados en el entorno del sistema, todos ellos aparecen en un *número arbitrario de copias*; y (c) $i_{out} \in \{0, 1, 2, \ldots, q\}$ representa una *región* distinguida que nos va a permitir codificar la salida del sistema. Usaremos el termino *región* i $(0 \leq i \leq q)$ para referirnos a la célula i, en el caso $1 \leq i \leq q$, o al entorno, en el caso $i = 0$.

Una regla de comunicación $(i, u/v, j)$, para $i, j \in \{0, 1, 2, \ldots, q\}, i \neq j$, $u, v \in \Gamma^*$, es aplicable a todo par de regiones distintas i, j tales que la región i contiene el multiconjunto de objetos u y la región j contiene el multiconjunto de objetos v. La aplicación de dicha regla provoca que el multiconjunto de objetos u pase de la región i a la región j y, simultáneamente, el multiconjunto de objetos v pase de la región j a la región i.

Una regla de división $[a]_i \rightarrow [b]_i[c]_i$ es aplicable a una célula i tal que contiene el objeto a. La aplicación de dicha regla produce la creación de dos nuevas células con la misma etiqueta i, de tal manera que en la primera célula creada, el objeto a es reemplazado por b, y en la segunda, el objeto a se sustituye por c. Los restantes objetos existentes en la célula i, se replican y se copian en las dos células nuevas. La célula de salida i_{out} no puede ser dividida.

Las reglas de un sistema P de tejidos se ejecutan en paralelo, de manera no determinista y maximal, tal y como suele ser habitual en sistemas celulares con membranas. En la aplicación de las reglas existe una restricción: si en un cierto paso, se está aplicando una regla de división a una célula, entonces en ese paso sólo se aplicará esa regla a esa célula.

Una *configuración* de un sistema P de tejidos en un instante está descrita por los multiconjuntos de objetos sobre el alfabeto Γ asociados a todas las células presentes en el sistema, así como el multiconjunto de objetos sobre $\Gamma - \mathcal{E}$ asociados al entorno en ese momento. La *configuración inicial* del sistema $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$ es $(\mathcal{M}_1, \cdots, \mathcal{M}_q; \emptyset)$. De manera similar a como se ha hecho en las secciones anteriores, se introduce el concepto de computación, utilizándose la misma notación que la indicada en la sección 1.2.

1.4. Resolución eficiente de problemas en sistemas celulares

En esta sección vamos a usar el término *sistema celular* para referirnos indistintamente a un sistema P con membranas activas o a un sistema P de tejidos con división celular. El objetivo de esta sección es introducir el concepto de resolubilidad eficiente de un problema de decisión a través de una familia de sistemas celulares.

Un problema de decisión, X, es un par ordenado (I_X, θ_X), en donde I_X es un lenguaje sobre un alfabeto finito (cuyos elementos se denominan *instancias*) y θ_X es una función booleana total sobre I_X. Existe una correspondencia biunívoca natural entre problemas de decisión y lenguajes asociados. Los *sistemas celulares reconocedores* serán utilizados para atacar la resolución de problemas de decisión.

Definición 1.3. *Diremos que un sistema celular con membranas* Π *es un sistema reconocedor si satisface las condiciones siguientes:*

- *El alfabeto de trabajo* Γ *del sistema contiene dos objetos distinguidos* `yes` *y* `no`*. En sistemas P de tejidos, al menos una copia de cada uno tiene que estar presente en alguno de los multiconjuntos iniciales del sistema.*
- *Existe un alfabeto (de entrada)* Σ *estrictamente contenido en* Γ*. En sistemas P de tejidos se verifica* $\Sigma \cap \mathcal{E} = \emptyset$*, siendo* $\mathcal{E}$ *el alfabeto del entorno.*
- *Los multiconjuntos iniciales del sistema* $\mathcal{M}_1, \ldots, \mathcal{M}_q$ *son sobre* $\Gamma \setminus \Sigma$*.*
- *Existe una membrana o célula de entrada* i_{in}*.*
- *La región de salida será el entorno.*
- *Todas las computaciones paran.*
- *Si* $\mathcal{C}$ *es una computación de* Π*, entonces al menos uno de los objetos* `yes` *o* `no`*, pero no ambos, debe haber sido enviado a la región de salida, y sólo en el último paso de la computación.*

Para cada multiconjunto m sobre Σ, la *configuración del sistema* Π *con entrada* $m \in \Sigma^*$ es $(\mathcal{M}_1, \ldots, \mathcal{M}_{i_{in}} + m, \ldots, \mathcal{M}_q, \emptyset$.

Si Π es un sistema celular reconocedor y $\mathcal{C}$ es una computación de parada de Π, entonces el resultado es `yes` (respectivamente, `no`) si este objeto aparece en el entorno asociado a la configuración de parada y, en cambio, ni el objeto

`yes` ni el objeto `no` aparecen en el entorno de ninguna configuración anterior. Diremos que $\mathcal{C}$ es de *aceptación* (respectivamente, de *rechazo*) si la respuesta es afirmativa (respectivamente, negativa).

Notaremos por **AM** (respectivamente, **TDC**) la clase de todos los sistemas P reconocedores con membranas activas (respectivamente, sistemas P de tejidos reconocedores con división celular).

1.4.1. Clases de complejidad polinomial en sistemas celulares

Los sistemas celulares proporcionan dispositivos no deterministas y, en consecuencia, será interesante que en nuestra definición aparezcan elementos diferenciadores en relación con el concepto de *aceptación* clásico de las máquinas de Turing no deterministas.

Definición 1.4. *Un problema de decisión $X = (I_X, \theta_X)$ es resoluble en tiempo polinomial por una familia $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ de sistemas celulares reconocedores si se verifican las siguientes propiedades:*

- *La familia $\mathbf{\Pi}$ es* polinomialmente uniforme *por máquinas de Turing; es decir, existe una máquina de Turing determinista que trabaja en tiempo polinomial y tal que construye el sistema $\Pi(n)$ a partir de $n \in \mathbb{N}$.*
- *Existe un par de funciones (cod, s) sobre I_X que son computables en tiempo polinomial, de tal manera que satisfacen lo siguiente:*
  - *Para cada instancia $u \in I_X$, $s(u)$ es un número natural y $cod(u)$ es un multiconjunto de entrada del sistema $\Pi(s(u))$.*
  - *Para cada $n \in \mathbb{N}$, el conjunto $s^{-1}(n)$ es finito.*
  - *La familia $\mathbf{\Pi}$ is* polinomialmente acotada *con respecto a (X, cod, s); es decir, existe una función polinómica $p(n)$, tal que para cada instancia $u \in I_X$, toda computación de $\Pi(s(u))$ con entrada $cod(u)$ es de parada y realiza, a lo sumo, $p(|u|)$ pasos.*
  - *La familia $\mathbf{\Pi}$ es* adecuada *con respecto a (X, cod, s); es decir, para cada instancia $u \in I_X$, si existe una computación de aceptación de $\Pi(s(u))$ con entrada $cod(u)$, entonces $\theta_X(u) = 1$.*
  - *La familia $\mathbf{\Pi}$ es* completa *con respecto a (X, cod, s); es decir, para cada $u \in I_X$, si $\theta_X(u) = 1$, entonces toda computación de $\Pi(s(u))$ con entrada $cod(u)$ es de aceptación.*

De la adecuación y completitud se deduce que todo sistema de la familia es *confluente*, en el siguiente sentido: cada computación de dicho sistema con el mismo multiconjunto de entrada, debe dar siempre la misma respuesta.

Si $\mathbf{R}$ una clase de sistemas celulares reconocedores, notaremos $\mathbf{PMC_R}$ al conjunto de problemas de decisión que pueden ser resueltos en tiempo polinomial por familias de sistemas de $\mathbf{R}$.

1.5. Soluciones del problema SAT basadas en sistemas P

En esta sección, vamos a proporcionar dos soluciones eficientes del problema `SAT` a través de familias de sistemas P con membranas activas y de familias de sistemas P de tejidos con división celular, de acuerdo con la Definición 1.4.

1.5.1. Una solución de SAT a través de sistemas P con membranas activas

La aplicación f de $\mathbb{N}\times\mathbb{N}$ en $\mathbb{N}$ definida por $f(m,n)=\frac{(m+n)\cdot(m+n+1)}{2}+m$ es una función computable y biyectiva entre los conjuntos citados. Usualmente, notaremos $f(m,n)=\langle m,n\rangle$. Para cada par de números naturales $m,n\in\mathbf{N}$ consideramos el sistema P reconocedor con membranas activas $\Pi(\langle m,n\rangle)=(\Gamma,\Sigma,\mu,\mathcal{M}_1,\mathcal{M}_2,R,2)$ de grado 2, definido como sigue:

- El alfabeto de entrada es $\Sigma=\{x_{i,j},\overline{x}_{i,j}:1\leq i\leq m,1\leq j\leq n\}$.
- El alfabeto de trabajo es

$$\begin{aligned}\Gamma \;=\; & \Sigma\cup\{c_k:1\leq k\leq m+2\}\;\cup\;\{d_k:1\leq k\leq 3n+2m+3\}\cup\\ & \cup\{r_{i,k}:0\leq i\leq m,1\leq k\leq 2n\}\;\cup\{e,t\}\;\cup\;n\{Yes,No\}\end{aligned}$$

- El conjunto de etiquetas es $\{1,2\}$.
- La estructura inicial de membranas es $\mu=[\;[\;]_2\;]_1$.
- Los multiconjuntos iniciales son $\mathcal{M}_1=\emptyset$ y $\mathcal{M}_2=\{d_1\}$.
- La membrana de entrada es la etiquetada por 2.
- El conjunto R consiste de las siguientes reglas:

(a) $\{[d_k]_2^0 \to [d_k]_2^+[d_k]_2^- : \ 1 \leq k \leq n\}$.

(b) $\{[x_{i,1} \to r_{i,1}]_2^+, \ [\ \overline{x}_{i,1} \to r_{i,1}]_2^- : \ 1 \leq i \leq m\}$. $\{[x_{i,1} \to \lambda]_2^-, \ [\overline{x}_{i,1} \to \lambda]_2^+ : \ 1 \leq i \leq m\}$.

(c) $\{[x_{i,j} \to x_{i,j-1}]_2^+, \ [x_{i,j} \to x_{i,j-1}]_2^- : \ 1 \leq i \leq m, \ 2 \leq j \leq n\}$. $\{[\ \overline{x}_{i,j} \to \ \overline{x}_{i,j-1}]_2^+, \ [\ \overline{x}_{i,j} \to \ \overline{x}_{i,j-1}]_2^- : \ 1 \leq i \leq m, \ 2 \leq j \leq n\}$.

(d) $\{[d_k]_2^+ \to [\ ]_2^0 d_k, \ , \ [d_k]_2^- \to [\ ]_2^0 d_k : \ 1 \leq k \leq n\}$. $\{d_k[\ ]_2^0 \to [d_{k+1}]_2^0 : \ 1 \leq k \leq n-1\}$.

(e) $\{[r_{i,k} \to r_{i,k+1}]_2^0 : \ 1 \leq i \leq m, \ 1 \leq k \leq 2n-1\}$.

(f) $\{[d_k \to d_{k+1}]_1^0 : \ n \leq k \leq 3n-3\}$; $[d_{3n-2} \to d_{3n-1}e]_1^0$.

(g) $e[\ ]_2^0 \to [c_1]_2^+$; $[d_{3n-1} \to d_{3n}]_1^0$.

(h) $\{[d_k \to d_{k+1}]_1^0 : \ 3n \leq k \leq 3n+2m+2\}$.

(i) $[r_{1,2n}]_2^+ \to [\ ]_2^- r_{1,2n}$.

(j) $\{[r_{i,2n} \to r_{i-1,2n}]_2^- : \ 1 \leq i \leq m\}$.

(k) $r_{1,2n}[\ ]_2^- \to [r_{0,2n}]_2^+$.

(l) $\{[c_k \to c_{k+1}]_2^- : \ 1 \leq k \leq m\}$.

(m) $[c_{m+1}]_2^+ \to [\ ]_2^+ c_{m+1}$.

(n) $[c_{m+1} \to c_{m+2}t]_1^0$.

(o) $[t\ ]_1^0 \to [\ ]_1^+ t$.

(p) $[c_{m+2}]_1^+ \to [\ ]_1^- Yes$.

(q) $[d_{3n+2m+3}]_1^0 \to [\ ]_1^+ No$.

Sea $\varphi = C_1 \wedge \cdots \wedge C_m$ una fórmula proposicional en FNC tal que $Var(\varphi) = \{x_1, \ldots, x_n\}$ y $C_i = y_{i,1} \vee \cdots \vee y_{i,k_i}$, $1 \leq i \leq m$, donde $y_{i,i'} \in \{x_j, \neg x_j : 1 \leq j \leq n\}$ son los literales de φ. Consideramos las funciones $cod(\varphi) = \bigcup_{i=1}^m \{x_{i,j} : x_j \in C_i\} \cup \{\overline{x}_{i,j} : \neg x_j \in C_i\}$ y $s(\varphi) = \langle m, n \rangle = \frac{(m+n)\cdot(m+n+1)}{2} + m$. La ejecución del sistema $\Pi(s(\varphi))$ con entrada $cod(\varphi)$ se estructura en cuatro fases:

- *Fase de generación de valoraciones*: se generan todas las posibles valoraciones de verdad del conjunto $\{x_1, \ldots, x_n\}$ y se ejecuta en $3n-1$.
- *Fase de sincronización*: se prepara al sistema para la fase de chequeo y consume $2n$ pasos.
- *Fase de chequeo*: se determina cuántas cláusulas son verdaderas por cada valoración y se ejecuta en $2m$ pasos.
- *Fase de salida*: el sistema proporciona la salida correspondiente en función del chequeo anterior.

1.5.2. Una solución de SAT a través de sistemas P de tejidos con división celular

Para cada $m, n \in \mathbf{N}$ consideramos el sistema P de tejidos reconocedor con división celular $\Pi(\langle m, n\rangle) = (\Gamma, \Sigma, \mu, \mathcal{M}_1, \mathcal{M}_2, R, 2)$ definido así:

- El alfabeto de entrada es $\Sigma = \{x_{i,j}, y_{i,j} : \ 1 \leq i \leq n, \ 1 \leq j \leq m\}$
- El alfabeto de trabajo es

$$\begin{aligned}\Gamma \ = \ & \Sigma \cup \{a_i, t_i, f_i \mid 1 \leq i \leq n\} \cup \{r_i \mid 1 \leq i \leq m\} \cup \\ & \cup \{T_i, F_i \mid 1 \leq i \leq n\} \cup \{T_{i,j}, F_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq m+1\} \cup \\ & \cup \{b_i \mid 1 \leq i \leq 2n+m+1\} \cup \{c_i \mid 1 \leq i \leq n+1\} \cup \\ & \cup \{d_i \mid 1 \leq i \leq 2n+2m+nm+1\} \cup \{e_i \mid 1 \leq i \leq 2n+2m+nm+3\} \cup \\ & \cup \{f, g, \texttt{yes}, \texttt{no}\}\end{aligned}$$

- El alfabeto del entorno es $\mathcal{E} = \Gamma - \{\texttt{yes}, \texttt{no}\}$.
- El conjunto de etiquetas es $\{1, 2\}$.
- Los multiconjuntos iniciales son $\mathcal{M}_1 = \{\texttt{yes}, \texttt{no}, b_1, c_1, d_1, e_1\}$ y $\mathcal{M}_2 = \{f, g, a_1, a_2, \ldots, a_n\}$.
- La célula de entrada es la etiquetada por 2.
- El conjunto $\mathcal{R}$ está formado por las siguientes reglas:

1. **Regla de división:**
   (a) $[\ a_i\ ]_2 \rightarrow [\ T_i\ ]_2 [\ F_i\ ]_2$, para $i = 1, 2, \ldots, n$.
2. **Reglas de comunicación:**
   (b) $(1, b_i/b_{i+1}^2, 0)$, para $i = 1, \ldots, n$,
   (c) $(1, c_i/c_{i+1}^2, 0)$, para $i = 1, \ldots, n$,
   (d) $(1, d_i/d_{i+1}^2, 0)$, para $i = 1, \ldots, n$,
   (e) $(1, e_i/e_{i+1}, 0)$, para $i = 1, \ldots, 2n+2m+nm+2$.
   (f) $(1, b_{n+1}c_{n+1}/f, 2)$,
   (g) $(1, d_{n+1}/g, 2)$.
   (h^*) $(1, f^2/f, 0)$,
   (h) $(2, c_{n+1}T_i/c_{n+1}\ T_{i,1}, 0)$,
   (i) $(2, c_{n+1}F_i/c_{n+1}\ F_{i,1}, 0)$, para $i = 1, \ldots, n$,
   (j) $(2, T_{i,j}/t_i\ T_{i,j+1}, 0)$,
   (k) $(2, F_{i,j}/f_i\ F_{i,j+1}, 0)$, para $i = 1, \ldots, n$ y $j = 1, \ldots, m$.

(l) $(2, b_i/b_{i+1}, 0)$,
(m) $(2, d_i/d_{i+1}, 0)$, para $i = n+1, \ldots, 2n+m$.
(n) $(2, b_{2n+m+1}\ t_i\ x_{i,j}/b_{2n+m+1}\ r_j, 0)$,
(o) $(2, b_{2n+m+1}\ f_i\ y_{i,j}/b_{2n+m+1}\ r_j, 0)$, para $1 \leq i \leq n$ y $1 \leq j \leq m$,
(p) $(2, d_i/d_{i+1}, 0)$, para $i = 2n+m+1, \ldots, 2n+m+nm$.
(q) $(2, d_{2n+m+nm+j}\ r_j/d_{2n+m+nm+j+1}, 0)$, para $j = 1, \ldots, m$.
(r) $(2, d_{2n+2m+nm+1}/f\ \texttt{yes}, 1)$.
(s) $(2, \texttt{yes}/\lambda, 0)$.
(t) $(1, e_{2n+2m+nm+3}\ f\ \texttt{no}/\lambda, 0)$

Sea $\varphi = C_1 \wedge \cdots \wedge C_m$ una fórmula proposicional en FNC tal que $Var(\varphi) = \{x_1, \ldots, x_n\}$, y $C_i = y_{i,1} \vee \cdots \vee y_{i,k_i}$, $1 \leq i \leq m$, donde $y_{i,i'} \in \{x_j, \neg x_j : 1 \leq j \leq n\}$ son los literales de φ. Consideramos las funciones $cod(\varphi) = \bigcup_{i=1}^{m}\{x_{i,j} : x_i \in C_j\} \cup \{\overline{x}_{i,j} : \neg x_i \in C_j\}$ y $s(\varphi) = \langle m, n\rangle = \frac{(m+n)\cdot(m+n+1)}{2} + m$. La ejecución del sistema á $\Pi(s(\varphi))$ con entrada $cod(\varphi)$ se estructura en seis fases:

- *Fase de generación de valoraciones*: se generan todas las posibles valoraciones de verdad del conjunto de variables $\{x_1, \ldots, x_n\}$ y consume n pasos.
- *Fase de generación de contadores*: se generan contadores necesarios para el desarrollo sincronizado de las computaciones.
- *Fase de preparación para el chequeo*: se prepara al sistema para realizar el chequeo de cláusulas.
- *Fase de chequeo de cláusulas*: se determina qué cláusulas son verdaderas por cada valoración de verdad. Esta fase consume nm pasos.
- *Fase de chequeo de la fórmula*: se determina si existe alguna valoración que haga verdadera todas las cláusulas.
- *Fase de salida*: el sistema proporciona la salida correspondiente en función de lo analizado anteriormente.

Se prueba que la familia $\mathbf{\Pi}$ antes definida, es polinomialmente uniforme por máquinas de Turing deterministas ya que los sistemas han sido definidos a través de expresiones recursivas y los recursos necesarios para describir $\Pi(\langle m, n\rangle)$ es cuadrático en máx$\{m, n\}$: (a) el tamaño del alfabeto es $6nm+12n+7m+12 \in \Theta(nm)$; (b) el número de células iniciales es $2 \in \Theta(1)$; (c) el número inicial de objetos es $n + 8 \in \Theta(n)$; (d) el número de reglas es $4nm + 10n + 3m + 16 \in \Theta(nm)$; y el límite superior de longitud de las reglas es $5 \in \Theta(1)$.

1.6. Aplicaciones de los sistemas P

Los sistemas celulares con membranas proporcionan un marco de computación que se caracteriza por ser un modelo próximo a la realidad de la célula, desde un punto de vista biológico. Son modelos matemáticos con propiedades muy atractivas desde el punto de vista computacional. Algunos sistemas celulares son computacionalmente completos en el sentido de ser capaces de resolver cualquier problema resoluble por máquinas de Turing. Además, otros sistemas son eficientes desde el punto de vista computacional en tanto que proporciona soluciones polinomiales de problemas computacionalmente duros.

En su origen, los sistemas celulares tienen inspiración biológica. No obstante, proporcionan un buen marco teórico que posibilita abordar cuestiones en otros ámbitos muy diferentes de la biología. Por ejemplo, las redes de ordenadores, como Internet, puede ser representados como una estructura de membranas en la que los nodos que forman la red se pueden interpretar como las membranas del sistema, y el paso de información entre los ordenadores o nodos vienen a ser las reglas que se generan o se disparan. Así mismo, sistemas denominados complejos, como aquellos que surgen en el estudio de la dinámica de poblaciones, pueden ser interpretados como sistemas celulares, en los que los individuos de las distintas especies interaccionan entre sí dentro de hábitats que permiten el trasvase de los mismos. Los sistemas P proporcionan un formalismo computacional para describir el comportamiento del sistema a ser modelizadas. En los últimos años, se han propuesto varios modelos de sistemas P para describir sistemas celulares oscilatorios [44], traducción de señales [85], control de regulación de genomas [93], quorum sensing [92], dinámica de poblaciones [87], sistemas metabólicos [61, 24, 62], y ecosistemas reales [23, 22, 35]. Las distintas variantes presentadas se diferencian unas de otras en algunos aspectos sintácticos o semánticos: en el tipo de reglas, en la estructura de membranas, o bien en la estrategia aplicada a la ejecución de reglas en los compartimentos definidos por las membranas.

"Debemos evitar decir que disponemos de 'implementaciones' de sistemas P, debido al inherente no determinismo y paralelismo masivo del modelo básico, características que no se pueden aplicar, en principio, en los equipos electrónicos habituales - pero que podrían ser implementados en un hardware dedicado y reconfigurable [...] o en una red local."

Gheorghe Păun (2005)

2

Aplicaciones Software para la Computación Celular con Membranas

Actualmente no existe ninguna implementación biológica, ya sea *in vivo* o *in vitro*, de los sistemas P. La única forma de analizar y ejecutar tales dispositivos es mediante programas informáticos corriendo sobre computadores electrónicos que están limitados por las leyes de la física. Por tanto, los simuladores de sistemas P son herramientas que asisten a los investigadores en la extracción resultados de los modelos y validación experimental de los mismos, mediante la reproducción de su computación.

Sin embargo, estos simuladores deben de ser lo más eficientes posible en términos de tiempo de computación para servir como herramientas útiles con un corto tiempo de respuesta. A este efecto, las plataformas paralelas son nuestro objeto de estudio, ya que pueden implementar efectivamente el paralelismo de los sistemas P (normalmente de manera parcial).

Este capítulo presenta un estudio de la simulación de sistemas P. Primero, se discute la arquitectura típica de los simuladores de sistemas P (Sección 2.1) y el marco de simulación P-Lingua (Sección 2.2). Finalmente, mostramos una caracterización de las plataformas paralelas utilizadas para este propósito (Sección 2.3), y las soluciones paralelas desarrolladas hasta el momento (Sección 2.4).

2.1. Simuladores de sistemas P

Los sistemas P son dispositivos bioinspirados que trabajan de una forma masivamente paralela y no determinista. Aunque existen estudios preliminares que analizan los problemas relacionados con su implementación real, aún queda bastante camino para alcanzar dicho hito. Es por ello que la *simulación* de sistemas P usando computadores convencionales se convierte en una necesidad vital para el avance de las actividades científicas en paradigmas de computación que están por ser implementados (en particular, la Computación Celular con Membranas).

Los simuladores P proporcionan tres funcionalidades principales para la comunidad científica, así los simuladores puede ser usados con *propósitos educacionales*, de *asistencia en el diseño* de modelos de sistemas P, o como *núcleo de herramientas software más elaboradas*.

En los últimos años se ha avanzado mucho en el desarrollo de simuladores [39]. La mayoría de ellos muestran una arquitectura software típica, incluyendo tres módulos principales: *entrada* (la definición del sistema P), *núcleo* (motor de simulación), y *salida* (presentación de resultados).

2.2. El proyecto software P-Lingua

Cada modelo de sistema P muestra unas restricciones semánticas que determinan la manera en que las reglas son aplicadas. Por tanto, los simuladores tienen que ser capaces de soportar diferentes escenarios. Además, los simuladores deben recibir una definición precisa del sistema P que va a ser simulado. Durante esta memoria, el término *entrada del simulador* se referirá a esta definición.

En P-Lingua [83, 11], se considera una aproximación para la definición de la entrada del simulador donde ésta se estandariza, de tal manera que los simuladores procesarán siempre el mismo formato. P-Lingua es en sí un lenguaje de definición para describir sistemas P. El proyecto P-Lingua provee herramientas software libre bajo licencia GNU GPL [2], para la compilación, simulación y depuración, las cuales están integradas en la biblioteca Java llamada *pLinguaCore*. La biblioteca también incluye varios simuladores secuenciales para reproducir las computaciones de las variantes de sistemas P soportadas. Adicionalmente, se pueden exportar ficheros de definición de P-Lingua a otros formatos de fichero (libres de errores) con el fin de dar interoperabilidad entre los distintos entornos software de simulación (este es el método principal para obtener la entrada de los simuladores implementados en este trabajo).

Hay tres ventajas principales por el cual se decide usar P-Lingua y pLinguaCore para este trabajo. Primero, P-Lingua es un formato estándar para definir sistemas P que se puede traducir a otros formatos sintácticamente libre de errores. Segundo, una manera natural para implementar nuevos algoritmos de simulación pasa por primero desarrollar un simulador en pLinguaCore, ya que este proceso en esta biblioteca software es relativamente sencillo. Tercero, el mismo fichero de entrada P-Lingua puede ser usado tanto para los simuladores desarrollados como por los simuladores de pLinguaCore. Esto conforma un buen método para validar los simuladores desarrollados.

2.3. Plataformas de simulación de sistemas P

Una buena plataforma de computación para simular sistemas P debería proveer un buen balance de rendimiento, flexibilidad, escalabilidad, paralelismo y coste [52]. Las tres primeras propiedades son consideradas los atributos de calidad más importantes de las plataformas para aplicaciones de Computación con Membranas [75]:

- *Rendimiento*: se refiere a la velocidad a la cual la plataforma es capaz de simular sistemas P. Una medida adecuada para ello puede ser el número de aplicaciones de reglas que se realizan por unidad de tiempo.

- *Flexibilidad*: significa la habilidad de la plataforma para soportar la ejecución de un gran rango de sistemas P. Un alto grado de flexibilidad implica soportar una gran diversidad de sistemas P (de acuerdo a las variantes designadas a ser simuladas).

- *Escalabilidad*: se refiere a la habilidad de la plataforma para simular sistemas P con aumento de tamaño sin repercutir en la habilidad de realizar sus funciones o reducir el rendimiento. El tamaño de los sistemas P se puede determinar a este respecto mediante el número de membranas, el número de reglas definidas y los multiconjuntos iniciales.

Es difícil asegurar que cualquier plataforma de computación incluya los tres atributos. Un factor que promueve un atributo puede demorar otro. De hecho existen dos conexiones principales entre ellos [75]: *flexibilidad vs rendimiento* y *flexibilidad vs escalabilidad*.

2.4. Simulación paralela de sistemas P

En esta sección se aporta una revisión general de los simuladores paralelos desarrollados hasta el momento para sistemas P.

- *Simuladores basados en cluster* (un conjunto de computadores interconectados mediante una red local [89]):
  - Simulador de *Ciobanu y Guo* [34]: simula un conjunto restrictivo de sistemas P de transición usando un cluster basado en Linux, usando C++ y la biblioteca MPI. Los autores indican que la mayor limitación estaba en la comunicación y cooperación entre membranas.
  - Simulador de *Fernández et al.* [97]: aporta ciertas mejoras al simulador de *Ciobanu y Guo* con el fin de resolver parcialmente el problema del proceso de comunicación.
  - Simulador de *Syropoulos et al.* [96]: este simulador trabaja en Java, y hace uso de la biblioteca *RMI* (Remote Method Invocation) para distribuir el trabajo. Al igual que en los simuladores anteriores, se distribuye cada membrana por cada procesador. Sin embargo, el objetivo era mostrar el uso de los sistemas P como un fundamento de la computación distribuida.
  - Simulador de *Diez Dolinski et al.* [40]: se trata de una alternativa novedosa, que provee una solución altamente escalable al problema del crecimiento exponencial del espacio creado por los sistemas P. Se utilizan algoritmos *MapReduce* sobre los entornos distribuidos, y se amplía P-Lingua (P-Lingua distribuido) para definir las correspondientes entradas.
- *Simuladores basados en FPGA* (chips reconfigurables que el diseñador puede programar, permitiendo usar ciertos recursos como procesadores y memoria "a la carta" [72]):
  - Simulador de *Petreska and Teuscher* [88]: el primer simulador basado en FPGAs que da un total soporte para un subconjunto particular de sistemas P de transición, también permitiendo reglas de división y disolución. Está implementado en *VHDL* (VHSIC hardware description language), y tiene algunas limitaciones, como por ejemplo [75]: no implementa paralelismo dentro de las membranas, es inflexible, y tiene una limitada escalabilidad.

- Simulador de *Fernández et al.* en FPGA [43]: es una alternativa que presenta un diseño de un circuito que puede ser implementado en FPGAs para la selección de las reglas activas de un sistema P de transición.
- Simulador de *Nguyen et al.* [77, 75, 76]: se trata de una evolución en la simulación de sistemas P en FPGAs. Su sistema hardware, denominado Reconfig-P, junto a P-Builder, provee una aproximación elegante al balanceo de rendimiento, flexibilidad y escalabilidad. El sistema construido por encima de la FPGA es capaz de adaptar el diseño del circuito a las características del sistema P. Por ejemplo, el no determinismo se soporta gracias a un algoritmo diseñado para tal fin, denominado DND [76].

- *Simuladores basados en microcontroladores* (circuito integrado que contiene procesador, memoria y unidades de entrada/salida):
  - Simulador de *Gutiérrez et al.* [53, 52]: el objetivo de este simulador es balancear la flexibilidad y el rendimiento a un bajo coste. Se utiliza el *PIC* (Peripheral Interface Controller) como solución, y para tal fin, se han diseñado distintos algoritmos y arquitecturas para mejorar la comunicación entre las unidades que manejan las membranas [97, 18, 52].

- *Simuladores basados en GPUs* (procesadores de las tarjetas gráficas, que pueden ser usados como multiprocesadores especiales altamente paralelos [78], y programados con modelos de programación como CUDA [57, 5] y OpenCL [74, 6]):
  - *PMCGPU*: se trata del proyecto iniciado en nuestro trabajo. Consiste en una serie de subproyectos que tienen como objetivo simular diferentes variantes de sistemas P:
    - *PCUDA*: es el primer simulador para sistemas P basado en CUDA (ver Capítulo 4), conformando una plataforma flexible para simular sistemas P, pero comprometiendo el rendimiento y la escalabilidad.
    - *PCUDASAT*: es una rama del proyecto PCUDA, en la simulación eficiente de una familia de sistemas P con membranas activas que resuelve instancias del problema `SAT` (ver Capítulo 5). Se trata de una plataforma inflexible, pero que incrementa el rendimiento y la escalabilidad.

  - *TSPCUDASAT*: es una variante de PCUDASAT, donde se trata a una familia de sistemas P a modo de tejidos con división celular que resuelven `SAT` (ver Capítulo 5).
  - *ABCD-GPU*: es un proyecto reciente en la simulación de sistemas PDP (ver Capítulo 7). Incluye simuladores basados en C++, y para plataformas de procesadores multinúcleo (con OpenMP [10]) y GPU (con CUDA).

- *Simulación de sistemas P neuronales con impulsos*: los primeros simuladores fueron iniciados por *Cabarle et al.* [19, 20]. El algoritmo implementado usa una representación de matrices del modelo, introducida por *Zeng et al.* [102]. Por tanto, la implementación es relativamente sencilla para la GPU. Además, se aprovecha la simulación del no determinismo como un nivel de paralelismo, de tal manera que cada camino de computación se lleva a cabo por un multiprocesador de la GPU.
- *Simulación de sistemas P de evolución-comunicación con energía y sin reglas antiport*: el trabajo inicial fue llevado por *Juayong et al.* [54]. Como para los sistemas P neuronales de impulsos, el simulador hace uso de una representación de matrices, donde se representa la semántica del modelo.
- *Simulación de sistemas P numéricos y enzimáticos*: este trabajo es llevado a cabo por *García-Quismondo et al.* [47]. Los modelos simulados son usados para modelizar controladores de robots, por lo que los resultados son significativos para el campo de la Inteligencia Artificial.
- *Simuladores de soluciones de procesamiento de imágenes con sistemas P a modo de tejidos*: *Díaz-Pernil et al* [81] presentan un nuevo trabajo en la simulación de modelos de sistemas P a modo de tejidos, pero específicos de algoritmos de procesamiento de imágenes (por ejemplo, suavizado de imágenes). Estas aplicaciones son soluciones específicas para imágenes, donde la simulación de los sistemas P se lleva a cabo implícitamente en el código fuente.

"Durante más de una década, algunos profetas han expresado la opinión de que la organización de los computadores ha llegado a sus límites y que los avances verdaderamente significativos sólo pueden hacerse mediante la interconexión de múltiples equipos."

Gene Amdahl (1967)

3

Computación de Alto Rendimiento

Actualmente es difícil ser conciso a la hora de definir la *Computación de Alto Rendimiento (HPC, de las siglas en inglés)*. Puesto que no existe una única definición, adoptaremos la siguiente: *"la Computación de Alto Rendimiento se refiere generalmente a la práctica de agregar potencia de computación, obteniendo mucho más rendimiento que en una estación de trabajo, con el fin de resolver instancias de problema grandes en ciencia, ingeniería o negocios"* [3].

El término HPC se ha asociado históricamente a la *Computación Paralela*, ya que, a día de hoy, es la forma más adoptada para mejorar el rendimiento de computación. El paralelismo es la base para la aceleración de aplicaciones grandes y complejas del mundo real. A veces, HPC se confunde con el término supercomputación, aunque éste último se refiera al desarrollo de sistemas que aporten la mayor tasa operacional de los computadores actuales (supercomputadores). Se invierten vastas sumas de dinero en implementar supercomputadores, y son usados solo por expertos especializados en problemas especiales. Sin embargo, un sistema de alto rendimiento puede ser usado y administrado sin mucho gasto ni experiencia específica.

Este capítulo introduce brevemente los conceptos necesarios para entender el significado de Computación de Alto Rendimiento y la *Computación Paralela* [79, 49, 1]. También haremos un breve repaso de las tecnologías paralelas actuales, con especial mención a las tarjetas gráficas.

3.1. Computación paralela

3.1.1. Necesidad de paralelismo

Desde la popularización de los computadores como medio para resolver problemas de forma rápida y automática, ha existido siempre una creciente demanda de potencia computacional. Por un lado, los desarrolladores de aplicaciones software han exigido siempre más velocidad en el procesamiento de cálculo con el fin de conseguir más funcionalidad. Y por otro lado, conforme se aumenta la potencia computacional, son más los problemas que se pueden considerar para ser resueltos. Aplicaciones de modelización, simulación y análisis aplicadas a áreas como microbiología, bioquímica, astrofísica, física de partículas, ingeniería, internet y redes sociales, son actualmente las más demandantes de potencia de cálculo. Sin unos computadores lo suficientemente eficientes, estos problemas nunca podrán llegar a ser manejados.

Este tipo de exigencias ha propiciado que año tras año la velocidad de los microprocesadores se haya ido aumentando. Esta mejora de rendimiento en los procesadores ha ido ligada, mayoritariamente, con el nivel de densidad de transistores por chip. Conforme los transistores son más pequeños, la velocidad de estos se incrementa. Sin embargo, esta bajada de escala en los chip también aumenta la cantidad de energía consumida y disipada como calor. Además, la escala de los transistores también tiene un límite conocido por las leyes físicas de lo diminuto, la física cuántica. Aunque aún no se haya llegado a tal límite, el nivel de densidad ha ido disminuyendo. Desde 1986 a 2002, la velocidad de los procesadores aumentaba de media un 50 % por año. Sin embargo, desde 2002 este aumento se ha visto reducido a un 20 % [49].

Este cambio en el incremento del rendimiento de los procesadores está asociado también al cambio del diseño de los mismos. En el año 2005, la mayoría de los fabricantes de microprocesadores cambiaron de estrategia, pasando de producir procesadores monolíticos de baja latencia a sistemas multiprocesador. De esta forma se aprovecha mejor la cantidad de transistores por chip mediante la creación de elementos redundantes de computación. Esta estrategia, gracias a unos diseños de chips muchos más sencillos, consigue crear tecnologías más baratas y con un menor consumo de energía [49].

3.1.2. Computación concurrente y paralela

Normalmente en la literatura podemos encontrar confundidos los términos de concurrencia, paralelismo y distribuido. Conviene realizar una distinción en esta memoria [79]. Consideramos *Computación Concurrente* al desarrollo de

programas donde múltiples tareas se ejecutan a la vez. Por ejemplo, un sistema operativo se puede considerar un programa de Computación Concurrente, ya que puede llevar múltiples tareas a la vez, incluso usando un solo procesador.

Por otro lado encontramos la *Computación Paralela*, el cual hace mención al estudio de resolución de problemas mediante el uso de múltiples procesadores que trabajan en conjunto. Cabe mencionar que el control de los procesadores se realiza de manera explícita mediante *programación paralela*. Decimos que un programa es *secuencial* si la ejecución de sus instrucciones se realiza de forma sucesiva. Un programa secuencial no se ejecutará más rápido en más procesadores, sino que tendrá que ser reescrito para poder aprovechar el paralelismo. Este proceso de conversión se denomina *paralelización*.

3.1.3. Estimación de rendimiento

En este apartado introduciremos brevemente conceptos importantes para entender, de manera formal, el objetivo principal de la Computación Paralela: mejorar el rendimiento de la ejecución de programas.

Primero, necesitamos conocer qué significa que un programa tenga un buen rendimiento. Existen distintas métricas para analizar un algoritmo, según los recursos de los que se haga uso e interese medir. Así pues, hablar sobre rendimiento dependerá de la métrica o métricas adoptadas.

Las métricas más importantes a la hora de analizar un algoritmo son las correspondientes a la velocidad (o tiempo de ejecución) y espacio de memoria, ya que estas dos se basan en el uso de recursos bastante limitados. Este análisis se basa en la estimación de consumo de los recursos correspondientes. De esta manera se nos permite comparar el coste relativo de distintos algoritmos para resolver el mismo problema. Para ello se suele hacer uso del concepto de *razón de crecimiento*, el cual hace mención al incremento del costo de un algoritmo conforme crece el tamaño de entrada.

Una manera usual de estimar el coste de tiempo de un algoritmo es a través del número de operaciones básicas requeridas. Como la cantidad real y exacta de operaciones básicas de un algoritmo es difícil de calcular, se suele llevar a cabo una estimación del mejor y peor caso mediante el análisis asintótico del algoritmo. Este análisis se refiere al estudio de un algoritmo conforme el tamaño de entrada se vuelve grande, o alcanza un límite. La cota superior de un algoritmo indica la máxima razón de crecimiento. Se indica mediante una notación especial llamada O grande (*big O notation*).

El objetivo en este trabajo es comparar un programa secuencial (tomando T_{sec} unidades de tiempo) y un programa paralelo (tomando T_{par} unidades de

tiempo). El valor T_{sec} debería de ser el tiempo de ejecución del programa más rápido ejecutándose en uno del mismo procesador paralelo. Asumiremos que la *aceleración* (o *speedup*) conseguida por un programa paralelo viene dado por

$$S = \frac{T_{sec}}{T_{par}}$$

Como el número resultante no tiene unidades, añadiremos una "x" al final. Obsérvese que un speedup lineal viene dado cuando $S = p$, siendo p el número de procesadores. Denotaremos también por eficiencia de un programa paralelo la fracción $\frac{S}{p}$.

Sin embargo, no todo el código de un programa es paralelizable, pero cuanto más grande sea la parte paralelizable, mejor será el rendimiento conseguido. Esta observación fue hecha por Gene Amdahl en 1960 [14], mediante la *ley de Amdahl*: asúmase que A es la parte paralelizable, y B el resto, entonces el speedup de un programa paralelo viene dado por

$$S = \frac{T_{sec}}{A \cdot \frac{T_{sec}}{p} \ + \ B \cdot T_{sec}}$$

La afirmación general que podemos obtener de la ley de Amdahl es que si la fracción B de nuestro programa se mantiene no paralelizable, entonces nunca podremos obtener una aceleración mayor que $\frac{1}{B}$. Sin embargo, esta restricción se puede superar si se toma en cuenta el tamaño del problema, ya que, normalmente, la parte de tiempo paralelizable se incrementa conforme se aumenta el tamaño de la instancia del problema. Esto se postuló en la *ley de Gustafson* [51], como $S(p) = p - \alpha(p - 1)$, siendo p el número de procesadores, y α la parte no paralelizable.

3.2. Plataformas paralelas

3.2.1. Tipos de plataformas paralelas

A fin de ejecutar un programa en paralelo de forma efectiva, debemos de utilizar *plataformas paralelas*, las cuales contienen más de un procesador interconectado. Se denomina *arquitectura* paralela a la estructura interna de estas plataformas, pero usaremos a menudo los términos arquitectura y plataforma como similares.

Las arquitecturas paralelas pueden ser clasificadas, según su estructura de control (ejecución de tareas paralelas) mediante la *taxonomía de Flynn*. Esta

clasificación, usada desde 1996 [79, 1], se realiza acorde al número de flujos de instrucciones y el número de flujos de datos manejados simultáneamente:

- *SISD (Single Instruction, Single Data)*: solo un flujo de instrucción por flujo de datos.
- *SIMD (Single Instruction, Multiple Data)*: todos los procesadores comparten el mismo flujo de instrucciones. Sin embargo, cada procesador tiene su propio flujo de datos. Este tipo de arquitectura explota el paralelismo de datos, requiriendo menos hardware que los MIMD.
- *MISD (Multiple Instruction, Single Data)*: cada procesador tiene su flujo de instrucciones, pero comparten el mismo de datos. No es una arquitectura muy común, aunque lo usaba el computador experimental Carnegie-Mellon C.mmp (1971).
- *MIMD (Multiple Instruction, Multiple Data)*: cada procesador tiene su flujo de instrucciones y de datos propio. Este es el típico caso de plataforma paralela. Una variante de este modelo se denomina *SPMD (Single Program, Multiple Data)*, y se basa en ejecutar instancias del mismo programa sobre diferentes datos, teniendo además la misma expresividad que el modelo MIMD[1].

De manera alternativa, las plataformas paralelas MIMD también se pueden clasificar según el modelo de comunicación entre tareas, como sigue:

- Plataformas de *memoria distribuida*: cada procesador tiene su propia memoria privada, y la comunicación se realiza mediante la red de interconexión de forma explícita (con paso de mensajes usando la biblioteca *MPI* [4], o funciones remotas con RMI). El rendimiento de estas plataformas también recae en la topología de la red (anillo, toroidal, hipercubo, mariposa, etc.).
- Plataformas de *memoria compartida*: los procesadores se conectan a un sistema de memoria global mediante una red de interconexión denominada *bus*. La interacción de las tareas se realiza directamente mediante esa memoria compartida. Las bibliotecas estándar más usadas son POSIX threads (*Pthreads*) y *OpenMP* [10]. Existen dos implementaciones diferentes para esa memoria:

[1]MIMD puede ser traducido a SPMD usando construcciones if-else

- *UMA (Uniform Memory Access)*: el tiempo de acceso de cada dato es el mismo para todos los procesadores, ya que todos están conectados con la misma memoria.
- *NUMA (Nonuniform Memory Access)*: el tiempo de acceso de cada dato se diferencia por cada procesador. La razón principal es porque por debajo existe un sistema de memoria distribuida, que es transparente para los procesadores (requiriendo protocolos de coherencia), obteniendo una mayor escalabilidad comparado con sistemas UMA. Un uso típico de los sistemas NUMA son las arquitecturas *SMP (symmetric multiprocessing)*, usada en la mayoría de los supercomputadores de ahora.

3.3. Elementos de una arquitectura

Las arquitecturas paralelas están basadas en el modelo de *Von Neumann*. Este se compone de una *memoria principal* (donde se almacenan datos e instrucciones), una *Unidad Central de Proceso, o CPU* (que trabaja de acuerdo a los clic de un *reloj*), y una red de *interconexión* entre la CPU, la memoria y periféricos de entrada/salida [79]. El *sistema operativo* es el software que toma el control de estos elementos hardware, así como de la ejecución del software.

Diremos que un *programa* es un código (datos e instrucciones) almacenado en disco duro, creado a partir de un *código fuente* mediante un *compilador*. Cuando se ejecuta, el sistema operativo crea un *proceso*, que instancia un programa en memoria principal. Un proceso contiene tanto el código del programa, así como un bloque de memoria, que se inicializa de dos formas distintas: memoria *estática* (reservado en tiempo de compilación) o memoria *dinámica* (reservado en tiempo de ejecución). A veces, cuando se usa memoria dinámica justo al comienzo y final del proceso, se suele también denominar memoria estática.

La mayoría de los sistemas operativos modernos son multiusuario (varios usuarios pueden acceder a la vez) y multitarea (pueden ejecutar varios procesos concurrentemente). Como hemos mencionado, un programa paralelo coopera con el fin de resolver un problema. Esto se puede conseguir con la multitarea, comunicando los procesos, o con bibliotecas especiales para ejecutar procesos de manera coordinada en distintas máquinas (por ejemplo, con paso de mensajes MPI). Sin embargo, el método más común de paralelismo es mediante *hilos* (o threads). Un hilo es una tarea independiente dentro de un proceso, que se ejecuta concurrentemente y puede cooperar con otros hilos rápidamen-

te mediante memoria compartida. Algunas bibliotecas estándar para hilos son Pthreads y OpenMP [10].

Otro punto crítico en una plataforma paralela es el sistema de memoria, el cual ha de ser lo suficientemente rápida para alimentar con datos e instrucciones los procesadores que lo componga. Hoy en día, el sistema de memoria se implementa mediante una jerarquía de subsistemas, la cual viene formada habitualmente por, de abajo (más grande y lento) hacia arriba (más pequeño y rápido): *memoria secundaria* (donde se almacenan los programas, implementado por discos duros), *memoria principal* (accesible desde CPU, donde se almacenan los procesos, e implementado por RAM o memoria de acceso aleatorio), *memoria caché* (donde se almacenan los últimos datos accedidos, junto a sus cercanos, lo cual evita accesos a memoria principal con esta técnica, denominada *caching*, si se hace de forma automática se denomina *caché*, y si se hace de forma manual se denomina *scratchpad*) y *registros* (dentro de la CPU, donde se almacenan unidades de datos, o variables). De forma adicional, el sistema operativo (y opcionalmente el hardware) implementa una técnica denominada *memoria virtual*, a nivel de memoria principal, con el fin de poder ejecutar procesos que sean más grandes que la memoria RAM disponible, ya que las partes menos frecuentadas por un proceso se mueven a disco duro de forma automática y transparente.

3.3.1. Plataformas paralelas actuales

Actualmente existen muchas plataformas paralelas, cada una distinta según su arquitectura. A continuación pasaremos a introducir de manera esquemática las más importantes. Las siguientes plataformas paralelas están ordenadas de mayor a menor escalabilidad.

- *Redes de ordenadores*: se trata de la interconexión de procesadores mediante una red local o global. Existen dos arquitecturas principales: servidor-cliente (una máquina tiene el rol de administrar toda la red, lo cual puede causar un cuello de botella) y peer-to-peer (todas las máquinas son clientes, aunque esto puede hacer más difícil la sincronización). Existen muchas soluciones basadas en redes hoy en día:

  - *Computación Grid*: es una forma de Computación Distribuida donde se compone un supercomputador virtual mediante varias redes de computadores [45]. Esto incluye desde una aglomeración de redes empresariales o académicas, o a un gran grupo de voluntarios que

prestan sus recursos computacionales cuando no están en uso (como por ejemplo, el proyecto BOINC [7]).

- *Computación en Nube*: una alternativa a la Computación Grid, donde los recursos computacionales se "alquilan". Así, se ofrecen recursos de computación, almacenamiento y software "en la nube" como un servicio [99].
- *Clusters*: se trata, a nivel de redes locales, de una aglomeración de procesadores en configuración paralela [89]. El control de recursos, planificación de trabajos y usuarios se realiza a través de un nodo central. Los supercomputadores actuales se tratan de clusters de alta gama con una cantidad ingente de recursos.

- *Procesadores multinúcleo*: la mayoría de los procesadores actuales están implementados, en realidad, por varios núcleos de procesamiento interconectados mediante memoria caché. Estos núcleos se pueden explotar mediante el uso de hilos con OpenMP, como veremos en los simuladores desarrollados en este trabajo.

- *Aceleradores*: se trata de una nueva tendencia en HPC en la que se trata de mejorar el rendimiento de los nodos de computación. Son chips baratos pero masivamente paralelos que realizan funciones específicas de forma más rápida que las CPUs convencionales. Tienden a ofrecer cientos de procesadores especiales, y son programadas de manera específica. Los aceleradores son un claro ejemplo de Computación Heterogénea: la CPU es el dispositivo maestro que se ayuda del acelerador para conseguir un mayor rendimiento. Algunos claros ejemplos son:

  - *FPGAs (Field-Programmable Gate Array Chips):* son circuitos digitales integrados (ICs) que contiene una interconexión configurable entre sus bloques [72]. Muchos aceleradores se construyen con esta tecnología, ya que se puede programar para implementar procesadores específicos.
  - *Cell-BE (Cell Broadband Engine)*: desarrollado conjuntamente por Sony, Toshiba e IBM, es un procesador con arquitectura SIMD y heterogéneo, ya que contiene un procesador "cerebro" (Power Processor Element) y de 8 a 9 "músculos" (Synergistic Processing Elements) [58]. Actualmente se encuentra en reproductores Bluray y la videoconsola Playstation 3.

- *GPUs (Graphics Processing Units)*: término popularizado por NVIDIA en 1999, y también conocido como VPU (Visual Processing Unit), es el procesador incluido en las tarjetas gráficas actuales. Es un chip especializado para manipular de forma rápida gráficos en 3 dimensiones. Su estructura altamente paralela los convierte en unos procesadores que explotan el paralelismo de datos a través de su arquitectura SIMD [78, 57]. En las siguientes secciones pasaremos a detallar más este tipo de acelerador, y su importancia actualmente en la HPC.

3.4. Computación GPU

Con la demanda del sector comercial de vídeo y juegos, estaba previsto por Elster y otros [41] autores que el desarrollo de los procesadores gráficos darían lugar a dispositivos adecuados para Computación de Alto Rendimiento. La era de la GPGPU (General-Purpose computing on Graphics Processing Units) comenzó verdaderamente con la introducción de CUDA (de NVIDIA) [57, 42] y los entornos Stream SDK (de AMD) en 2007, por lo que las GPUs fueron programables más fácilmente. Actualmente, las GPUs son consideradas como procesadores masivamente paralelos para soluciones en la aceleración de aplicaciones demandantes de cómputo (*Computación GPU* [8]).

En esta sección introduciremos la evolución de la GPU, y detallaremos el entorno utilizado a lo largo del trabajo.

3.4.1. Evolución de la arquitectura de la GPU

Las primeras tarjetas de vídeo, como componente de un PC común, se dedicaban tan solo a dibujar los *píxeles*[2] en pantalla según los datos recibidos desde la CPU. Con el fin de liberar la CPU de realizar cálculos gráficos, se creó la *GPU (Graphics Processing Unit)* [38] y se ubicó dentro de la tarjeta de vídeo. La GPU es un procesador de propósito específico, optimizada para trabajar con gráficos. Precisamente, la GPU está basada en el tipo de arquitectura *streaming*.

Los cálculos gráficos más complejos son los basados en tres dimensiones de tiempo real (por ejemplo, como el generado por los videojuegos en 3D). Por ello, el pipeline gráfico de la GPU está basado en streaming de triángulos y texturas, con el objetivo de transformar los datos en 3D (coordenadas de los

[2]Un punto con un solo color en una imagen de tramas (bitmap).

triángulos del modelo, fáciles de entender por los programadores), a píxeles a dibujar en una pantalla 2D.

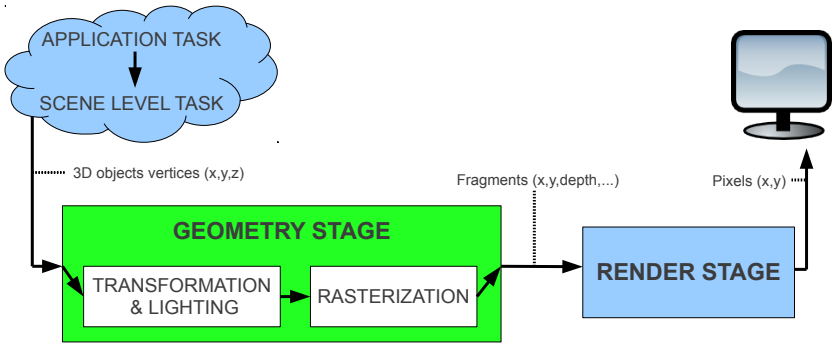


Figura 3.1: Proceso general de gráficos, desde su creación hasta la salida por pantalla.

La Figura 3.1 muestra la arquitectura general del pipeline gráfico 3D. La entrada a la GPU son las coordenadas de los objetos 3D (principalmente formado por triángulos), que entran en la etapa de *Geometría*, donde las coordenadas 3D se transforman a coordenadas 2D (fragmentos). En la última etapa, *Rendering*, el color final de cada píxel se calcula en base al color asociado, luminiscencia, textura, efectos alfa (transparencia) y de niebla de cada fragmento de la salida de la etapa de Geometría. Como se puede apreciar en la figura, la etapa de Geometría está divida a su vez en otras dos subetapas. La primera se denomina *Transformación y Luces*, que es el proceso de proyectar los objetos en tres dimensiones a dos dimensiones, y calcular a la vez los efectos de luces en la escena. La segunda etapa, *Rasterization* (o *Triangle Setup* en otros trabajos), configura y corta los triángulos, haciendo fragmentos para la etapa de Rendering. Las tareas a nivel de aplicación (IA, cámara, interacción, etc.) y tareas de nivel de escena (colisiones, física, etc.) son siempre llevadas a cabo por la CPU ya que son tareas de propósito general, y no necesariamente de gráficos.

Sin embargo, no todas la etapas comentadas anteriormente fueron implementadas en la primera tarjeta gráfica. Se fueron incorporando a la GPU de manera sucesiva, liberando trabajo a la CPU. Pero para este propósito fue necesario incorporar paralelismo, de tal forma que la GPU comenzó a ser un procesador SIMD específico para gráficos. Al final del proceso, y llegando a la GPU con todas las etapas (Geometría y Rendering) implementadas en chip, nació la arquitectura denominada *pipeline de función fija*, donde el compor-

tamiento de la GPU era estática, realizando siempre las mismas operaciones sobre vértices y fragmentos [78]. El pipeline, por tanto, no era flexible para los programadores.

En 2002, la solución fue la de desarrollar un *pipeline programable*, donde la GPU pudiera ejecutar pequeños programas llamados *shaders* sobre vértices y fragmentos. Existen dos tipos: *vertex shader* (que reemplaza las transformaciones y luces) y *fragment shader* (reemplazando las texturas, suma de colores y efectos de niebla). Inicialmente, los códigos shader se programaban en lenguajes de bajo nivel, pero al poco tiempo surgieron lenguajes de alto nivel que hacían fácil la tarea de programar las unidades de shader, como por ejemplo *Cg* y *GLSlang*.

El problema del pipeline programable es que las unidades de vertex shader y fragment shader son usadas en distintas proporciones, de tal forma que hay momentos en los que muchas unidades de vertex están libres mientras que no hay suficientes de fragment, y viceversa. Para solventar este problema se introdujo la *unidad de shader unificada*. Estas unidades son capaces de ejecutar códigos de vertex y fragment, introduciendo además un lenguaje único de shader [73]. AMD introdujo su primera unidades unificadas de shader en el Xenos GPU para la Xbox 360.Esta nueva unidad es la clave del rápido desarrollo de la GPGPU: los programadores pueden enfocarse en programar solo esta unidad [78].

3.4.2. Programación con GPGPU

La evolución de la unidad unificada de shader ha dado lugar a una nueva potencia de computación basada en gráficos con una naturaleza altamente paralela: una GPU contiene de 16 a miles de unidades de shader. Por tanto, la GPU se ha considerado también para ejecutar aplicaciones no necesariamente gráficas, que cumplan las siguientes características [78]: grandes requerimientos computacionales, paralelismo substancial, y que el rendimiento sea más importante que la latencia. Dado el éxito alcanzado acelerando aplicaciones en la GPU, Mark Harris, ahora investigador en NVIDIA, acuñó el término *GPGPU (General-Purpose computing on the GPU)* [8] en 2002. Hoy en día este esfuerzo, también conocido como *Computación GPU*, ha posicionado a la GPU como uno de los aceleradores más potentes y baratos en HPC.

La GPU, como un acelerador, tiene su propio modelo de programación, basado en SPMD: se procesan muchos elementos independientes en paralelo usando el mismo programa [78]. Esto se consigue gracias a que se dedica más hardware para computación que para control, por lo que una ramificación

incoherente (hilos que toman caminos de código distintos) conlleva una penalización. La solución adoptada fue la de trabajar con grupos de elementos o bloques, siendo tales bloques procesados en paralelo si toman el mismo ramal (los bloques son tratados de forma SIMD). El tamaño de bloque se denomina *granularidad de ramificación*, y se ha ido reduciendo con la evolución de la GPU. Este concepto se usará después en CUDA como *warps*.

La GPGPU ha evolucionado junto a la programabilidad de la GPU [78]. Inicialmente, cuando se programaba la GPU para gráficos, los desarrolladores tenían que pensar en la geometría de cada región, fragmentos con shader y buffers de gráficos. Por tanto, un programador GPGPU tenía que adaptar su problema a datos gráficos, y aplicar sus operaciones traduciéndolas a shaders. Esto se conoce como GPGPU *antigua*. La dificultad de escribir aplicaciones GPGPU fue resuelta introduciendo una interfaz más natural, directa y no gráfica al hardware de la GPU. Esto fue llevado a cabo mediante la abstracción de la GPU como un procesador de streaming [78]. *BrookGPU* y *Sh* fueron los proyectos académicos que introdujeron esta idea. Con esto comenzó la *nueva* era de la GPGPU, donde los programadores definen directamente programas SPMD de propósito general usando hilos. Los nuevos modelos de programación, como CUDA [57] y OpenCL[74, 6], han permitido la rápida evolución de la GPGPU, y son los entornos más utilizados hoy en día a este respecto.

3.4.3. El modelo de programación CUDA

El modelo de programación CUDA está basado en la Computación Heterogénea: el sistema consiste en un anfitrión, o *host*, que es una CPU tradicional, y uno o más dispositivos, o *devices*, que son procesadores altamente paralelos como la GPU.

En muchos programas modernos existen secciones que exhiben una rica cantidad de paralelismo de datos. Los dispositivos CUDA aprovechan esta propiedad para acelerar su ejecución. Un programa CUDA es un código fuente unificado que cubre ambos lados: el host es tratado con código ANSI C/C++ [56], y el device con un lenguaje C extendido con palabras claves especiales. El código device está compuesto por funciones que se pueden llamar desde el host, y se ejecutarán de manera paralela en el dispositivo. Éstas se denominan *kernels*. Los hilos de CUDA ejecutan siempre el mismo kernel, pero pueden operar sobre distintos datos gracias a que ciertas variables que tienen valores distintos para cada hilo (SPMD) pueden ser usadas como índices. Estos hilos son más ligeros que los de la CPU, por lo que el programador CUDA puede asumir que se requiere pocos ciclos de reloj para generarlos y planificarlos,

además de lanzar muchos de ellos para conseguir un buen rendimiento.

En la Figura 3.2 se ilustra la ejecución de un programa CUDA. La ejecución siempre comienza en el host, y cuando un kernel se invoca, se pasa al device, donde se genera un gran número de hilos para aprovechar el abundante pralelismo de datos. Todos los hilos de un kernel se denominan colectivamente *grid*. Cuando todos los hilos completan su ejecución, el correspondiente grid acaba, y por tanto el kernel, devolviendo la ejecución a la CPU.

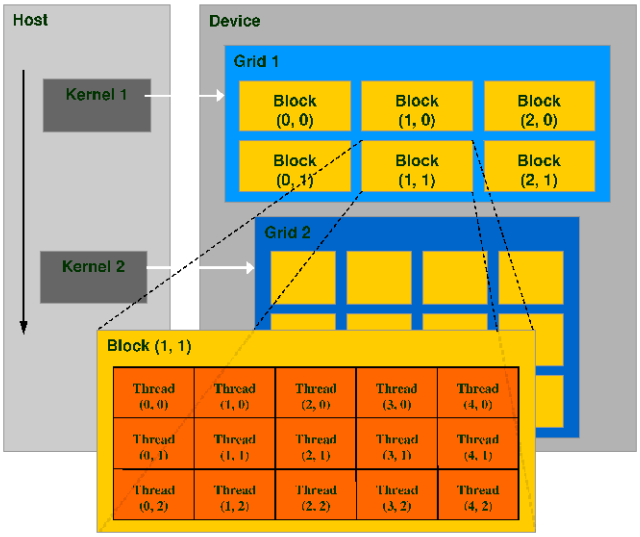


Figura 3.2: El modelo de ejecución de hilos de CUDA

Un grid está compuesto normalmente por miles de hilos para poder usar todos los recursos hardware de la GPU. Los hilos están organizados en una jerarquía de dos niveles dentro del grid, como se ilustra en la Figura 3.2. En el nivel superior, cada grid consiste en uno o más *bloques de hilos*. En el nivel inferior, cada bloque se organiza en un array tridimensional de hilos. Todos los bloques en el grid tienen el mismo número y organización de hilos. Cada bloque se identifica por un identificador bidimensional, y cada hilo (dentro de su bloque) por un identificador tridimensional. Además, un bloque solo puede contener, como mucho, 512 hilos.

CUDA permite que los hilos dentro de un bloque coordinen sus actividades usando una función de sincronización de barrera. Cuando un hilo ejecuta esta función, se bloquea hasta que el resto de hilos del bloque la ejecuten. Los hilos de diferentes bloques solo se pueden sincronizar acabando el kernel.

El modelo de memoria es también un aspecto a considerar en CUDA, ya que la jerarquía de memoria se administra de forma explícita y manual. En CUDA, el host y los dispositivos tienen espacios de memoria separados. Esto refleja el hecho que los dispositivos son GPUs que contienen su propia memoria DRAM (Dynamic Random Access Memory). Para poder ejecutar de forma efectiva un kernel en el dispositivo, primero se necesita reservar memoria en él y transferir los datos relevantes desde el host. De forma similar, después de la ejecución del kernel, es necesario transferir los datos de vuelta para conocer el resultado. Desde este punto de vista, podemos asumir que CUDA usa un modelo de memoria estática, aunque en las nuevas GPUs de NVIDIA se permite memoria dinámica pero con limitaciones.

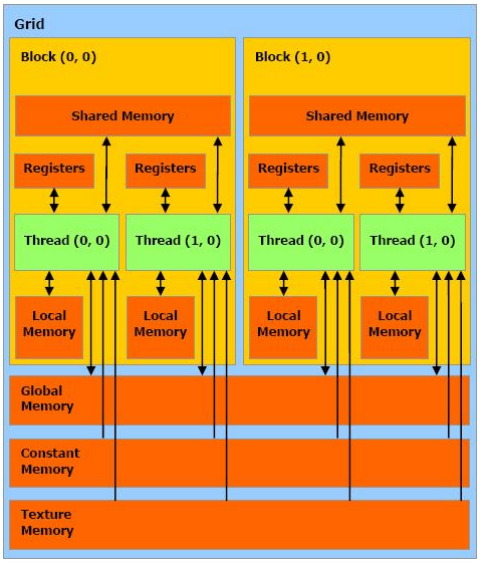


Figura 3.3: Modelo de memoria de CUDA

La Figura 3.3 muestra un resumen del modelo de memoria de CUDA. En la parte más baja de la jerarquía encontramos las memorias *global* y *constante*. Estas memorias son accesibles desde el host de forma bidireccional. La memoria constante es de solo lectura pero usa una caché intermedia (*cacheable*), y la memoria global no es cacheable pero es de lectura y escritura, además de ser la de mayor tamaño. Por tanto, la memoria global es la usada comúnmente para indicar los datos de entrada, y recoger los datos resultantes.

En la parte alta de la jerarquía, CUDA provee de una pequeña porción de memoria denominada *compartida* para cada bloque de hilos. El acceso a esta memoria es comparable al de los *registros*. Sin embargo, esta memoria

no es accesible desde el host. Son los hilos quienes han de usarlo para ocultar accesos a memoria global. Puesto que se usa de forma explícita, esta memoria es scratchpad. Finalmente, se puede reservar una pequeña porción de memoria privada para cada hilo (que se almacenará en memoria global), denominada memoria *local*.

Como se puede observar en la figura, los hilos dentro de un bloque pueden cooperar rápidamente a través de la memoria compartida. Pero los hilos de diferentes bloques solo pueden hacerlo a través de la memoria global, junto con operaciones especiales denominadas *atómicas* (funciones especiales para unas ciertas operaciones como suma y resta, que se realizan de forma sincronizada).

Por último, cabe destacar dos estrategias para poder utilizar de forma eficiente la jerarquía de memoria. Por un lado, se considera una buena estructura algorítmica a lo siguiente: primero los hilos de cada bloque leen sus porciones de datos desde memoria global a compartida, después éstos trabajan directamente en memoria compartida, para finalmente, copiar los resultados de vuelta a memoria global. Una estrategia bien conocida en programación paralela, y usada en CUDA es el *tiling*. En ésta, los tres pasos comentados con una partición de los datos se combinan, de tal forma que se repiten los tres pasos por cada porción (o *tile*), minimizando los accesos a memoria global.

3.4.4. Arquitecturas de GPU modernas

En esta sección se introduce la arquitectura de la tarjeta gráfica utilizada en los experimentos realizados en este trabajo: la NVIDIA Tesla C1060 [60] (cuya arquitectura se denomina G200).

La Tesla C1060 está basada en un vector de procesadores escalables que contiene 240 núcleos *SPs* (Streaming Processor), organizados en 30 multiprocesadores *SMs* (Streaming Multiprocessor). Contiene además 4 GBytes de memoria *device* (o global) en GDDR3 (memoria DRAM optimizada para gráficos). La comunicación con la CPU y la memoria principal se realiza a través de un bus *PCI Express* x16 (ver Figura 3.4), que ofrece un ancho de banda de 4 GB/sec por dirección.

Cada SM contiene las siguientes unidades: ocho núcleos aritméticos SPs, una unidad de doble precisión, una caché de instrucciones, una caché constante de solo lectura, 16 KBytes de memoria compartida, un conjunto de 16384 registros de 32 bits, dos unidades de operaciones complejas de punto flotante (SFU) y acceso a la memoria fuera del chip (device o global). Los SPs son capaces de ejecutar tres instrucciones por ciclo de reloj, funcionando a 1,296 GHz.

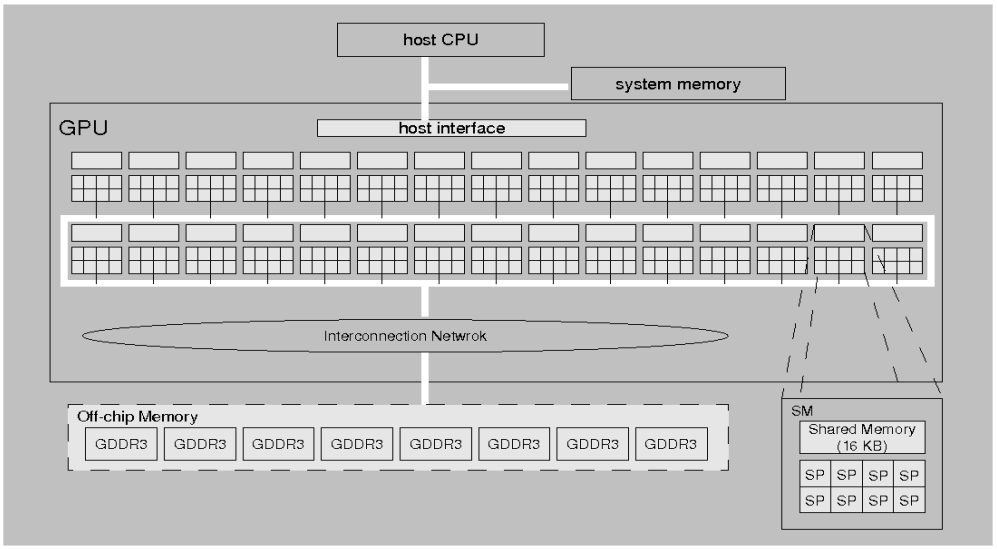


Figura 3.4: Arquitectura unificada de Tesla T10, basada en G200.

Las memorias local y global (device) no están cacheadas, por lo que cada acceso de memoria genera un acceso explícito. Un multiprocesador toma 4 ciclos de reloj para lanzar una instrucción de memoria (los necesarios para registros y memoria compartida), mientras que el acceso a memoria global siempre incurre unos 400 a 600 ciclos de reloj adicionales [5]. Además, el máximo ancho de banda accediendo a memoria global se consigue con accesos concurrentes a posiciones contiguas. Esto es una parte crítica del proceso de adaptación de aplicaciones a la GPU.

Un SM es un dispositivo hardware específicamente diseñado para capacidades multi-hilo. Cada SM maneja y ejecuta hasta 1024 hilos en hardware sin sobrecarga de planificación. Cada hilo tiene su propio estado de ejecución, y puede ejecutar un camino de código independiente. Sin embargo, los SMs ejecutan hilos de forma que todos los hilos aplican la misma instrucción en diferentes datos (*SIMT, Single-Instruction Multiple-Thread*) [60]. Los hilos se crean, manejan, planifican y ejecutan en grupos de 32 hilos (que es la granularidad de ramificación de las GPUs de NVIDIA). Estos grupos se denominan *warp*, y se lanzan uno cada vez a los núcleos SP de los SMs. Si algún hilo ejecuta instrucciones distintas que otros del mismo warp, entonces la unidad de planificación se romperá, por lo que los hilos del warp se ejecutarán de forma secuencial. Esto también forma parte del proceso de adaptación de aplicaciones

a la GPU.

3.4.5. Consideraciones de rendimiento

En esta sección resumiremos brevemente algunas consideraciones de rendimiento a tener en cuenta cuando se programa GPUs. Es importante entender que el código implementado para la GPU debe de ser previamente premeditado. Si se realiza de forma abrupta, la aplicación no será acelerada adecuadamente.

Existen algunas operaciones de paralelismo de datos que son adecuadas para la Computación GPU. Así, el éxito de acelerar muchas aplicaciones recae en ser capaces de traducir partes del algoritmo a alguna de estas operaciones. Destacamos cuatro operaciones, que se han convertido en primitivas computacionales de la GPU [78]:

- *Scatter/gather*: escribir a o leer de memoria. Históricamente, la GPU permitía realizar lectura eficiente (gather mediante memoria caché), pero la escritura (scatter) de forma más lenta (más aún cuando implican accesos atómicos). Aveces es mejor dejar que los hilos escriban cada valor de salida mediante la lectura de varios elementos de entrada, en vez asignar un elemento de entrada a cada hilo y que escriban en varias posiciones arbitrarias de memoria (lo que requiere a su vez sincronización).

- *Map*: aplicar una operación a todos los elementos de una colección. Esto es típicamente expresado como un bucle *for* en código secuencial, y que puede se realizado fácilmente en paralelo. Es, en realidad, un claro ejemplo de paralelismo de datos.

- *Reduce*: aplicar repetidamente una operación binaria asociativa para reducir una colección de elementos a un solo valor (por ejemplo, suma, media aritmética, mínimo, etc.). Se puede aplicar en paralelo de forma iterativa, donde los hilos reducen los valores de los elementos. Por tanto, se requiere de una paso de sincronización en cada iteración.

- *Scan* (también conocido como parallel-prefix-sum): toma un vector A y retorna un vector B de la misma longitud, de tal forma que cada elemento $B[i]$ representa una reducción del sub-vector $A[1 \dots i]$. Esto se usa principalmente en algoritmos de ordenación como quicksort, y operaciones con matrices dispersas.

Finalmente, una caracterización de lo que las GPUs hacen bien nos permitiría definir algoritmos más eficientes [78]. A continuación proporcionamos

cuatro caracterizaciones a considerar en nuestros algoritmos, que son de gran importancia en Computación GPU. Es, de hecho, la mejor forma de entender como trabaja la GPU:

- *Enfatizar paralelismo*: las GPUs prefieren ejecutar miles de hilos ligeros para maximizar las oportunidades de ocultar las latencias de accesos a memoria. Los algoritmos deberían de permitir dividir la computación en piezas independientes. Para este propósito, se hace necesario reducir los recursos asignados a cada hilo, e intentar evitar sincronización.

- *Minimizar divergencia de ramificación*: en CUDA, el warp es la unidad de paralelismo. Un warp es ejecutado en paralelo, pero los warps que pertenecen al mismo bloque de hilos se ejecutan secuencialmente. Si un warp se rompe porque hay divergencia, entonces no hay paralelismo real (solo entre los bloques de hilos pues se distribuyen ente los SMs). Cabe destacar que una granularidad de ramificación alta incrementa la posibilidad de romperse, pero si es baja dará lugar a un bajo grado de paralelismo real.

- *Maximizar la intensidad aritmética*: la computación es relativamente barata en GPUs, pero el ancho de banda es preciado. Es mejor maximizar las operaciones computacionales por transacción de memoria, utilizando memoria compartida y registros para tal fin.

- *Explotar el ancho de banda de streaming*: sin embargo, las GPUs y su memoria (GDDR3) ofrecen un ancho de banda (pico) 10 veces superior al de CPUs y DRAM tradicionales. Esto se obtiene mediante patrones de acceso a memoria de streaming: accesos concurrentes a posiciones de memoria alineadas. Una buena forma de maximizar el ancho de banda en un algoritmo es mediante la estrategia scatter/gather.

Parte II

Simulación Paralela Aplicada a Soluciones Eficientes de Problemas Computacionalmente Duros

"La llegada de los aceleradores en la Computación de Alto Rendimiento ofrece novedosas vías para desarrollar simuladores nuevos y eficientes para sistemas P y Biología de Sistemas."

J.M. Cecilia, G.D. Guerrero, J.M. García, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez

4

Simulación de Sistemas P con Membranas Activas en la GPU

Con el fin de validar experimentalmente modelos basados en sistemas P, se hace necesario tener simuladores que se ejecuten en ordenadores electrónicos [39]. Además, deben de ser lo más eficiente posible para poder manejar instancias de gran tamaño, lo cual es uno de los principales retos de los simuladores de sistemas P actuales [39].

Este modelo de computación paralela nos lleva a buscar tecnologías computacionales altamente paralelas donde el simulador paralelo puede ejecutarse de forma eficiente. La nueva generación de GPUs, como se mencionó en el capítulo anterior, son considerados procesadores masivamente paralelos que pueden soportar la ejecución de miles de hilos de forma concurrente. Muchas aplicaciones de propósito general (no gráfico) han sido portadas a estas plataformas hasta la fecha, obteniendo buenas aceleraciones comparadas con sus correspondientes versiones secuenciales [94, 95].

En este capítulo, remarcamos la necesidad de utilizar arquitecturas paralelas que mejoren la eficiencia de los simuladores de sistemas P. Para este propósito, presentaremos un simulador paralelo para la clase de sistemas P reconocedores con membranas activas empleando CUDA [27, 25, 30, 70, 67, 50, 31, 71], dado el hecho de que en el modelo teórico, la creación de un espacio de trabajo exponencial expresado en número de membranas y objetos se produce de mo-

do natural en un tiempo lineal. También se provee un test de rendimiento del simulador con una familia de sistemas P que explotan el paralelismo intrínseco de los sistemas P, y demuestran que la GPU está mejor posicionada que las CPUs para simular sistemas P conforme se incrementa el tamaño de instancia de problema.

4.1. Algoritmo de simulación

El simulador que se ha desarrollado está basado en el simulador secuencial para sistemas P incluido en PLinguaCore [46]. En este diseño, el proceso de simulación está dividido en dos etapas: *selección* y *ejecución*. La etapa de selección consiste en la búsqueda de reglas a ser aplicadas en cada membrana para una configuración dada. Las reglas seleccionadas son ejecutadas (realizando la transición del sistema simulado) en la etapa de ejecución.

El proceso de simulación consta de un bucle principal, donde las etapas de selección y ejecución son ejecutadas. El bucle termina cuando se cumple una de las dos condiciones siguientes: que el sistema simulado alcance una configuración de parada, o que se alcance un número límite de iteraciones definidos al comienzo de la simulación. La entrada de datos de la etapa de selección consta de una descripción de membranas y su correspondiente estructura en árbol enraizado, multiconjuntos de objetos asociados a cada membrana, y el conjunto de reglas definido en el sistema. Los datos de salida de esta etapa consisten en una serie de multiconjuntos de reglas asociados a cada membrana (la membrana activa de la regla seleccionada), indicando la selección realizada.

Cabe hacer hincapié en que la etapa de selección consiste en la búsqueda de reglas a ser ejecutadas en cada membrana. Es en esta etapa donde el no determinismo se presenta cuando existe más de una posible selección maximal de reglas (pero solo una de ellas se puede ejecutar). Por ejemplo, cuando hay una regla de división y de send-in que pueden ser seleccionadas en una misma membrana. El simulador asume que el sistema P a ser simulado es confluente con el fin de evitar simular el no determinismo. Por tanto, a diferencia de trabajar con todo el árbol de posibles computaciones, el simulador selecciona y simula tan solo una rama de computación, dado que siguiendo cualquier rama se obtiene la misma respuesta (por la propiedad de confluencia). La rama de computación puede ser seleccionada mediante la elección de un multiconjunto de reglas a ser ejecutado con el menor coste. Medimos el coste en término del número de membranas y operaciones de sincronización, ya que son las condiciones que perjudican el rendimiento de la simulación. En este sentido, introducimos las

siguientes prioridades entre tipos de reglas a la hora de ser seleccionadas (de mayor a menor prioridad): disolución (decrementan el número de membranas), evolución (no desarrollan comunicación entre membranas, lo que evita sincronización), send-out (precisan de comunicación entre una membrana y su padre, añadiendo un objeto), send-in (precisan de comunicación entre una membrana y su padre, reservando un objeto), y división (incrementan el número de membranas en el sistema).

Este algoritmo de simulación fue implementado en lenguaje C/C++ [56] dentro un simulador independiente. La primera aproximación fue llevada a cabo con memoria dinámica: los espacios de memoria correspondientes para los objetos y membranas se creaban y eliminaban en tiempo de ejecución según el estado del sistema P simulado. El segundo desarrollo fue enfocado hacia el simulador paralelo en CUDA, obteniendo los resultados presentados en [27, 30]. Dado que estos mostraban una sobrecarga de tiempo por el administrador de memoria dinámica, se realizó un *simulador secuencial rápido* que actuaba de manera semejante al simulador paralelo en CUDA. El rendimiento que ofrece este nuevo simulador es mayor que el original, ya que utiliza memoria estática para la representación del sistema. Este simulador es el que utilizaremos para describir los resultados de este trabajo.

También cabe mencionar que los simuladores reciben como entrada un fichero en formato binario que describe un sistema P, su configuración inicial y su estructura de membranas. La finalidad de este formato binario es la de ahorrar espacio, ganando en eficiencia. Este fichero binario lo crea el parser de pLinguaCore a partir de un fichero en formato P-Lingua, y después de comprobar errores sintácticos y semánticos.

El marco completo de simuladores para sistemas P con membranas activas, incluyendo el simulador secuencial, secuencial rápido, y paralelo con CUDA, se denomina *PCUDA*. Éste es un subproyecto del proyecto PMCGPU, y puede ser descargado del sitio web: `http://sourceforge.net/p/pmcgpu` [12].

4.2. Simulación con GPU y CUDA

El simulador paralelo está compuesto por 5 kernels distintos para implementar las etapas de selección y ejecución del algoritmo de simulación. El primer kernel implementa la etapa de selección, junto a la ejecución de las reglas de evolución. Los otros cuatro kernels implementan la ejecución de los otros tipos reglas (disolución, división, send-out, send-in).

A la hora de diseñar un código en CUDA, hay que identificar el trabajo

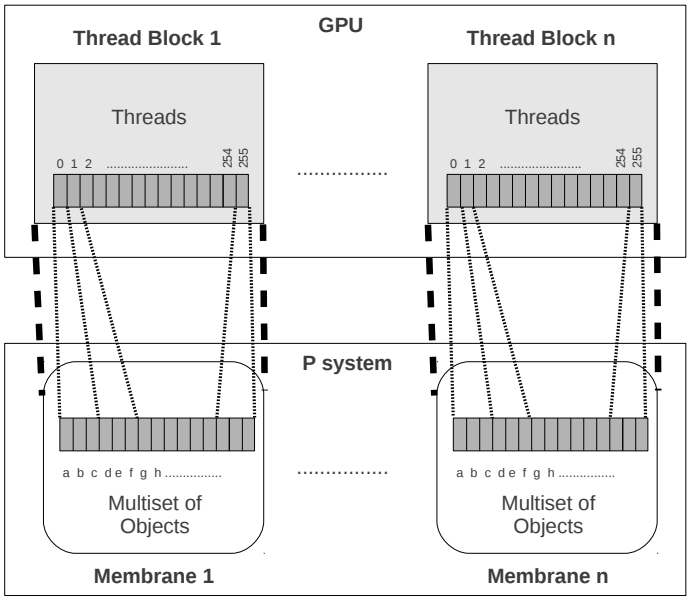


Figura 4.1: Diseño básico del simulador paralelo en GPU.

que va a realizar cada bloque de hilos y a su vez, cada hilo. En este caso, el diseño general de los kernels distribuye los bloques e hilos como sigue (ver la Figura 4.1). Se atribuye cada membrana del sistema simulado a cada bloque de hilos. Así conseguimos identificar el paralelismo entre membranas con el paralelismo entre los bloques de hilos. Con esta decisión de diseño hay que tomar precauciones, ya que puede existir comunicación entre membranas (si están conectadas según el árbol jerárquico), mientras que los bloques de hilos son todos independientes entre sí. La comunicación se debería de realizar, en todo caso, mediante el uso de sucesivas llamadas a kernels.

Por otro lado, cada hilo dentro de un bloque representa un tipo de objeto (o varios) del alfabeto del sistema P simulado. Puesto que todos los bloques deben tener el mismo número de hilos, la parte común que existe para todas las membranas es todo el alfabeto de objetos (ya que los multiconjuntos son distintos para cada una). Con este aspecto de diseño se consigue tener un número de bloques con el mismo número de hilos. Cabe destacar que muchos hilos trabajarán con objetos que no existen realmente en la membrana, ya que normalmente no todo el alfabeto de objetos está presente en una membrana. De hecho, el simulador asigna varios objetos al mismo hilo con el fin de simular

sistemas P de mayor tamaño, no restringiendo el alfabeto al máximo de hilos por bloque.

De esta forma, cada bloque se ejecuta en paralelo calculando el munticonjunto de reglas seleccionado para su membrana correspondiente, y cada hilo individual es el encargado de identificar las reglas que se pueden ejecutar con el objeto del que se encarga. Para ello se accede a una estructura de datos en memoria global, la cual almacena las reglas del sistema clasificadas por objeto, etiqueta y carga (así, la indexación es más sencilla por parte de un hilo). Un hilo, con un objeto asociado, selecciona una regla si: existe dicho objeto en la membrana, existe una regla correspondiente para ese objeto, etiqueta y carga de la membrana (definida por el bloque donde está el hilo), y no existe otra regla seleccionada que no permita ejecutarse (por ejemplo, si hay una regla de send-in, no puede seleccionarse una de división).

Después de ejecutar la etapa de selección, también se ejecutan en este kernel las reglas de evolución. Esto se hace por tres razones principales: las reglas de evolución no implican comunicación entre membranas (por lo tanto, cada bloque puede ejecutar las reglas una vez haya realizado la etapa de selección, ya que no tiene que esperar a sincronizarse con otros por no haber comunicación), la ejecución de las reglas de evolución se hace de forma maximal, y además esta decisión nos permite usar una cantidad menor de memoria global al no tener que almacenar la selección de reglas de evolución para los kernels de la etapa de ejecución.

El resto de las reglas a ser aplicadas se ejecutan en 4 kernels diferentes, uno por cada tipo de regla, como se comentó anteriormente. Primero se mueven los datos necesarios para la simulación del sistema P a la GPU. Entonces, el código llama al kernel de selección, quien retorna las reglas seleccionadas para la configuración actual del sistema P. De entre las posibles reglas seleccionadas habrá unos tipos de reglas diferentes; por tanto, se identifican dichos tipos a fin de lanzar solo los kernels necesarios para completar la etapa de ejecución. Como se mencionó con anterioridad, el proceso es una iteración hasta cumplir con una condición de parada. Finalmente, y según el tipo de información requerida por el usuario, la CPU copia de nuevo los datos finales que residen en la GPU como la configuración de parada. Con el fin de no saturar la simulación con la transferencia de datos de cada configuración, el simulador solo muestra la cantidad de reglas aplicadas por cada transición y la configuración de parada con la respuesta final del sistema P reconocedor.

4.3. Análisis comparativo de rendimiento

Para evaluar el rendimiento del simulador se ha diseñado una familia de sistemas P, llamados sistemas P de testeo, donde es fácil conocer el número de membranas y el número de objetos en un momento dado. Estos sistemas P de testeo también se ajustan al comportamiento de la GPU ya que solo existen reglas de evolución y división (por lo tanto, no hay comunicación entre membranas ni disolución), y todo objeto en cada membrana siempre evoluciona de acuerdo a una regla. Con esto, se consigue realizar un banco de pruebas con sistemas P de diferente tamaño (en términos de membranas y tipos de objetos definidos en el alfabeto).

El sistema P definido es de la siguiente forma: $\Pi = (O, H, \mu, \omega_1, \omega_2, R)$, donde: $O = \{d, o_i : 0 \leq i < n\}$, $H = \{1, 2\}$, $\mu = [[]_2]_1$, $\omega_1 = \emptyset$,$\omega_2 = O$, $R =$

(i) Reglas de evolución: $[o_i \rightarrow o_i]_2^0, 0 \leq i < n$

(ii) Regla de división: $[d]_2^0 \rightarrow [d]_2^0\,[d]_2^0$

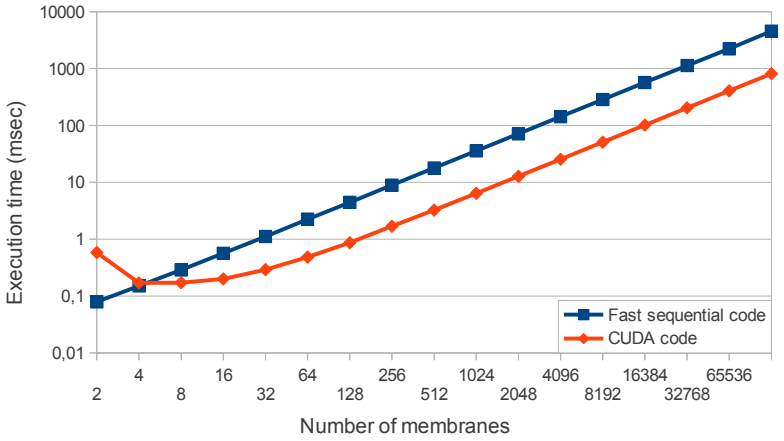
Por lo tanto, los sistemas P de testeo nos permiten controlar el número de objetos en el sistema modificando el parámetro n. Además, el número de reglas va ligado al número de objetos, y el número de membranas en cada paso de computación es igual a 2^s, donde s es el número de paso o transición del sistema. Por último, el número de reglas de evolución seleccionadas y ejecutadas por membrana en cada paso es invariable, ya que hay una regla definida por tipo de objeto, y todos los objetos del alfabeto están presentes en cada membrana etiquetada con 2.

Con el fin de realizar comparativas justas entre el código paralelo (GPU) y el código secuencial (CPU), se desarrolló un simulador secuencial análogo al de CUDA. Este simulador, denominado en adelante *simulador secuencial adaptado* ó simplemente *simulador secuencial*, está basado en el simulador paralelo de tal forma que usa las mismas estructuras de datos, y además los algoritmos paralelos están implementados en bucles. También, existen partes del código que están optimizadas en su versión secuencial mientras que en el código paralelo no era posible (por ejemplo, los nuevos multiconjuntos se crean de forma ordenada en el secuencial, con lo que en la siguiente iteración no es necesario recorrer huecos vacíos en las estructuras de datos).

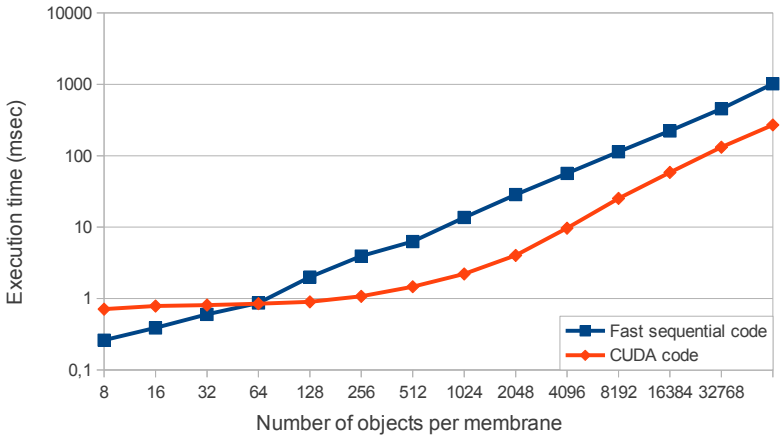
La Figura 4.2 muestra los resultados que se han obtenido para el simulador entre la versión secuencial desarrollada en el lenguaje C++, y el simulador desarrollado en CUDA. Notar que en ambas gráficas, el eje Y está representado de forma exponencial. Por otro lado, el equipo usado en los experimentos consta

de dos GPUs NVIDIA Tesla C1060 conectada a dos procesadores Intel i5 Xeon Nehalem de 4 cores con 12 GB de RAM. El sistema operativo instalado es una distribución Linux Ubuntu Server 10.04 de 64 bits.

Para los test realizados, se usan dos bancos de pruebas basados en los sistemas P de testeo. Estos bancos de pruebas cubren y explotan los dos niveles de paralelismo que los sistemas P tienen de forma natural. El primero testea el paralelismo a nivel de membranas, incrementando el número de éstas de forma exponencial, y el segundo testea el paralelismo a nivel de reglas, incrementando el número de objetos en cada membrana de forma exponencial.



(a) Variando el número de membranas (bloques de hilos)



(b) Variando el número de objetos (hilos por bloque)

Figura 4.2: Rendimiento de la simulación para los simuladores secuencial y basado en CUDA.

La Figura 4.2(a) muestra los resultados para el banco de pruebas que incrementa el número de membranas exponencialmente, manteniendo fijos el número de objetos por membrana (2560 objetos). El simulador para la CPU incrementa su tiempo exponencialmente desde el principio (para cuatro membranas), hasta alcanzar la configuración final con 32768 membranas. Nuestro simulador CUDA, que asigna 256 hilos por bloque (cada hilo maneja 10 objetos por membrana), también incrementa su tiempo de ejecución de forma exponencial, pero la diferencia de rendimiento es notable, y esta diferencia se muestra mejor cuando hay un número mayor de membranas (ya que los recursos de la GPU están totalmente utilizados).

La Figura 4.2(b) muestra el comportamiento de ambos simuladores ejecutando el banco de pruebas que incrementa el número de objetos por membrana. En este caso, el número de membranas se fija a 1024, lo que implica tener un número suficiente de bloques para distribuir el trabajo entre los multiprocesadores. La simulación comienza con pocos objetos por membrana, lo que implica pocos hilos por bloque en CUDA. La Figura 4.2(b) muestra que el código secuencial inicialmente obtiene mejor rendimiento que el código paralelo hasta que la simulación alcanza 32 objetos por membrana. Menos objetos implica tener un número menor de 32 hilos por bloque, lo que no completa un *warp*, por lo que los recursos en la GPU están mal utilizados.

El código secuencial incrementa su tiempo de simulación junto con el número de objetos ya que solamente un hilo trabaja con todos los objetos que aparecen en cada membrana. El tiempo de simulación del código CUDA se mantiene constante hasta alcanzar 256 objetos por membrana. El tiempo incrementa un poco más rápido desde este sistema P ya que los siguientes tamaños implican tener más objetos que hilos por bloque. Por lo tanto, los objetos de cada membrana se distribuyen entre todos los hilos de un bloque: 512 objetos por membrana implica tener dos objetos por hilo, 1024 objetos por membrana implica 4 objetos por hilo, etc. De otra forma, se requeriría un solo hilo sobrecargado, lo que reduciría el rendimiento del simulador, por lo que es mejor usar hilos ligeros.

Por otro lado, se ha llevado a cabo una serie de pruebas con un ejemplo real de la literatura, correspondiente a una solución eficiente a `SAT` con membranas activas (ver la Sección 1.5.1). En los experimentos llevados a cabo, la ejecución de kernels siempre supera en rendimiento al código secuencial. La aceleración conseguida se incrementa con el tamaño de instancia, alcanzando el máximo de 1.67x para 4096 membranas y 914 objetos. Sin embargo, el tiempo completo de simulación paralela (considerando el tiempo de administración de memoria) solo mejora al código secuencial cuando la sobrecarga se oculta por el tiempo

de computación de kernels. Así, se obtiene un máximo de speedup de 1.48x para la instancia mas grande ejecutada en los experimentos.

Sin embargo, cabe destacar que las aceleraciones para este banco de pruebas son muy pequeñas comparada con las conseguidas en el ejemplo simple de test anteriormente descrito en esta sección. El rendimiento en realidad se degrada hasta 4 veces para este ejemplo (desde 5.7x a 1.38x para 65536 membranas en cada sistema P). La principal razón es que existe una bajo grado de intensidad de reglas en el sistema P. Hablaremos de esto con más detalle en la siguiente sección (caracterización de la simulación), pudiendo así entender mejor cuales son las características de los sistemas P que son bien manejados por la GPU.

4.4. Caracterización de la simulación con GPU

Cabe destacar dos restricciones que presenta el simulador paralelo en los sistemas P que puede simular satisfactoriamente (ver Tabla 4.1). Estos surgen por decisiones de diseño y peculiaridades en el modelo de programación CUDA:

- El simulador puede trabajar con sistemas P que tengan solo dos niveles en la jerarquía de membranas (membrana piel y membranas elementales). Esto es una decisión de diseño cuyo fin es el de simplificar el problema de la reserva de objetos por parte de las reglas send-in en la membrana padre (lo que significa accesos concurrentes entre distintos bloques de hilos al mismo espacio de memoria, estrategia a evitar en CUDA).

- El simulador puede trabajar con sistemas P cuyo número de objetos definidos en el alfabeto sea divisible por un número menor de 512 (el máximo número de hilos en un bloque), con el fin de distribuir los objetos entre los hilos de forma equitativa.

Por otro lado, el modelo de memoria de CUDA es manejado de forma estática. Así, el programador debe estimar la cantidad de memoria (global) que requeriría la ejecución de un kernel para reservarla estáticamente antes de lanzarlo. Si la estimación no fuera suficiente para ejecutar y almacenar todos los datos (falta de espacio de memoria en memoria global o compartida), la ejecución de la GPU debería de finalizar, la CPU retomaría el mando y reubicaría los recursos en memoria global, reservando más espacio, y ejecutando de nuevo el kernel. Esto provocaría un gasto de tiempo muy grande, por lo que la memoria estática es un aspecto clave a tener en cuenta cuando se diseña soluciones en CUDA. Una solución, simple pero eficaz, sería la de reservar

Tabla 4.1: Principales limitaciones en el simulador paralelo

Parámetro	**Limitación**
Niveles en la jerarquía de membranas	2
Máximo tamaño del alfabeto	65535
Máximo tamaño del conjunto de etiquetas	65535
Máxima multiplicidad en un objeto en una membrana elemental	65535
Tamaño del alfabeto	Divisible por un número menor que 512

la memoria suficiente como para almacenar el peor caso (mayor tamaño que puede tomar una instancia) de un problema.

Para los sistemas P con membranas activas, el peor caso se presentaría cuando, en una configuración dada, todos los diferentes objetos definidos en el alfabeto están presentes en el multiconjunto de una membrana. Aunque este es el peor caso a tratar, no se presenta en la mayoría de los sistemas P (aunque los sistemas P de testeo sí que cumple tener siempre el tamaño de su peor caso). Por lo tanto, el rendimiento del simulador depende totalmente del sistema P simulado.

En este sentido, hay algunas características que deberían de ser consideradas cuando se diseñen soluciones de sistemas P con el fin de acelerar con éxito su simulación:

- **Densidad de objetos por membrana**: Con el fin de incrementar la utilización de los hilos de CUDA, deberían de aparecer más objetos diferentes en los multiconjuntos de cada membrana.
- **Intensidad de reglas**: Todos, o la mayoría de objetos deberían de reaccionar según las reglas definidas. En este caso, se podría contemplar la reutilización de los objetos definidos en los sistemas P a la hora de diseñar soluciones.
- **Reducción de la comunicación entre las membranas**: Deberían de haber menos objetos enviados o retomados de la piel, lo que reduciría la comunicación y la sincronización.

"Si P = NP, entonces el mundo sería un lugar profundamente diferente al que estamos acostumbrados. No habría valor en los "saltos creativos", no habría diferencia fundamental entre resolver un problema y reconocer la solución una vez se encuentre. Cualquiera que pudiera apreciar una sinfonía sería Mozart; cualquiera que pudiera seguir un argumento paso a paso seria Gauss..."

Scott Aaronson, MIT

5

Simulación Paralela de Sistemas P en Soluciones del problema SAT

A la hora de simular sistemas P en la GPU, los mayores retos que nos podemos encontrar son (1) el manejo dinámico del espacio de memoria y (2) el crecimiento exponencial del espacio de trabajo en cuanto se incrementa el número de variables implicadas a la hora de ejecutar la simulación.

Después de analizar con detenimiento el simulador flexible para sistemas P con membranas activas (ver Capítulo 4), se pudo observar que el caso de los sistemas P de testeo presentaba una densidad de objetos del 100 % (todos los objetos del alfabeto están presentes en las membranas y evolucionan), obteniendo un buen rendimiento en su simulación. Sin embargo, para la familia $\mathbf{\Pi_{am-SAT}}$ [86] (Sección 1.5.1), la densidad de objetos es de media un 15 %, por lo que un 85 % de hilos están siendo desaprovechados en la simulación dando lugar a la pérdida de eficiencia de la simulación en general.

Dado que el comportamiento del simulador CUDA de membranas activas no era el deseado para este tipo de soluciones, el siguiente paso fue desarrollar un simulador específico para la familia $\mathbf{\Pi_{am-SAT}}$. Usando como base el diseño del simulador anterior, y optimizando el uso de hilos y de memoria, se consigue acelerar de forma notoria el tiempo de simulación para instancias de `SAT` grandes.

En este capítulo analizamos el comportamiento de la GPU simulando con-

cretamente los sistemas P de la familia $\mathbf{\Pi_{am-SAT}}$. Este trabajo fue primeramente publicado en [26], y será detallado en la Sección 5.1. Además ha sido mejorado para adaptarse a las nuevas arquitecturas de sistemas de GPUs y multi-GPUs [29, 32] y para supercomputadores [28].

Analizaremos también la simulación paralela de otra solución eficiente a `SAT` basada en una familia de sistemas P de tejido con división celular (Sección 1.5.2). Este simulador fue presentado en [82], y está detallado en la Sección 5.2. El objetivo de este simulador es estudiar cuales son los ingredientes de diferentes variantes de sistemas P que están bien posicionadas para ser manejadas por la GPU. Esto se lleva a cabo mediante una comparativa de rendimiento con el simulador en GPU de sistemas P con membranas activas. El análisis de rendimiento y la caracterización de la simulación se explica en las Secciones 5.3 y 5.3.4, respectivamente.

5.1. Simulación paralela de una solución con sistemas P a modo de células

En esta sección describimos el simulador para la familia $\mathbf{\Pi_{am-SAT}}$, descrita en la Sección 1.5.1 (Capítulo 1). El marco de todos estos simuladores se denomina *PCUDASAT*, y puede ser descargado desde la página web correspondiente al proyecto PMCGPU: `http://sourceforge.net/p/pmcgpu` [12].

5.1.1. Diseño del simulador base: simulador secuencial

Como se mencionó previamente, el modelo de programación CUDA está basado en el lenguaje C/C++. Por tanto, el primer paso recomendado cuando se desarrolla aplicaciones en CUDA es comenzar con el algoritmo base escrito en C++. El diseño del simulador secuencial está basado en las cuatro etapas principales de la computación de los sistemas P de la familia, como se describe la Sección 1.5.1: generación, sincronización, chequeo y salida. Todas estas etapas son ejecutadas secuencialmente en este simulador, reproduciendo así el comportamiento del sistema P.

Primero, se ejecuta la etapa de generación, creando 2^n membranas mediante la división de cada una en n pasos, donde n es el número de variables de una instancia del problema `SAT`. Después, el simulador ejecuta la etapa de sincronización y la etapa de chequeo, que determina las membranas que codifican una solución (donde todas las cláusulas son verdad) del problema, y finalmente la etapa de salida lanza la respuesta al entorno.

La entrada del simulador es un fichero en formato DIMACS CNF, donde se codifica la fórmula en FNC. La salida se lee del entorno del sistema P, donde el resultado está almacenado al final de la ejecución. Es importante remarcar que la semántica de los sistemas P de la familia se reproduce en el algoritmo de simulación, por lo que el simulador es específico.

5.1.2. Diseño del simulador basado en GPU: simulador paralelo

El objetivo de este simulador paralelo es simular completamente el comportamiento de la computación del sistema P, haciéndolo en paralelo con la GPU. Para ello, el diseño se basa en el del simulador secuencial, el cual recrea las cuatro etapas de la computación del sistema P. Las primeras tres etapas se han desarrollado como kernels de CUDA, y la última (etapa de salida) se ejecuta en la CPU.

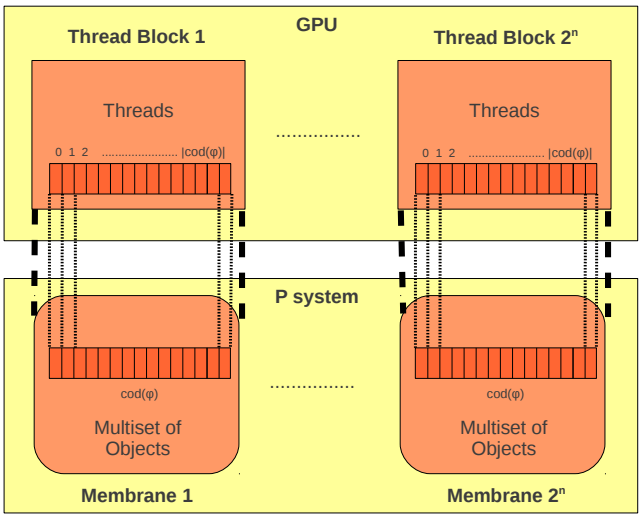


Figura 5.1: Diseño general del simulador paralelo, asignando bloques de hilos a membranas (asignación de verdad), e hilos a objetos del multiconjunto de entrada (literal de la fórmula).

De forma similar al diseño del simulador para sistemas P de membranas activas, este simulador asigna un bloque de hilos a cada membrana, como se muestra en la Figura 5.1. De esta forma, se implementa con CUDA el paralelismo entre membranas del sistema P. A su vez, cada hilo es asignado a cada

objeto del multiconjunto de entrada, el cual corresponde con cada literal de la instancia de `SAT`. Este diseño es común a todos los kernels definidos.

La etapa de generación es simulada usando tres kernels que computan las reglas correspondientes. Esto es un proceso iterativo de n pasos donde los kernels se llaman n veces. En cada iteración el simulador ajusta el número de bloques de hilos antes de llamar al kernel, ya que se crean nuevas membranas.

Cuando el espacio de trabajo exponencial se crea, las etapas de sincronización y de chequeo se ejecutan siguiendo las reglas correspondientes. Ambas etapas se ejecutan en el mismo kernel, y por tanto, en paralelo a cada membrana. La sincronización global no es necesaria ya que no existe comunicación entre las membranas internas en estas etapas de la computación. Finalmente, la etapa de salida se desarrolla en la CPU, chequeando la condiciones y lanzando el resultado de la computación.

5.1.3. Adaptando el simulador a la arquitectura de la GPU: simulador híbrido

Aunque el simulador paralelo reproduce totalmente la computación del sistema P con un algoritmo de simulación específico y optimizado, quizás otro diseño del sistema P podría obtener mejor rendimiento siempre que sea simulado por GPUs. En este sentido, el simulador híbrido [1] usa algunas heurísticas en la simulación para adaptarla a la idiosincrasia de la arquitectura GPU. En concreto, el objetivo de este simulador es el de reducir la sobrecarga de comunicación y sincronización.

La etapa de generación es simulada de forma análoga al simulador paralelo, creando un espacio de trabajo exponencial en el sistema y reproduciendo los mismos pasos, pero incluyendo todo dentro del mismo kernel, reduciendo la sobrecarga de sincronización y de lanzamiento de kernels.

Además, el kernel que representa la etapa de chequeo difiere sustancialmente, realizando de una forma más rápida la salida. Con esta modificación, el kernel presenta más paralelismo de datos cuando se comprueba las cláusulas. Para este propósito, cada hilo chequea si su correspondiente objeto codifica una cláusula cierta. Si es así, una variable compartida (una por bloque de hilo y cláusula) se hace cierta. Con esto se consigue hacer el chequeo de cláusulas de forma paralela, y no de forma secuencial como se hace en el modelo.

[1] Se denomina *simulador híbrido* puesto que no realiza exactamente los mismos pasos computacionales que el sistema P teórico, pero obtiene el mismo resultado.

5.2. Simulación paralela de la solución con sistemas P de tejidos

En esta sección describimos el simulador para la familia $\mathbf{\Pi_{tsp-SAT}}$, descrita en la Sección 1.5.2. Este marco de simulación se denomina *TSPCUDASAT*, y puede ser descargado de la página web correspondiente al proyecto PMCGPU: `http://sourceforge.net/p/pmcgpu` [12].

5.2.1. Simulación secuencial y estructuras de datos

Para una implementación sencilla, el algoritmo de simulación ha sido divido en cinco fases de simulación. Cabe destacar que éstas difieren en número a las fases denotadas en el modelo teórico (ver Sección 1.5.2):

- *Fase de generación* (reglas (a) a (e)): cubre dos fases del modelo teórico fase de generación de asignaciones de verdad y generación de contadores.
- *Fase de intercambio* (reglas (f) a (g)): comprende la primera parte de la fase de chequeo del modelo teórico.
- *Fase de sincronización* (reglas (h) a (m)): comprende la segunda parte de la fase de preparación de chequeo.
- *Fase de chequeo* (reglas (n) a (p)): es la fase de chequeo de cláusulas.
- *Fase de salida* (reglas (q) a (t)): realiza las fases de chequeo de fórmula y salida del modelo teórico.

El simulador secuencial implementa estas cinco fases de simulación directamente en código fuente (C++). Cada una trabaja directamente con la estructura de fases que se explica a continuación. La entrada del simulador es la misma que la usada en el simulador para la familia $\mathbf{\Pi_{am-SAT}}$: un fichero en formato DIMACS CNF, y el simulador devuelve la respuesta a la simulación. Como se puede apreciar, actúa como un resolvedor de `SAT`, pero con un motor de inferencia basado en sistemas P.

Para esta solución, la representación de un sistema P de tejidos de la familia (fijando n y m) es doble para cada tipo de célula, y los elementos de las células se codifican dentro de enteros de 32 bits. La célula etiquetada por 1 se representa como un array que tiene una dimensión constante de 5 elementos: los tres contadores b, c y d (inicialmente en la célula), y los dos objetos *yes* y *no* (la respuesta final).

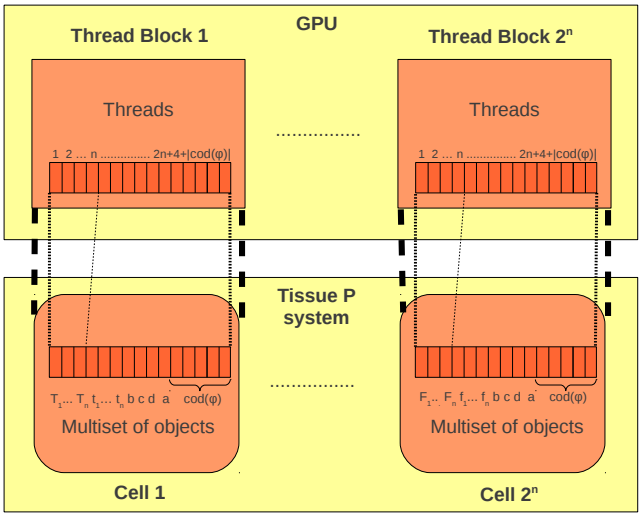


Figura 5.2: Diseño general del simulador paralelo para $\mathbf{\Pi_{tsp-SAT}}$.

Las células etiquetadas por 2 también son representadas por un array unidimensional. Todas ellas se almacenan dentro de un array global de gran tamaño, ya que se reserva inicialmente para almacenar el máximo número de células (2^n). Hemos determinado que el máximo número de objetos que aparecen en una célula 2 es $(2n) + 4 + |cod(\varphi)|$.

5.2.2. Diseño del simulador paralelo

El simulador paralelo está diseñado para también reproducir completamente una computación de los sistemas de la familia de sistemas P de tejidos. A diferencia del simulador para $\mathbf{\Pi_{am-SAT}}$, no se ha desarrollado un simulador híbrido. El diseño de este simulador paralelo se guía por la mencionada estructura de fases, usando kernels de CUDA separados para cada una.

Se aplica una asignación de trabajo en CUDA similar a la de otros simuladores [27, 26], resumiéndose en la Figura 5.2. Cada bloque de hilos se corresponde a cada célula etiquetada por 2 creada en el sistema. Sin embargo, a diferencia del anterior, la asignación de cada hilo difiere en cada fase:

- *Fase de generación*: el número de bloques de hilos se incrementa iterativamente junto al número de células en cada paso de computación. Cada bloque de hilos contiene $(2n) + 4 + |cod(\varphi)|$ hilos. Los hilos son por tanto usados para copiar cada elemento de la célula que se divide.

- *Fase de ejecución*: se ejecuta dentro del kernel de la fase de generación, usando la misma cantidad de bloques de hilos, y solo usando los hilos correspondientes para realizar el intercambio al final de la fase.

- *Fase de sincronización*: el numero de hilos es n (número de variables). Si se usara la misma cantidad de hilos que en la fase de generación, casi todos estarían desocupados: preferimos lanzar menos hilos, pero realizar trabajo efectivo. Hemos corroborado experimentalmente este hecho.

- *Fase de chequeo*: el número de hilos es $|cod(\varphi)|$. Esto es, cada hilo es usado para ejecutar, en paralelo, las reglas (n) a (o). El resultado a nivel de resolución del problema `SAT` es que cada hilo comprueba si un literal hace verdadera la cláusula.

- *Fase de salida*: las reglas del tipo (q) son ejecutadas secuencialmente en un kernel separado, usando otra vez $|cod(\varphi)|$ hilos por bloque, y 2^n bloques.

Cabe mencionar que para esta solución hemos llevado a cabo una serie de pequeñas mejoras de rendimiento centradas en la implementación en GPU. Hemos identificado que el simulador se ejecuta dos veces más rápido usando estas mejoras, las cuales están orientadas a dos aspectos del rendimiento de la computación con GPU. El primero de ellos es *resaltar el paralelismo*, que tiene como objetivo incrementar el número de hilos por bloque (a la cantidad recomendada entre 64 y 256). La segunda es sobre *explotar el ancho de banda de streaming*. Para ello, los datos se cargan primero a la memoria compartida y operados allí, evitando accesos costosos a memoria global. Usaremos en adelante el simulador mejorado.

5.3. Análisis de rendimiento

En esta sección mostramos las pruebas de rendimiento llevadas a cabo para los simuladores introducidos para la familias $\mathbf{\Pi_{am-SAT}}$ y $\mathbf{\Pi_{tsp-SAT}}$. Todos los experimentos fueron ejecutados en un servidor Ubuntu server Linux 10.04 de 64 bits, con un procesador (a 2 GHz) dual-socket Intel i5 Xeon Nehalem de 4 núcleos, 12 GBytes de RAM, y dos GPUs NVIDIA Tesla C1060 (240 núcleos a 1.30 GHz, 4 GBytes de memoria). Los ficheros de entrada (en formato DIMACS CNF) para los simuladores se han generado aleatoriamente con el programa WinSAT [13].

5.3.1. Simulador a modo de células

En esta sección, analizamos el rendimiento de los tres simuladores a modo de células: el simulador secuencial (desde ahora, *am-sat-seq*), el simulador en CUDA (*am-sat-gpu*) y el simulador híbrido (*am-sat-gpu-hyb*). Hemos desarrollado un banco de pruebas incrementando el número de bloques de hilos por grid (variando el número de variables de la fórmula), y manteniendo fijo el número de hilos por bloque (número de literales) a 256.

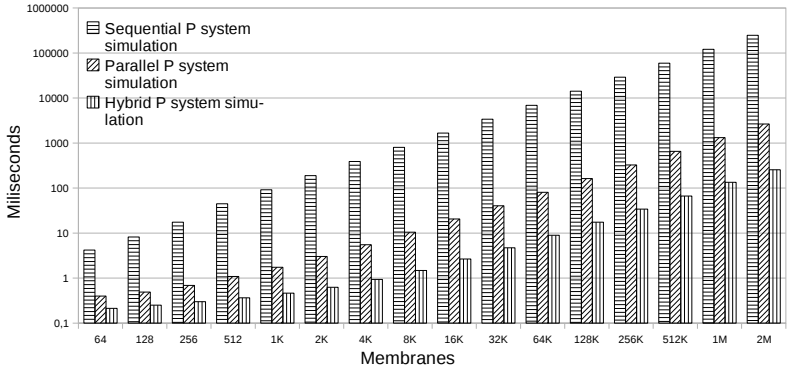


Figura 5.3: Tiempo de simulación de *am-sat-seq*, *am-sat-gpu* y *am-sat-gpu-hyb*.

La Figura 5.3 muestra el rendimiento experimental de los simuladores (en escala logarítmica). Se incrementa el número de variables en la fórmula hasta alcanzar una configuración de 2^{11} membranas. Una vez que los recursos de la GPU se ocupan completamente, el tiempo de ejecución incrementa linealmente con el número de bloques. En este caso, se reporta hasta un 94x de speedup entre el simulador *am-sat-gpu* y *am-sat-seq*. Sin embargo, la figura muestra que el speedup se mantiene constante (10x) cuando el número de membranas es mayor a 128 K. Este es el número de bloques que llena el pipeline de la GPU, teniendo el simulador híbrido un mejor rendimiento general que el paralelo.

5.3.2. Simulador a modo de tejidos

A continuación mostramos la comparativa de rendimiento entre los dos simuladores a modo de tejidos: el simulador secuencial (*tsp-sat-seq*), y el paralelo en GPU (*tsp-sat-gpu*). Para este análisis usaremos el mismo test utilizado para el caso a modo de células.

La Figura 5.4 muestra el speedup alcanzado por el simulador *tsp-sat-gpu*, tomando en cuenta además el tiempo consumido por el manejo de memoria

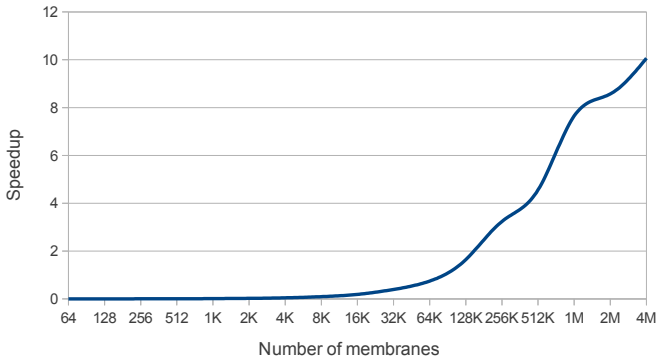


Figura 5.4: Speedup para *tsp-sat-gpu* vs *tsp-sat-seq*, con manejo de datos.

(reserva y transferencia). Se observa que, dado que el tiempo de manejo de datos es fijo para todos los tamaños, el speedup supera la unidad solo a partir de 128 K células. Los sistemas con un número menor de células se ejecutan más lentos en la GPU por este hecho. Sin embargo, para sistemas suficientemente grandes, el speedup es tal como si considerásemos solo el tiempo de kernel. El máximo speedup se obtiene para 4 M células, hasta 10x.

5.3.3. Comparativa modo células vs tejidos

A continuación compararemos los simuladores para las soluciones basadas en sistemas P con membranas activas (*am-sat-gpu*) y sistemas P de tejido con división celular (*tsp-sat-gpu*). Destacaremos las siguientes diferencias:

- Pasos de computación: fijados m (cláusulas) y n (variables), números naturales, los sistemas P a modo de células toman $5n + 2m + 3$ pasos, y los de tejido $2n + 2m + nm + 1$. Por tanto, la computación de la solución basada en tejidos es mayor en número de pasos, si $m > 3 + \frac{2}{n} \simeq 3$.
- Fases: *am-sat-gpu* consta de 4 fases (en 3 kernels), mientras que *tsp-sat-gpu* usa 5 fases (en 4 kernels).
- Requerimientos de memoria: cada membrana en *am-sat-gpu* se representa por $|cod(\varphi)|$ elementos, mientras que *tsp-sat-gpu* usa $2n + 4 + |cod(\varphi)|$. En total, *tsp-sat-gpu* requiere $(2n + 4)2^n$ bytes extra.

La Figura 5.5 compara ambas soluciones. Podemos observar que los kernels de *am-sat-gpu* mejoran a *tsp-sat-gpu* en hasta 2.9x. Sin embargo, si consideramos el manejo de datos en la GPU podemos ver que el comportamiento es

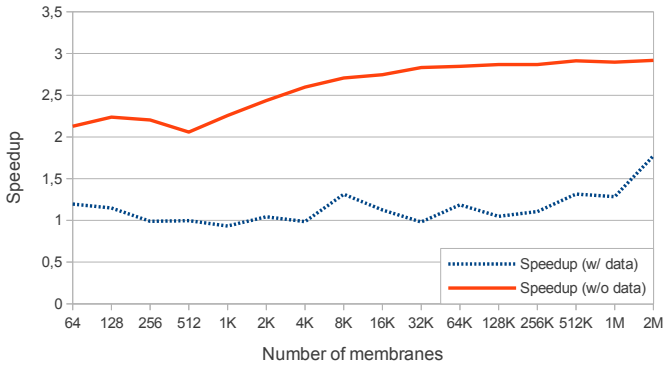


Figura 5.5: Speedup para *tsp-sat-gpu* vs *am-sat-gpu*, con y sin manejo de datos.

similar, aunque *am-sat-gpu* se ejecuta un poco mejor, y para 2 M membranas, el speedup es casi 1.8x. Dado que los kernels de *tsp-sat-gpu* han sido adaptados a la GPU, podríamos pensar que *am-sat-gpu* podría aun obtener mas ventaja.

5.3.4. Caracterizando la simulación en GPU

En esta sección caracterizaremos las simulaciones llevadas a cabo. Identificaremos las dos propiedades que hacen que *am-sat-gpu* mejore en rendimiento a *tsp-sat-gpu*.

- *Cargas*: el modelo de los sistemas P con membranas activas asocian cargas a las membranas. Estas pueden ser usadas para almacenar información si son usadas de manera efectiva, como por ejemplo para codificar asignaciones de verdad, ahorrándose así memoria en almacenamiento de objetos que reproduzcan este comportamiento.

- *Reglas sin cooperación*: el modelo de los sistemas P con membranas activas define reglas sin cooperación (un solo objeto en la parte izquierda). Esta propiedad ayuda a los hilos asignados a cada regla a trabajar con un solo objeto en paralelo. Reglas con un grado de cooperación superior requieren pasos intermedios de sincronización en la simulación.

Parte III

Simulación Paralela Aplicada a Modelos Computacionales en Biología

"Esencialmente, todos los modelos son erróneos, pero algunos son útiles."

George E. P. Box

6

Algoritmos de Simulación para Sistemas P de Dinámica de Poblaciones

La Computación Celular con Membranas engloba tanto el estudio de la base teórica de los modelos como las aplicaciones de los mismos a áreas como la Biología de Sistemas [33, 98, 85, 80, 36] y la Dinámica de Poblaciones [36, 22, 37]. En este último sentido, los *sistemas P de Dinámica de Poblaciones*, o sistemas PDP, forman un marco de modelización de dinámica de poblaciones [35, 21, 64], que permite la evolución simultánea de un gran número de especies, además de poder manejar un gran número de objetos auxiliares.

Desde su introducción, se han definido varios algoritmos con el fin de capturar de manera más fidedigna la semántica del marco de modelización. Estos algoritmos seleccionan las reglas de acuerdo con la probabilidad asociada, a la vez que mantienen la semántica del paralelismo maximal de los sistemas P.

En este capítulo introducimos el marco formal de los sistemas PDP (Sección 6.1). Describiremos el primer algoritmo diseñado, denominado *BBB (Binomial Block Based algorithm)*, en la Sección 6.2. Con respecto al presente trabajo, se han definido dos nuevos algoritmos, denominados *DNDP (Direct Non-Deterministic distribution with Probabilities)* (Sección 6.3) y *DCBA (Direct distribution based on Consistent Blocks Algorithm)* (Sección 6.4).

6.1. Sistemas P de Dinámica de Poblaciones

Los *sistemas P de Dinámica de Poblaciones* son una variante de los *sistemas P multientorno con membranas activas* [36], el cual es un modelo formado por una red de entornos, cada uno conteniendo un sistema P que utiliza características como cargas eléctricas. Todos los sistemas P comparten el mismo *esqueleto*, en el sentido de que tienen el mismo alfabeto de trabajo, la misma estructura de membranas y el mismo conjunto de reglas. Sin embargo, en este marco, cada regla tiene asociada una función de probabilidad que puede variar según el entorno.

Definición 6.1. *Un sistema PDP de grado* (q, m)*,* $q, m \geq 1$*, tomando* $T \geq 1$ *unidades de tiempo, es una tupla*

$$\Pi = (G, \Gamma, \Sigma, T, \mathcal{R}_E, \mu, \mathcal{R}, \{f_{r,j} : r \in \mathcal{R}, 1 \leq j \leq m\}, \{\mathcal{M}_{i,j} : 1 \leq i \leq q, 1 \leq j \leq m\})$$

donde:

- $G = (V, S)$ *es un grafo dirigido. Sea* $V = \{e_1, \ldots, e_m\}$*.*
- Γ *y* Σ *son alfabetos tales que* $\Sigma \subsetneq \Gamma$*.*
- T *es un número natural.*
- $\mathcal{R}_E$ *es un conjunto finito de reglas de la forma* $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$*, donde* $x, y_1, \ldots, y_h \in \Sigma$*,* $(e_j, e_{j_l}) \in S, 1 \leq l \leq h$*, y* $p_r : \{1, \ldots, T\} \longrightarrow [0, 1]$ *es una función computable tal que para cada* $e_j \in V$ *y* $x \in \Sigma$*, la suma de las funciones asociadas con las reglas del tipo* $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$ *es la función constante* 1*.*
- μ *es un árbol enraizado etiquetado por* $1 \leq i \leq q$*, y por símbolos del conjunto* $EC = \{0, +, -\}$*.*
- $\mathcal{R}$ *es un conjunto finito de reglas de la forma* $u[v]_i^\alpha \rightarrow u'[v']_i^{\alpha'}$*, donde* $u, v, u', v' \in M_f(\Gamma)$*,* $u + v \neq \emptyset$*,* $1 \leq i \leq q$ *y* $\alpha, \alpha' \in \{0, +, -\}$*, tal que no existe ninguna regla* $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$ *ni* $u[v]_i^\alpha \rightarrow u'[v']_i^{\alpha'}$ *con* $x \in u$*.*
- *Para cada* $r \in \mathcal{R}$ *y* $1 \leq j \leq m$*,* $f_{r,j} : \{1, \ldots, T\} \longrightarrow [0, 1]$ *es una función computable tal que para cada* $u, v \in M_f(\Gamma)$*,* $1 \leq i \leq q$*,* $\alpha, \alpha' \in \{0, +, -\}$ *y* $1 \leq j \leq m$*, la suma de las funciones* $f_{r,j}$ *con* $r \equiv u[v]_i^\alpha \rightarrow u'[v']_i^{\alpha'}$*, es la función constante* 1*.*

- *Para cada* i, j $(1 \leq i \leq q, 1 \leq j \leq m)$, $\mathcal{M}_{i,j}$ *es un multiconjunto finito sobre* Γ.

Un sistema PDP definido como antes puede ser visto como un conjunto de m entornos $e_1, \ldots, e_m$ interconectados por los arcos de un grafo dirigido G. Cada entorno e_j solo puede contener símbolos del alfabeto Σ y todos ellos también contienen un esqueleto de sistema P, $\Pi_j = (\Gamma, \mu, \mathcal{M}_{1,j}, \ldots, \mathcal{M}_{q,j}, \mathcal{R})$, de grado q, donde:

(a) Γ es el alfabeto de trabajo cuyos elementos son denominados objetos.

(b) μ es un árbol enraizado que describe una estructura de membranas consistente en q membranas etiquetadas de forma inyectiva por $1, \ldots, q$. La membrana piel (raíz del árbol) está etiquetada por 1. También asociamos las cargas eléctricas del conjunto $\{0, +, -\}$ con las membranas.

(c) $\mathcal{M}_{1,j}, \ldots, \mathcal{M}_{q,j}$ son multiconjuntos finitos sobre Γ, describiendo los objetos ubicados inicialmente en las q regiones de μ, dentro del entorno e_j.

(d) $\mathcal{R}$ es el conjunto de las reglas de evolución de cada sistema P. Toda regla $r \in \mathcal{R}$ en Π_j tiene una función computable $f_{r,j}$ asociada con ella. Para cada entorno e_j, denotamos por $\mathcal{R}_{\Pi_j}$ el conjunto de reglas con probabilidades obtenidos mediante la unión de cada $r \in \mathcal{R}$ con la función correspondiente $f_{r,j}$.

Además se dispone de un conjunto $\mathcal{R}_E$ de reglas de comunicación entre entornos, y el número natural T representa el tiempo de simulación del sistema. El conjunto de reglas de todo el sistema es $\bigcup_{j=1}^{m} \mathcal{R}_{\Pi_j} \cup \mathcal{R}_E$.

La *semántica* de los sistemas PDP está definida a través de un modelo sincronizado (asumiendo un reloj global) y no determinista. A continuación describimos los aspectos semánticos de estos sistemas.

Una regla de evolución $r \in \mathcal{R}$, de la forma $u[\,v\,]_i^{\alpha} \rightarrow u'[\,v'\,]_i^{\alpha'}$, es aplicable a cada membrana etiquetada por i, cuya carga eléctrica es α, que contiene el multiconjunto v, y que su padre contiene el multiconjunto u. Cuando dicha regla se aplica, los objetos de los multiconjuntos v y u se eliminan de la membrana i y de la membrana padre, respectivamente. De manera simultánea, los objetos del multiconjunto u' se añade a la membrana padre de i, y los objetos del multiconjunto v' se introducen en la membrana i. La aplicación también reemplaza la carga de la membrana i a α'. En cada entorno e_j, la regla r tiene

asociada una función de probabilidad $f_{r,j}$ que provee un índice de la aplicabilidad cuando varias reglas compiten por los objetos. En el modelo, el grado de cooperación viene dado por $|u| + |v|$.

Una regla $r \in \mathcal{R}_E$, de la forma $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \dots (y_h)_{e_{j_h}}$ es aplicable al entorno e_j si contiene el objeto x. Cuando dicha regla se aplica, el objeto x pasa de e_j a $e_{j_1}, \dots, e_{j_h}$ modificándose en los objetos $y_1, \dots, y_h$ respectivamente. En cualquier momento t $(1 \leq t \leq T)$, para cada objeto x en el entorno e_j, si existe una regla de comunicación del tipo $(x)_{e_j} \xrightarrow{p_r} (y_1)_{e_{j_1}} \dots (y_h)_{e_{j_h}}$, entonces una de esas reglas será aplicada. Si más de una regla pudiera ser aplicada a un mismo objeto en un instante dado, el sistema seleccionará una regla aleatoriamente, de acuerdo a su probabilidad que viene dado por $p_r(t)$.

Para cada j $(1 \leq j \leq m)$, existe otra restricción concerniente a la consistencia de cargas: a fin de aplicar simultáneamente varias reglas de $\mathcal{R}_{\Pi_j}$ a la misma membrana,

Una *descripción instantánea* o *configuración* del sistema en cualquier instante t es una tupla de los multiconjuntos de objetos presentes en los m entornos en cada una de las regiones de cada Π_j, junto a las polarizaciones de las membranas en cada sistema P. Asumiremos que todos los entornos están inicialmente vacíos y que todas las membranas inicialmente tienen polarización neutral.

En cada unidad de tiempo, una configuración dada se transformará en otra configuración mediante la aplicación de reglas del sistema como sigue: en cada paso de transición, las reglas a aplicar son seleccionadas de forma no determinista de acuerdo a las probabilidades asignadas a ellas, y todas las reglas aplicables son aplicadas simultáneamente de forma maximal. De esta forma, obtenemos una *transición* desde una configuración a la siguiente del sistema. Diremos que una *computación* es una secuencia de configuraciones de tal forma que el primer término de la secuencia es la configuración inicial, y cada configuración no inicial se obtiene de la anterior mediante la aplicación de reglas.

En [21] se presentó el marco general basado en sistemas P para la modelización de dinámica de ecosistemas, el cual ha sido usado para ecosistemas reales, como en el de unas aves carroñeras del Pirineo Catalán [22], y el mejillón cebra en el pantano de Ribarroja [21]. El mencionado marco de modelización tiene asociadas una herramientas informáticas que ofrecen, entre otras, las siguientes características: una interfaz de usuario gráfica (entendiéndose por usuario los expertos ecólogos y los diseñadores de modelos), definición del modelo y parámetros del ecosistema, ejecución de simulaciones, creación de datos estadísticos en forma de tablas, gráficas etc. Esta aplicación se denomina Me-

CoSim [84], y su núcleo es pLinguaCore [46], el cual carga un fichero definido en P-Lingua, configura los correspondientes parámetros, llama a pLinguaCore y colecciona los resultados de la simulación.

Como hemos visto en los capítulos anteriores, a fin de simular un modelo de sistemas P es necesario definir algoritmos de simulación que sean capaces de representar los elementos sintácticos (estructura de membranas, multiconjunto de objetos, reglas, etc.) y de reproducir fielmente la semántica (cómo se aplican las reglas) del modelo. Con respecto a los sistemas PDP, remarcamos las siguientes propiedades semánticas que deberían de ser reflejadas en un algoritmo de simulación: comportamiento probabilístico (reproducción de la aleatoriedad inherente a los procesos de la naturaleza), competición de recursos (las reglas dictaminan como evolucionan grupos de individuos, y además, cómo éstos compiten entre sí), maximalidad del modelo, y consistencia de reglas.

6.2. Binomial Block Based algorithm (BBB)

En esta sección describiremos el algoritmo BBB, que es el punto de partida de los algoritmos desarrollados y presentados en este trabajo. Este algoritmo fue presentado en [21], y está incluido en la librería pLinguaCore [46].

Consideremos un sistema PDP de grado (q, m) con $q \geq 1,\ m \geq 1$, tomando T unidades de tiempo, $T \geq 1$ (ver Sección 6.1), $\Pi = (G, \Gamma, \Sigma, T, \mathcal{R}_E, \mu, \mathcal{R}, \{f_{r,j} : r \in \mathcal{R}, 1 \leq j \leq m\}, \{\mathcal{M}_{i,j} : 1 \leq i \leq q, 1 \leq j \leq m\})$

La computación del sistema es una secuencia de configuraciones C_t, $0 \leq t \leq T$ construida por la aplicación de las reglas de $R = \bigcup_{j=1}^{m} \mathcal{R}_{\Pi_j} \cup \mathcal{R}_E$. El algoritmo está basado en el concepto de bloque de reglas, el cual se introduce a continuación (Definiciones 6.2, 6.3, y 6.4).

Definición 6.2. *La parte izquierda de las reglas se define como sigue:*

(a) *Dada una regla* $r \in \mathcal{R}_E$ *de la forma* $(x)_{e_j} \xrightarrow{p} (y_1)_{e_{j_1}} \cdots (y_h)_{e_{j_h}}$ *donde* $e_j \in V$ *y* $x, y_1, \ldots, y_h \in \Sigma$*:*

- *La parte izquierda de* r *es* $LHS(r) = (e_j, x)$.
- *La parte derecha de* r *es* $RHS(r) = (e_{j_1}, y_1) \cdots (e_{j_h}, y_h)$.

(b) *Dada una regla* $r \in \mathcal{R}$ *de la forma* $u[v]_i^{\alpha} \to u'[v']_i^{\alpha'}$ *donde* $1 \leq i \leq q$, $\alpha, \alpha' \in \{0, +, -\}$ *y* $u, v, u', v' \in \Gamma^*$*:*

- *La parte izquierda de* r *es* $LHS(r) = (i, \alpha, u, v)$. *La carga de* $LHS(r)$ *es* $charge(LHS(r)) = \alpha$.

- *La parte derecha de r es $RHS(r) = (i, \alpha', u', v')$. La carga de $RHS(r)$ es $charge(RHS(r)) = \alpha'$.*

La carga de $LHS(r)$ es la segunda componente de la tupla (ídem para $RHS(r)$).

Definición 6.3. *Dados $x \in \Gamma$, $l \in H$, y $r \in \mathcal{R}$ tales que $LHS(r) = (i, \alpha, u, v)$, diremos que* (x, l) aparece *en $LHS(r)$ con multiplicidad k en cualquiera de los siguientes casos:*

- *$l = i$, y x aparece en el multiconjunto v con multiplicidad k.*
- *l es la etiqueta de la membrana padre de i, y x aparece en el multiconjunto u con multiplicidad k.*

Definición 6.4. *Las reglas de $\mathcal{R}$ y $\mathcal{R}_E$ pueden ser clasificadas en bloques como sigue: (a) el bloque asociado con (i, α, u, v) es $B_{i,\alpha,u,v} = \{r \in \mathcal{R} : LHS(r) = (i, \alpha, u, v)\}$; y (b) el bloque asociado con (e_j, x) es $B_{e_j,x} = \{r \in \mathcal{R}_E : LHS(r) = (e_j, x)\}$.*

Proporcionamos también a continuación la noción de consistencia de reglas y conjunto de reglas (Definiciones 6.5 y 6.6, respectivamente).

Definición 6.5. *Dos reglas, $r_1 \equiv u_1[v_1]_{i_1}^{\alpha_1} \to u_1'[v_1']_{i_1}^{\alpha_1'}$ y $r_2 \equiv u_2[v_2]_{i_2}^{\alpha_2} \to u_2'[v_2']_{i_2}^{\alpha_2'}$, son consistentes si y solo si $(i_1 = i_2 \wedge \alpha_1 = \alpha_2 \to \alpha_1' = \alpha_2')$*

Definición 6.6. *Un conjunto de reglas es consistente si todo par de reglas del conjunto es consistente.*

Después de la introducción de estas nociones, mostramos el pseudocódigo del algoritmo BBB (Algoritmo 6.2.1). Se divide en dos fases principales (como es común en otros modelos): *Selección* (Algoritmo 6.2.2) y *Ejecución* (Algoritmo 6.2.3). En el primero se seleccionan las reglas determinando el número de veces que se aplica cada una (y consumiendo los objetos de la parte izquierda), y en la segunda se aplican de forma efectiva (generando los objetos y cargas correspondientes). La selección de reglas se realiza de manera que los bloques son considerados en un orden aleatorio. Una vez que un bloque se selecciona, las reglas que lo compone son seleccionadas de forma maximal, pero de acuerdo a una distribución multinomial. Esta distribución se implementa utilizando binomiales y normalización en cada regla y según las restantes. Por último, se eliminan todas las copias de la parte izquierda en la configuración para que los siguientes bloques a ser seleccionados actúen de forma consecuente.

Algoritmo 6.2.1 PROCEDIMIENTO PRINCIPAL BBB

Entrada: Un sistema PDP de grado (q, m) con $q, m \geq 1$, y $T \geq 1$.

1: **para** $t \leftarrow 0$ a $T-1$ **hacer**
2: $R_{sel} \leftarrow$ FASE-SELECCIÓN (C_t, R) ▷ (Algoritmo 6.2.2)
3: $C_{t+1} \leftarrow$ FASE-EJECUCIÓN (C_t, R_{sel}) ▷ (Algoritmo 6.2.3)
4: **fin para**

Algoritmo 6.2.2 FASE-SELECCIÓN BBB

1: Las reglas de $R = \mathcal{R}_E \cup \mathcal{R}_{\Pi_j}, 1 \leq j \leq m$ son agrupadas en bloques B_l según la Definición 6.4.
2: Sea $F_b(N, p)$ una función que retorna un número natural aleatorio obtenido según la distribución binomial $B(N, p)$.
3: Se considera un orden aleatorio en la familia de todos los bloques de reglas.
4: **para cada** bloque de reglas $B_l = \{r_1, \ldots, r_s\}$, de acuerdo al orden aleatorio **hacer**
5: Se elige un orden aleatorio sobre las reglas $\{r_1, \ldots, r_s\}$.
6: $(h, \alpha, u, v) \leftarrow l$ y $c_{r_1}, \ldots, c_{r_s}$ son las probabilidades constantes de $\{r_1, \ldots, r_s\}$.
7: $N \leftarrow max\{n : r_1^n$ es aplicable en la configuración $C_t\}$.
8: **si** $N > 0$ **entonces**
9: $d \leftarrow 1$
10: **para** $k \leftarrow 1$ a $s-1$, de acuerdo al orden aleatorio **hacer**
11: $c_{r_k} \leftarrow \frac{c_{r_k}}{d}$
12: $n_{r_k} \leftarrow F(N, c_{r_k})$
13: $N \leftarrow N - n_{r_k}$
14: $q \leftarrow 1 - c_{r_k}$
15: $d \leftarrow d * q$
16: **fin para**
17: $n_{r_s} \leftarrow N$
18: **para** $k \leftarrow 1$ a s **hacer**
19: $C_t \leftarrow C_t - n_{r_k} * LHS(r_k)$
20: $R_{sel} \leftarrow R_{sel} \cup \langle r_k, n_{r_k} \rangle$
21: **fin para**
22: **fin si**
23: **fin para**
24: **devolver** R_{sel}

Algoritmo 6.2.3 FASE-EJECUCIÓN BBB

1: **para cada** $\langle r, n \rangle \in R_{sel}$ **hacer**
2: **si** $n > 0$ **entonces**
3: $C_t \leftarrow C_t + n * RHS(r)$
4: Actualizar las cargas de las membranas de C_t usando $RHS(r)$
5: **fin si**
6: **fin para**
7: **devolver** C_t

Este algoritmo de simulación ha conseguido reproducir la dinámica de muchos modelos de forma satisfactoria, como [22, 21]. Sin embargo, tiene una serie de desventajas que restringe los modelos de sistemas PDP que pueden ser correctamente simulados: no maneja bien las reglas con intersecciones en las partes izquierdas (competición de objetos), no maneja la consistencia de cargas, y no soporta, de manera explícita, la evaluación de funciones probabilísticas. Estas carencias dieron lugar al desarrollo de nuevos algoritmos de simulación que mejoran la flexibilidad de los simuladores, y que ofrecen características suplementarias a los nuevos requisitos de los modelos actuales.

6.3. Direct Non-Deterministic distribution with Probabilities (DNDP)

En esta sección describimos el primero de los dos algoritmos propuestos en este trabajo para la simulación de sistemas PDP. Se denomina DNDP [65, 66], el cual está inspirado en el algoritmo DND [76] con extensión para probabilidades. La entrada es un sistema PDP de grado (q, m) tomando T unidades de tiempo. El pseudocódigo de la rutina principal del algoritmo DNDP se muestra en el Algoritmo 6.3.1.

De forma similar al algoritmo BBB (y otros modelos como membranas activas), la rutina principal de la simulación consta de un bucle dividido en dos fases: selección y ejecución. Previamente a estas dos fases, se ejecuta una rutina de inicialización de estructuras de datos (Algoritmo 6.3.2). En este caso, la selección se diferencia en que se compone de dos micro fases: primera fase de selección (Algoritmo 6.3.3) y segunda fase de selección (Algoritmo 6.3.4). La primera fase de selección realiza un recorrido de las reglas (sin considerar bloques) de forma aleatoria. Cada vez que se elige una regla, se selecciona un número aleatorio de veces según la distribución binomial con el número de

Algoritmo 6.3.1 PROCEDIMIENTO PRINCIPAL DNDP

Entrada: Un sistema PDP de grado (q, m) con $q \geq 1,\ m \geq 1$, tomando T unidades de tiempo, $T \geq 1$.

1: $C_0 \leftarrow$ la configuración inicial del sistema.
2: **para** $t \leftarrow 0$ to $T - 1$ **hacer**
3: $C'_t \leftarrow C_t$
4: *INICIALIZACIÓN* ▷ (Algoritmo 6.3.2)
5: *PRIMERA FASE DE SELECCIÓN*: consistencia ▷ (Algoritmo 6.3.3)
6: *SEGUNDA FASE DE SELECCIÓN*: maximalidad ▷ (Algoritmo 6.3.4)
7: *EJECUCIÓN* ▷ (Algoritmo 6.3.5)
8: $C_{t+1} \leftarrow C'_t$
9: **fin para**

aplicaciones posibles, y la probabilidad asociada. Las reglas son elegidas según un orden aleatorio, y cada una se selecciona si su aplicación es consistente con la selección de reglas actual. Al final de la fase primera se devuelve un multiconjunto consistente de reglas aplicables. La segunda fase de selección recorre el conjunto de reglas seleccionadas, una a una, comprobando que no queden más aplicaciones posibles. Al final de la fase se devuelve un multiconjunto consistente y maximal de reglas aplicables. Finalmente, y como ha sido común en los algoritmos anteriores, la ejecución crea los objetos de las partes izquierdas y genera las cargas a las membranas correspondientes.

Algoritmo 6.3.2 INICIALIZACIÓN DNDP

1: **para** $j \leftarrow 1$ a m **hacer**
2: $R_{E,j} \leftarrow$ conjunto ordenado de reglas de $\mathcal{R}_E$ asociadas con el entorno j
3: $A_j \leftarrow$ conjunto ordenado de reglas de $R_{E,j}$ con probabilidad > 0 en paso t
4: $LC_j \leftarrow$ conjunto ordenado de pares $\langle label, charge \rangle$ para todas las membranas de C_t contenidas en el entorno j
5: $B_j \leftarrow \emptyset$
6: **para cada** $\langle h, \alpha \rangle \in LC_j$ (siguiendo el orden establecido) **hacer**
7: $B_j \leftarrow B_j \cup$ conjunto ordenado de reglas de R_{Π_j} cuya probabilidad es > 0 en el paso t para el entorno j
8: **fin para**
9: **fin para**

Algoritmo 6.3.3 PRIMERA FASE DE SELECCIÓN DNDP: CONSISTENCIA

para $j \leftarrow 1$ a m **hacer**
  $R_j \leftarrow \emptyset$
  $D_j \leftarrow A_j \cup B_j$ con un *orden aleatorio*
  para cada $r \in D_j$ (siguiendo el orden considerado) **hacer**
    $M \leftarrow$ máximo número de veces que r es aplicable a C'_t
    si r es *consistente* con las reglas en $R_j^1 \wedge M > 0$ **entonces**
      $N \leftarrow$ máximo numero de veces que r es aplicable a C_t
      $n \leftarrow \min\{M, F_b(N, p_{r,j}(t))\}$
      $C'_t \leftarrow C'_t - n \cdot LHS(r)$
      $R_j \leftarrow R_j \cup \{< r, n >\}$
    fin si
  fin para
fin para

Algoritmo 6.3.4 SEGUNDA FASE DE SELECCIÓN: MAXIMALIDAD

para $j \leftarrow 1$ a m **hacer**
  $R_j \leftarrow R_j$ *con un orden según la probabilidad de reglas, de mayor a menor*
  para each $< r, n > \in R_j$ (siguiendo el orden seleccionado) **hacer**
    si $n > 0 \vee (r$ es *consistente* con las reglas en $R_j^1)$ **entonces**
      $M \leftarrow$ máximo número de veces que r es aplicable a C'_t
      si $M > 0$ **entonces**
        $R_j \leftarrow R_j \cup \{< r, M >\}$
        $C'_t \leftarrow C'_t - M \cdot LHS(r)$
      fin si
    fin si
  fin para
fin para

Algoritmo 6.3.5 EJECUCIÓN DNDP

para cada $< r, n > \in R_j$, $n > 0$ **hacer**
  $C'_t \leftarrow C'_t + n \cdot RHS(r)$
  Actualizar la carga eléctrica de C'_t de acuerdo a $RHS(r)$
fin para

6.4. Direct distribution based on Consistent Blocks Algorithm (DCBA)

Los algoritmos introducidos hasta ahora (BBB y DNDP) comparten un inconveniente, relativo a una distorsión en la forma en que los bloques y reglas son seleccionados. En vez de seleccionar reglas y bloques según sus probabilidades de forma uniforme, este proceso de selección sesga aquellas con menor probabilidad. Esta sección introduce un algoritmo reciente, conocido como Direct distribution based on Consistent Blocks Algorithm (*DCBA*) [64, 63]. Este algoritmo resuelve el inconveniente mencionado, realizando una distribución uniforme de los recursos a aquellos bloques que compiten por ellos (con intersección en las partes izquierdas), y haciendo uso de un nuevo concepto denominado *bloque consistente*.

6.4.1. Definiciones de bloques y consistencia

El mecanismo de selección del DCBA asume que las reglas en $\mathcal{R}$ y $\mathcal{R}_E$ pueden ser clasificadas en bloques según las Definiciones 6.2 y 6.4. Sin embargo, este nuevo algoritmo utilizará un concepto ligeramente diferente de bloque, el cual considera la consistencia de reglas mediante la incorporación de la carga de la parte derecha.

Definición 6.7. *Dado un* (i, α, u, v) *donde* $1 \leq i \leq q$, $\alpha \in EC$, $u, v \in \Gamma^*$, *el bloque* $B_{i,\alpha,u,v}$ *es consistente si y solo si existe* α' *tal que, para cada* $r \in B_{i,\alpha,u,v}$, $charge(RHS(r)) = \alpha'$.

Definición 6.8. *Dado* i, α, α', u, v *donde* $1 \leq i \leq q$, $\alpha, \alpha' \in EC$, $u, v \in \Gamma^*$, *el bloque asociado con* $(i, \alpha, \alpha', u, v)$ *es el conjunto:*

$$B_{i,\alpha,\alpha',u,v} = \{r \in \mathcal{R} : LHS(r) = (i, \alpha, u, v) \wedge charge(RHS(r)) = \alpha'\}$$

Podemos denotar, por tanto, que la parte izquierda de un bloque B, denotado por $LHS(B)$, está definida como la parte izquierda de cualquier regla del bloque.

Definición 6.9. *Diremos que dos bloques* $B_{i_1,\alpha_1,\alpha'_1,u_1,v_1}$ *y* $B_{i_2,\alpha_2,\alpha'_2,u_2,v_2}$ *son mutuamente consistentes si y solo si* $(i_1 = i_2 \wedge \alpha_1 = \alpha_2) \Rightarrow (\alpha'_1 = \alpha'_2)$.

Definición 6.10. *Un conjunto de bloques* $\mathcal{B} = \{B^1, B^2, \ldots, B^s\}$ *es mutuamente consistente si y solo si* $\forall i, j$ (B^i *y* B^j *son mutuamente consistentes*).

6.4.2. Pseudocódigo del DCBA

El Algoritmo 6.4.1 describe el bucle principal del DCBA. Como en otros casos, se divide en dos fases: selección y ejecución. Antes de pasar a simular los pasos de computación, se construye una *tabla estática* ($\mathcal{T}$), en la cual se disponen los bloques en las columnas, y los objetos del alfabeto asociado a cada región del sistema en las filas. La tabla asocia bloques y objetos por región según las partes izquierdas. Esta información se complementa con los bloques asociados a las reglas de comunicación entre entornos.

Algoritmo 6.4.1 PROCEDIMIENTO PRINCIPAL DCBA

Entrada: Un sistema PDP de grado (q, m), $T \geq 1$, y $A \geq 1$ (*Precisión*). La configuración inicial se denomina C_0.

1: *INICIALIZACION* ▷ (Algoritmo 6.4.2)
2: **para** $t \leftarrow 1$ **a** T **hacer**
3: Calcular funciones de probabilidad $f_{r,j}(t)$ y $p(t)$.
4: $C'_t \leftarrow C_{t-1}$
5: *SELECCIÓN* de reglas:
- *FASE 1*: distribución ▷ (Algoritmo 6.4.3)
- *FASE 2*: maximalidad ▷ (Algoritmo 6.4.4)
- *FASE 3*: probabilidades ▷ (Algoritmo 6.4.5)

6: *EJECUCIÓN* de reglas. ▷ (Algoritmo 6.4.6)
7: $C_t \leftarrow C'_t$
8: **fin para**

Observación 6.1. *Cada columna de las tablas $\mathcal{T}_j$ contiene la información correspondiente a la parte izquierda del bloque.*

Observación 6.2. *Cada fila de las tablas $\mathcal{T}_j$ contiene la información relacionada con la competición por objetos: dado un objeto, la fila indica cuales son los bloques que lo requieren.*

Otra diferencia sustancial es la división de la fase de selección en tres micro fases: fase 1 de distribución (donde se calculan las aplicaciones de los bloques mediante el uso de la tabla de distribución), fase 2 de maximalidad (donde se asegura una aplicación maximal de los bloques seleccionados) y fase 3 de probabilidad (donde se traduce la selección de bloques a selección de reglas mediante el uso de números aleatorios con distribución normal). Refiérase a [64] para una explicación mas detallada con ejemplos de como aplicar el algoritmo.

Algoritmo 6.4.2 INICIALIZACIÓN

1: Construcción de la tabla de *distribución estática* $\mathcal{T}$:

- Etiquetas de columna: bloques $B_{i,\alpha,\alpha',u,v}$ de $\mathcal{R}$.
- Etiquetas de fila: pares (x, i), para todo objeto $x \in \Gamma$, y $0 \leq i \leq q$.
- En cada celda de la tabla poner $\frac{1}{k}$ si el objeto de la fila aparece en la LHS del bloque de la columna con multiplicidad k.

2: **para** $j = 1$ **a** m **hacer** ▷ (Construir las tablas *estáticas expandidas* $\mathcal{T}_j$)
3: $\mathcal{T}_j \leftarrow \mathcal{T}$. ▷ (Inicializar la tabla con la original $\mathcal{T}$)
4: Añadir una fila para cada bloque $B_{e_j,x}$ de $\mathcal{R}_E$, con el valor 1 en la fila $(x, 0)$.
5: Inicializar los multiconjuntos $B^j_{sel} \leftarrow \emptyset$ y $R^j_{sel} \leftarrow \emptyset$
6: **fin para**

La fase de selección 1 utiliza tres procedimientos de filtros auxiliares. El Filtro 1 descarta las columnas de la tabla correspondientes a bloques no aplicables por discrepancia de cargas en la parte izquierda y la configuración C'_t. El Filtro 2 descarta las columnas con objetos en la parte izquierda que no aparecen en C'_t. Finalmente, a fin de ahorrar espacio en la tabla, el Filtro 3 descarta filas vacías. Estos tres filtros son aplicados al comienzo de la Fase 1, y el resultado es una *tabla dinámica* $\mathcal{T}^t_j$.

Para más información y detalles de los algoritmos introducidos, refiérase a los artículos correspondientes. El algoritmo DNDP se describe en [65], en donde se realiza una comparativa de comportamiento y rendimiento con el algoritmo BBB, utilizando pLinguaCore y una implementación paralela en Java. De manera adicional, en [66] se presentó una verificación formal del algoritmo DNDP, demostrando la corrección del objetivo de cada módulo (o fase) que forma el algoritmo. En [64, 63] se presentó el algoritmo DCBA. En el mismo se incluyó una validación experimental, tanto del algoritmo DNDP como del DCBA, con el modelo de un ecosistema real relacionado con el quebrantahuesos en el Pirineo Catalán.

Algoritmo 6.4.3 FASE DE SELECCIÓN 1: DISTRIBUCIÓN

para $j = 1$ **a** m **hacer** ▷ (Por cada entorno e_j)
  Aplicar filtros a la tabla $\mathcal{T}_j$, usando C'_t y obteniendo $\mathcal{T}_j^t$:
    a. $\mathcal{T}_j^t \leftarrow \mathcal{T}_j$
    b. FILTRO 1 $(\mathcal{T}_j^t, C'_t)$.
    c. FILTRO 2 $(\mathcal{T}_j^t, C'_t)$.
    d. Comprobar la *mutua consistencia* para los bloques restantes en $\mathcal{T}_j^t$. **Si** existe al menos una inconsistencia **entonces** reportar el error, y acabar la ejecución (en caso de no activar el paso 3).
    e. FILTRO 3 $(\mathcal{T}_j^t, C'_t)$.
  (OPCIONAL) Generar un conjunto S_j^t de sub-tablas de $\mathcal{T}_j^t$, formado por los conjuntos de bloques *mutuamente consistentes*, de forma maximal en $\mathcal{T}_j^t$. Reemplazar $\mathcal{T}_j^t$ con una tabla elegida aleatoriamente de S_j^t.
  $a \leftarrow 1$
  repeat
    para cada fila X en $\mathcal{T}_j^t$ **hacer**
      $RowSum_{X,t,j} \leftarrow$ suma total de valores no nulos en la fila X.
    fin para
    $\mathcal{TV}_j^t \leftarrow \mathcal{T}_j^t$ ▷ (Una copia temporal)
    para cada posición no nula (X, Y) en $\mathcal{T}_j^t$ **hacer**
      $mult_{X,t,j} \leftarrow$ multiplicidad en C'_t y e_j del objeto en fila X.
      $\mathcal{TV}_j^t(X, Y) \leftarrow \lfloor mult_{X,t,j} \cdot \frac{(\mathcal{T}_j^t(X,Y))^2}{RowSum_{X,t,j}} \rfloor$
    fin para
    para cada columna no filtrada, etiquetada por el bloque B, en $\mathcal{T}_j^t$ **hacer**
      $N_B^a \leftarrow \min_{X \in rows(\mathcal{T}_j^t)}(\mathcal{TV}_j^t(X, B))$ ▷ (El mínimo de la columna)
      $B_{sel}^j \leftarrow B_{sel}^j + \{B^{N_B^a}\}$ ▷ (Acumular el valor al total)
      $C'_t \leftarrow C'_t - LHS(B) \cdot N_B^a$ ▷ (Eliminar el LHS del bloque)
    fin para
    FILTRO 2 $(\mathcal{T}_j^t, C'_t)$
    FILTRO 3 $(\mathcal{T}_j^t, C'_t)$
    $a \leftarrow a + 1$
  until $(a > A) \vee$ *(Todos los mínimos del paso 15 son* 0*)*
fin para

Algoritmo 6.4.4 FASE DE SELECCIÓN 2: MAXIMALIDAD

para $j = 1$ **a** m **hacer** ▷ (Para cada entorno e_j)
  Elegir un orden aleatorio de los bloques restantes en la tabla $\mathcal{T}_j^t$.
  para cada bloque B, siguiendo el orden impuesto **hacer**
    $N_B \leftarrow$ número de aplicaciones posibles de B en C'_t.
    $B^j_{sel} \leftarrow B^j_{sel} + \{B^{N_B}\}$ ▷ (Acumular el valor al total)
    $C'_t \leftarrow C'_t - LHS(B) \cdot N_B$ ▷ (Eliminar la LHS del bloque B, N_B veces.)
  fin para
fin para

Algoritmo 6.4.5 FASE DE SELECCIÓN 3: PROBABILIDAD

para $j = 1$ **a** m **hacer** ▷ (Para cada entorno e_j)
  para cada bloque $B^{N_B} \in B^j_{sel}$ **hacer**
    Calcular $\{n_1, \ldots, n_l\}$, con una multinomial $M(N_B, g_1, \ldots, g_l)$ según las probabilidades de las reglas $r_1, \ldots, r_l$ del bloque.
    para $k = 1$ **a** l **hacer**
      $R^j_{sel} \leftarrow R^j_{sel} + \{r_k^{n_k}\}$.
    fin para
  fin para
  $B^j_{sel} \leftarrow \emptyset$. ▷ (Útil para la siguiente iteración)
fin para

Algoritmo 6.4.6 EJECUCIÓN

para $j = 1$ **a** m **hacer** ▷ (Para cada entorno e_j)
  para cada regla $r^n \in R^j_{sel}$ **hacer** ▷ (Aplicar las RHS de las reglas)
    $C'_t \leftarrow C'_t + n \cdot RHS(r)$
    Actualizar las cargas eléctricas de C'_t con $RHS(r)$.
  fin para
  $R^j_{sel} \leftarrow \emptyset$. ▷ (Útil para la siguiente iteración)
fin para

"El arte de la conjetura, o arte estocástico, se define como el arte de evaluar con la mayor exactitud posible las probabilidades de las cosas, de modo que en nuestros juicios y acciones siempre podemos basarnos en lo que se ha encontrado para ser lo mejor, lo más adecuado, lo más seguro, lo más informado. Este es el único objetivo de la sabiduría del filósofo y la prudencia del estadista."

Jacob Bernoulli

7

Simulación Paralela de Sistemas PDP

Uno de los objetivos principales de los sistemas PDP es ayudar a los ecólogos a adoptar estrategias *a priori* en la administración de ecosistemas reales. Esto se logra mediante la ejecución de experimentos virtuales. Este hecho hace que el desarrollo de simuladores y herramientas informáticas se convierta en un punto crítico en el proceso de validación y experimentación virtual.

El marco de modelización de sistemas PDP está soportado por una herramienta informática denominada MeCoSim [84], la cual ofrece una interfaz gráfica, definición de modelos y parámetros, ejecución de simulaciones, recolección de datos estadísticos, etc. El núcleo de la aplicación es pLinguaCore [46], el cual hace uso de algoritmos de simulación para reproducir la ejecución de los modelos. Existen varios algoritmos implementados dentro de pLinguaCore: BBB (en versión secuencial), DNDP (en versión secuencial y paralela con hilos de la máquina virtual de Java) y DCBA (solo secuencial).

En este capítulo resumiremos las implementaciones desarrolladas en pLinguaCore para los algoritmos DNDP y DCBA. Además, introduciremos un conjunto de implementaciones alternativas para el DCBA en C++ [56], OpenMP [10] y CUDA [5]. Esta alternativa se encuentra dentro de una herramienta de simulación independiente, que se conectará con pLinguaCore en un futuro.

Los simuladores (versiones C++, OpenMP, y CUDA) están incluidos en el marco de simulación denominado *ABCD-GPU*, pudiéndose descargar de la web del proyecto PMCGPU: `http://sourceforge.net/p/pmcgpu` [12].

7.1. Implementación del algoritmo DNDP en pLinguaCore

En esta sección resumiremos la implementación del algoritmo DNDP llevada a cabo en la librería pLinguaCore. Para más detalle y ver el pseudocódigo de la implementación, nos referiremos al artículo de introducción al DNDP [65].

El aspecto clave de esta implementación es la solución paralela, para la cual se considera que las partes izquierdas de las reglas solo afectan a un único entorno a la vez. Por tanto, la selección de reglas se puede ejecutar en paralelo para cada entorno sin comprometer la concurrencia del sistema cuando se accede a objetos comunes. Sin embargo, la selección y ejecución de reglas debe de ser sincronizada a nivel del sistema, de tal forma que la fase de ejecución no puede comenzar antes de que se haga la selección de reglas en todos los entornos.

En la implementación dentro de pLinguaCore, se asigna un hilo de ejecución a cada entorno, en el cual se calcula una selección de reglas aplicando las dos fases de selección del algoritmo DNDP. El rendimiento de esta implementación se ha analizado usando unos sistemas PDP de prueba que constan de 16 entornos y un alto número de reglas (4000) [65], y usando Java Virtual Machine 1.6.0 en nuestro servidor de computación. En las pruebas se ejecutaron tanto la implementación del BBB, DNDP secuencial y DNDP paralelo (con 4 hilos sobre los 4 procesadores del servidor). Se comprobó que el simulador DNDP secuencial es ligeramente más rápido que el BBB (hasta un 1.07x de speedup), y el simulador DNDP paralelo también es ligeramente superior al DNDP secuencial (1.72x).

Podemos concluir que la Máquina Virtual de Java no es lo suficientemente eficiente para ejecutar el algoritmo DNDP. La aceleración conseguida no es tanta como la esperada cuando se simulan sistemas con 16 entornos en un procesador con 4 núcleos. Aunque siga siendo lo suficientemente potente para simular muchos de los modelos basados en sistemas PDP, los nuevos modelos bajo desarrollo requieren una gran cantidad de recursos, mucho mayor que los recursos en un ordenador personal usando Java. Por esta razón, usaremos nuestras implementaciones Java en pLinguaCore para procesos de validación, pero a fin de mejorar el rendimiento, construiremos implementaciones basadas en C++, empleando librerías que nos permitan manejar paralelismo real en plataformas paralelas.

7.2. Implementación de DCBA en pLinguaCore

PLinguaCore también se ha extendido para implementar el algoritmo DCBA. Esta ampliación, junto a otras mejoras y corrección de errores, dio lugar a la versión *pLinguaCore 3.0*, que puede ser descargada de la página web oficial de P-Lingua [11]. Esta implementación en Java del DCBA tiene como objetivo servir como un marco de validación de los modelos y algoritmos. Por tanto, no consideraremos el rendimiento de esta versión, dejándolo para las versiones en C++ y CUDA.

El mayor reto en la implementación del algoritmo DCBA es la representación de la tabla de distribución asociada a la fase 1 (para más información, ver [64]). Esta tabla puede ser dispersa para muchos sistemas (las reglas solo requieren una pequeña cantidad de objetos de unas ciertas regiones, por lo que las columnas están mayormente vacías). Por ello, la representación completa de dicha tabla puede acarrear un problema de memoria. En esta implementación se hace uso de una tabla de dispersión, en la cual las claves son pares de objetos columna y fila, siendo columna la correspondiente parte izquierda del bloque, y la fila el par objeto y membrana.

Este simulador devuelve toda la información de las tablas, y entradas y salidas de cada fase del DCBA. Por tanto, sirve de apoyo a la validación del algoritmo y los modelos.

7.3. Implementación de DCBA en C++

Como se acaba de mencionar, el algoritmo DCBA fue implementado primero dentro de pLinguaCore. Esta versión (la denominaremos *pdp-plcore-sim*) se validó con un modelo de ecosistema real [64]. Sin embargo, el rendimiento fue bajo por estar basado en Java. Así, la primera aproximación para mejorar la eficiencia fue hacer un simulador independiente en C++ [68] (llamémosle *pdp-seq-sim*). Elegimos C++ por su similitud con lenguajes para GPGPU (OpenCL y CUDA), y el soporte de varias librerías paralelas como OpenMP, PThreads y MPI.

El diseño, como con *pdp-plcore-sim*, está orientado a mejorar el rendimiento de la fase 1 (la cual se ha demostrado, tras varios experimentos, ser la fase que consume más tiempo y memoria, y por tanto la más prioritaria en cuanto a mejorar de eficiencia, según la Ley de Amdhal). El mayor reto es implementar la tabla de distribución estática $\mathcal{T}_j$. El tamaño de dicha tabla es $O(|B|\cdot|\Gamma|\cdot(q+1))$,

siendo $|B|$ el número de bloques de reglas, $|\Gamma|$ el tamaño del alfabeto, y $q+1$ el número de membranas más el entorno.

En esta implementación se ha diseñado un algoritmo que evita la creación de dicha tabla, traduciendo las operaciones sobre la tabla a operaciones directamente sobre la información de los bloques de reglas (usando las observaciones hechas en el Capítulo 6): las operaciones sobre columnas se corresponden con operaciones sobre bloques de reglas y sus partes izquierdas, y las operaciones sobre filas se corresponden con recorrer las partes izquierdas de los bloques pero almacenando los resultados en un array global, con una posición por fila.

Esta implementación se denomina de *tabla virtual* (ver Algoritmo 7.3.1), y utiliza algunas estructuras de datos auxiliares para simular la tabla de distribución (no implementada en sí): $activationVector$ (indica los bloques que están filtrados), $MinN$ (almacena los mínimos de cada columna), $BlockSel$ (almacena las selecciones de bloques), y $RuleSel$ (almacena las selecciones de reglas).

7.4. Implementación paralela de DCBA en plataformas multinúcleo con OpenMP

En esta sección resumiremos la primera aproximación de implementación paralela del algoritmo DCBA, empleando OpenMP para el uso de procesadores multinúcleo [68]. Desglosaremos las tres alternativas y la prueba de rendimiento entre dos generaciones de procesadores Intel. A este nuevo simulador lo denominaremos *pdp-omp-sim*.

De este análisis identificaremos nuevas formas de mejorar el tiempo de ejecución del simulador, minimizando los cuellos de botella de memoria y caché empleando compresión de datos y computación con GPU en un futuro. Ideas y referencias en la compresión y computación con GPU se puede encontrar en [15].

7.4.1. Diseño paralelo

Esta nueva implementación se ha paralelizado de tres formas: 1) simulaciones, 2) entornos e 3) híbrido:

- Las *simulaciones* que se requieran ejecutar se paralelizan con la directiva de OpenMP `#pragma omp parallel for` [10] en el bucle de simulación. Este bucle exterior para simulaciones se puede añadir fácilmente en el

Algoritmo 7.3.1 Implementación de la fase de selección 1 con tabla virtual

para $j = 1, \ldots, m$ **hacer** ▷ Por cada entorno
  para cada bloque B **hacer**
    $activationVector[B] \leftarrow true$
    si $charge(LHS(B))$ es diferente al presente en C'_t **entonces**
      $activationVector[B] \leftarrow false$ ▷ (Aplicar FILTRO 1)
    else si uno de los objetos en $LHS(B)$ no existe en C'_t **entonces**
      $activationVector[B] \leftarrow false$ ▷ (Aplicar FILTRO 2)
    fin si
  fin para
  Comprobar la mutua consistencia de bloques.
  repeat
    para cada bloque B con $activationVector[B] = true$ **hacer** ▷ (Suma)
      para cada objeto o^k en $LHS(B)$, asociado con la región i **hacer**
        $addition[o, i] \leftarrow addition[o, i] + \frac{1}{k}$
      fin para
    fin para
    para cada bloque B con $activationVector[B] = true$ **hacer** ▷ (Min.)
      $MinN[B] \leftarrow Min_{[o^k]_i \in LHS(B)}(\frac{1}{k^2} * \frac{1}{addition[o,i]} * C'_t[o, i])$.
      $BlockSel[B] \leftarrow BlockSel[B] + MinN[B]$.
    fin para
    para cada bloque B con $activationVector[B] = true$ **hacer** ▷ (Act.)
      $C'_t \leftarrow C'_t - LHS(B) * MinN[B]$
    fin para
    Aplicar FILTRO 2 (como en el paso 6).
    $a \leftarrow a + 1$
  until $a = A$ **o** *todo bloque activo* B *cumple* $MinN[B] = 0$
fin para

procedimiento principal del DCBA. La ventaja de ejecutar simulaciones en paralelo es que no existe dependencias de datos entre ellas, y por tanto, el problema es totalmente paralelo. Además, los usuarios normalmente ejecutan de 50 a 100 simulaciones por cada conjunto de parámetros de entrada, por lo que existen suficientes simulaciones para usar todos los núcleos de un procesador. Sin embargo, entre las desventajas destacamos el hecho de que cada simulación requerirá su propio espacio de memoria, incrementando el uso de memoria total. Además, puede haber problemas de balanceo de carga y conflicto de recursos entre los núcleos.

- Los *entornos* son paralelizados empleando la directiva `#pragma omp parallel` [10] para generar un conjunto de hilos para la simulación. Entonces los bucles **for** en el Algoritmo 7.3.1 que iteran sobre los entornos son paralelizados con `#pragma omp for` [10], los cuales tienen una barrera implícita que fuerza las dependencias entre las distintas fases. La ventaja de paralelizar entornos sobre simulaciones es que el uso de memoria no incrementa. Sin embargo, las dependencias ocurren dos veces en cada paso, requiriendo sincronización. Además, ya que muchos modelos usan desde 5 a 30 entornos, habrán máquinas modernas que tengan más núcleos que entornos disponibles en el sistema.

- La paralelización *híbrida* se acomete combinando entornos y simulaciones. Esto se lleva a cabo mediante parámetros por línea de comando, que periten especificar cuantos entornos y simulaciones se van a ejecutar en paralelo. Esto permite balancear la cantidad de recursos existentes, como por ejemplo, incrementar el número de simulaciones hasta una cierta cantidad de memoria disponible, y a partir de ahí, los entornos.

7.4.2. Evaluación experimental

Los experimentos para evaluar el rendimiento de los simuladores fueron ejecutados en dos máquinas, mostradas en la Tabla 7.1. Los sistemas de entrada fueron generados aleatoriamente (es decir, no se corresponden con modelos de la ecología), pero que contienen cantidades de datos similares a los ejemplos de la vida real. Los números de entornos y simulaciones se variaron (de un sistema PDP a otro) de 10 a 50. Los tiempos corresponden solamente a la parte paralelizable del código, ejecutándose entre los 1 y 8 núcleos (distribuidos en dos sockets) de la máquina Intel i5 y los 1 a 4 de la Intel i7.

El tiempo de ejecución secuencial se muestra en la Tabla 7.2. *Config.* es el coste de ejecutar la parte secuencial del código. Las otras dos columnas

Procesador	Velocidad	Velocidad Bus	Caché
i5 Nehalem (2x4)	2 Ghz	3x800 Mhz	2x4 MB
i7 Sandy Bridge (1x4)	3.4 Ghz	2x1333 Mhz	8 MB

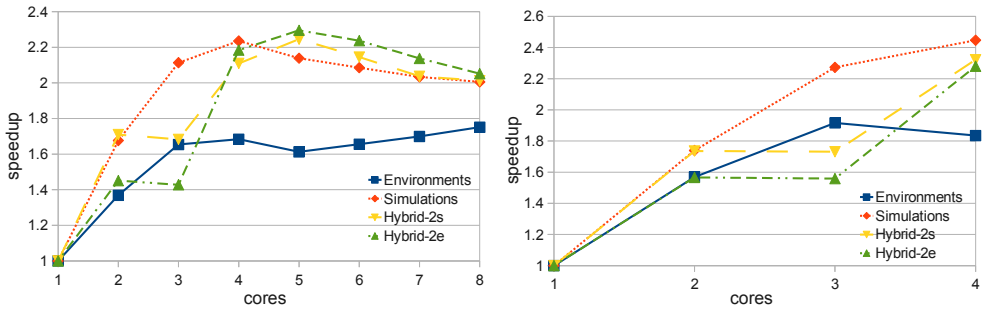
Tabla 7.1: Especificaciones de las máquinas.

Procesador	Config.	10 ent. & sim.	50 ent. & sim.
Nehalem	0.8 s	48.0 s	251.0 s
Sandy Bridge	0.35 s	19.9 s	97.8 s

Tabla 7.2: Tiempos de ejecución secuencial

representan los tiempos extremos de nuestros casos de test cuando se ejecutan en secuencial. Podemos observar que la parte de configuración es una parte pequeña del tiempo total, y el procesador Sandy Bridge es hasta 2.5 veces más rápido que el Nehalem.

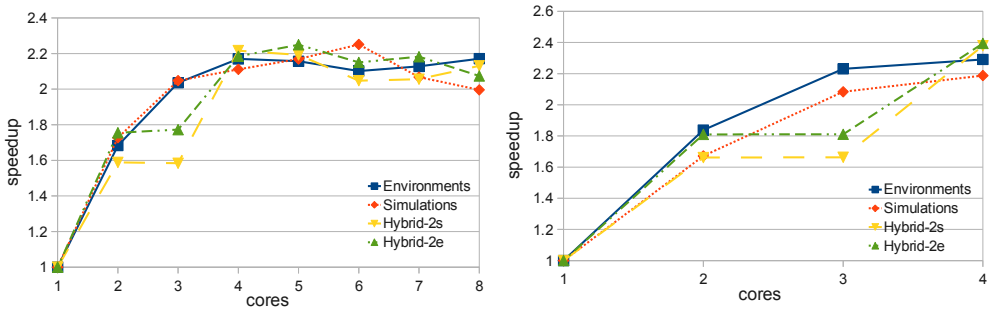
Las Figuras 7.1 y 7.2 muestran las mejoras de rendimiento, y son representativas de los otros test llevados a cabo con el mejor rendimiento siendo paralelizado por simulaciones o por el método híbrido ($2s$), que emplea dos simulaciones en paralelo y después distribuye los entornos. Como se puede observar, al incrementar el número de simulaciones, la ventaja de paralelizar por simulaciones también incrementa. Lo mismo se observa para entornos.



(a) Nehalem (b) Sandy Bridge

Figura 7.1: Aceleraciones ejecutando 50 simulaciones con 10 entornos

En el sistema Sandy Bridge, el máximo speedup es de 2.5x, y ocurre para 50 simulaciones y 50 entornos. Sin embargo, pasar de 3 a 4 núcleos solo incurre en 0.1 y 0.2, lo que sugiere que la computación está limitada por memoria. En el sistema Nehalem, el máximo speedup es de 2.3x, el cual apenas se supera usando el segundo socket de 4 núcleos, ya que no se cuenta con el subsistema de memoria NUMA de los dos sockets. Tras comprobar la mejora de tiempo empleando afinidad de hilos a núcleos concretos del sistema, los test



(a) Nehalem (b) Sandy Bridge

Figura 7.2: Aceleraciones ejecutando 10 simulaciones con 50 entornos

también indican que el código está limitado por memoria ya que la aceleración conseguida con 8 núcleos es menor que 5x.

En conclusión, los experimentos ejecutados muestran que las simulaciones están limitadas por memoria, y la proporción de código paralelizada consume sobre el 98 % del tiempo de ejecución. En este trabajo preliminar concluimos que paralelizar por simulaciones y técnicas híbridas da mejores resultados, además de usar nuevo hardware donde se incremente el ancho de banda de memoria.

7.5. Implementación paralela de DCBA en GPU con CUDA

El simulador previo *pdp-omp-sim* [68] es el punto de partida de la nueva implementación en CUDA. En este nuevo simulador (denominado *pdp-gpu-sim*) [69] y en *pdp-omp-sim*, se han mejorado tanto el código como las estructuras de datos, ahorrando un 27 % de memoria y obteniendo una ganancia de 1.25x de rendimiento en sistemas grandes.

A prima vista, los dos niveles de paralelismo identificados anteriormente (simulaciones y entornos) podrían encajar con el doble paralelismo de la arquitectura CUDA (bloques de hilos e hilos), por ejemplo asignando simulaciones a bloques de hilos, y entornos a los hilos (ya que estos se sincronizan fácilmente). Sin embargo, el número de entornos depende inherentemente del modelo (de 2 a 20 normalmente), lo que no es suficiente para rellenar un bloque de hilos. Por tanto, también paralelizaremos la ejecución de los bloques de reglas, pudiendo aprovechar de esta forma un gran número de hilos y distribuyendo simulaciones y entornos a lo largo de bloques de hilos. Ya que existen más bloques de reglas que hilos, crearemos del orden de 256 hilos que iteren

los bloques de reglas en tiles (porciones). Este diseño se muestra gráficamente en la Figura 7.3.

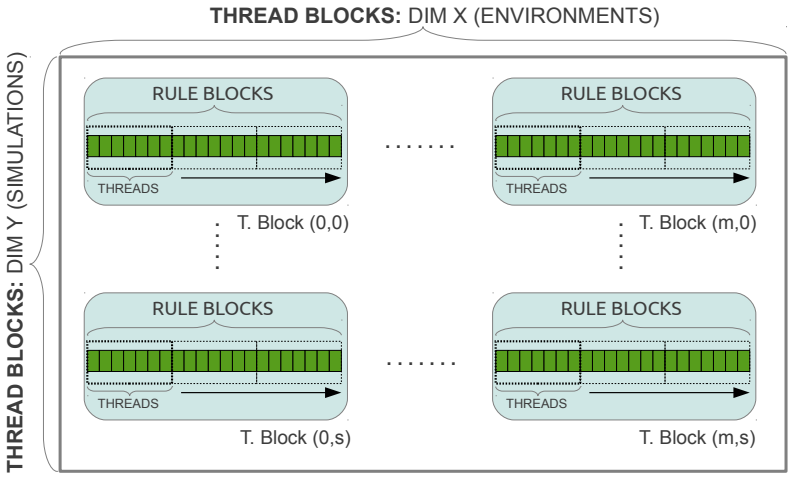


Figura 7.3: Diseño general del simulador basado en CUDA: un grid 2D y bloques de hilos unidimensionales. Los hilos iteran los bloques de reglas en tiles.

7.5.1. Implementación en GPU de las fases de DCBA

7.5.1.1. Implementación de la Fase 1 de selección

La implementación en el dispositivo se ha construido directamente desde el Algoritmo 7.3.1. La Fase 1 ha sido implementada empleando varios kernels, evitando la sobrecarga de uno solo:

- Kernel para Filtros (desde la línea 2 a 10 en el Algoritmo 7.3.1): los Filtros 1 y 2 se implementan aquí usando el diseño general en CUDA (Figura 7.3).

- Kernel para Normalización (desde la línea 11 a 20): las dos partes para sumar las filas y calcular los mínimos (denominado paso de normalización) se implementan aquí. Los dos pasos se sincronizan con la instrucción CUDA *syncthreads*. Mencionar que el trabajo asignado a los hilos es divergente (operación de scatter), esto es, cada hilo trabaja con un bloque de reglas, pero escribe la información por cada objeto que aparece en la parte izquierda. Por tanto, escribe en el vector *addition* con operaciones atómicas.

- Kernel para Actualizar y FILTRO 2 (desde la línea 21 a 26). Como antes, el trabajo de cada hilo es divergente. Por tanto, la actualización de la configuración también se implementa con operaciones atómicas.

También se ha tenido que tratar con una restricción en la GPU concerniente a las operaciones de punto flotante. Hemos encontrado que en varias GPUs de NVIDIA (especialmente, en nuestras GPUs con compute capability 1.3), la suma y la multiplicación son operaciones compatibles con el estándar IEEE 754, por lo que generan los mismos resultados que una CPU común. Sin embargo, muchas otras operaciones, como la división y raíz cuadrada, resultan en el valor de punto flotante más cercano al resultado matemático correcto (pero no son compatibles con el estándar) [100, 5]. Esto conlleva a pérdida de información, y a veces, a resultados incorrectos, en los cálculos normalizados de la Fase 1 (línea 14 en el Algoritmo 7.3.1). Inicialmente se usó valores de doble precisión para no perder precisión. Sin embargo, GPUs con compute capability 1.2 los degradaban a punto flotante. La solución adoptada fue la de usar dos valores enteros representando las facciones del vector de adición, por lo que la división real se realizaba solo al final del proceso acumulativo (línea 18 en el Algoritmo 7.3.1).

Finalmente, con el fin de evitar overflows, hemos implementado un bucle al comienzo que inicializa el vector *addition* al total de las sumas según la tabla estática. Después, ese vector se calcula substrayendo los valores de los bloques filtrados, en vez de sumar aquellos que están activos.

7.5.1.2. Implementación de la Fase 2 de Selección

La Fase 2 es en realidad un reto para ser paralelizada por bloques de reglas. La selección de bloques en esta fase se realiza de forma inherentemente secuencial: necesitamos conocer cuantos objetos necesita consumir un bloque antes de pasar a seleccionar el siguiente. La Fase 2 se implementa en un solo kernel siguiendo nuestro diseño general en CUDA.

El orden aleatorio de los bloques de reglas se *simula* con el planificador de hilos de CUDA (mediante la operación atómica *atomicInc*). Dado que no se realiza un orden aleatorio real, en las siguientes versiones se aplicarán números aleatorios. Nuestra primera versión (*ph2-simorder-oneseq*) fue la de lanzar 257 hilos, 256 que calculan el orden "aleatorio" y 1 que los itera.

Nuestra nueva versión (*ph2-simorder-dyncomp*) chequea dinámicamente los bloques que realmente compiten por objetos, y calcula los bloques que van a ser seleccionados en paralelo, y cuales dependen de la selección de otros. Esto requiere cálculos previos (ver [69] para más detalle). Se utilizan dos arrays que

se almacenan en memoria compartida para mejorar la eficiencia de la computación: uno indicará las competiciones (objetos de las partes izquierdas de los bloques, indicando cuales aparecen en otros bloques), y el otro indicará el orden de ejecución (si un bloque depende de otro, se le pondrá un orden de ejecución superior). El proceso es iterativo por cada bloque, pero los hilos comprueban en paralelo las correspondientes competiciones por objetos. Una vez conocidas las competiciones y las órdenes de ejecución, se pasa a seleccionar los bloques (directamente en memoria compartida, a la cual se copian las multiplicidades restantes en las configuraciones), y solo aquellos a los que les toque el turno.

Los experimentos muestran que *ph2-simorder-dyncomp* consigue un 20 % de mejora de rendimiento con respecto a *ph2-simorder-oneseq*.

7.5.1.3. Implementación de la Fase 3 de Selección

La Fase 3 calcula el número de veces que una regla es aplicada usando una distribución binomial y el número de selección del bloque. Esto se ha implementado en un solo kernel. Sin embargo, con el fin de generar números aleatorios binomiales, hemos tenido que desarrollar una librería CUDA basada en *cuRAND* [9], denominada *cuRNG_BINOMIAL*, ya que no existe tal implementación en ninguna librería según nuestro conocimiento. El módulo implementa el algoritmo BINV, propuesto por *Voratas Kachitvichyanukul* y *Bruce W. Schmeiser* [55] como el mejor para valores pequeños de n: BINV se ejecuta con una velocidad proporcional a $n \cdot p$. Se ha implementado con las mejoras propuestas en el mismo artículo [55]. Para valores mayores ($n \cdot p > 10$), se ha asumido una aproximación con la distribución normal, la cual está implementada en la librería cuRAND. La librería implementa una función de dispositivo *inline*, la cual ejecuta el código comentado en la GPU dentro del kernel (no se requiere generarlos previamente).

La implementación de la fase se traduce directamente del pseudocódigo del algoritmo DCBA. Además, es la que mejor explota el paralelismo, comparado con las otras fases del algoritmo.

7.5.1.4. Implementación de la Ejecución (Fase 4)

La Fase 4 se implementa directamente como se muestra en el pseudocódigo del DCBA usando nuestro diseño general de CUDA. En este caso, vamos a otro nivel de paralelismo para los hilos, y ahora trabajan independientemente con cada regla. Como antes, los hilos iteran las reglas en tiles, y añaden la correspondiente parte derecha (si tiene un número de aplicaciones mayor que 0).

Finalmente, dado que la operación es divergente (de reglas a añadir objetos), usamos operaciones atómicas para actualizar la configuración del sistema.

7.5.2. Resultados de rendimiento del simulador

Con el fin de analizar el rendimiento del simulador, hemos construido una serie de bancos de prueba mediante el generador de sistemas PDP aleatorio (denominado *pdps-rand*). Remarcar que estos sistemas generados no tienen significado biológico, teniendo como único propósito el estresar el simulador para analizar su rendimiento (ver [69] para más detalle).

Se han llevado a cabo dos bancos de pruebas: en el primero se analiza la escalabilidad del simulador incrementando el tamaño del sistema de tres formas, y en el segundo se realiza un *profile* mostrando los porcentajes de tiempos consumidos por cada fase. Todos los experimentos fueron ejecutados en nuestro servidor de GPU: Linux de 64 bits, con dos procesadores (4 núcleos cada uno a 2 Ghz) Intel i5 Nehalem, 12 GBytes de RAM y dos NVIDIA Tesla C1060 (240 núcleos a 1.30 GHz y 4 GBytes de memoria).

Dentro del primer banco de pruebas, el primer análisis está dirigido a la escalabilidad del simulador cuando se incrementa el grado de cooperación (ver Definición 6.2). Podemos asumir que cuanto mayor sea el grado de cooperación, mayor será el grado de competición entre bloques generados aleatoriamente. Los experimentos mostraron que el simulador *pdp-gpu-sim* trabaja mejor con longitudes de partes izquierdas (LHS) menores que 3. La aceleración conseguida es de 6.6x y 2.3x para longitudes entre 1 y 2, entre *pdp-gpu-sim* y *pdp-omp-sim*.

El segundo test está basado en incrementar el nivel de paralelismo a nivel de hilo en CUDA. Por tanto, se incrementa el número de bloques de reglas. La aceleración conseguida para *pdp-gpu-sim* versus *pdp-omp-sim* se muestra en la Figura 7.4. El número de simulaciones se fija a 50, entornos a 20, y la longitud media de LHS es 1,5. El speedup se estabiliza a 7x contra una sola CPU, 4,3x para 2, y 3x contra 4 CPUs.

El tercer test está dirigido al segundo nivel de paralelismo en CUDA, concerniente a los bloques de hilos. Es decir, incrementando simulaciones y entornos. El número de entornos se fija a 1 cuando se incrementan las simulaciones, y viceversa. Una tendencia mostrada en los experimentos es que el speedup se estabiliza a 3,5x para valores altos. Sin embargo, el paralelismo sobre simulaciones se lleva mejor en la GPU, dando menos speedups para entornos.

Como se estableció en [68], paralelizar por simulaciones da lugar a mejores aceleraciones en plataformas multinúcleo. Por tanto, el test finalizó con la

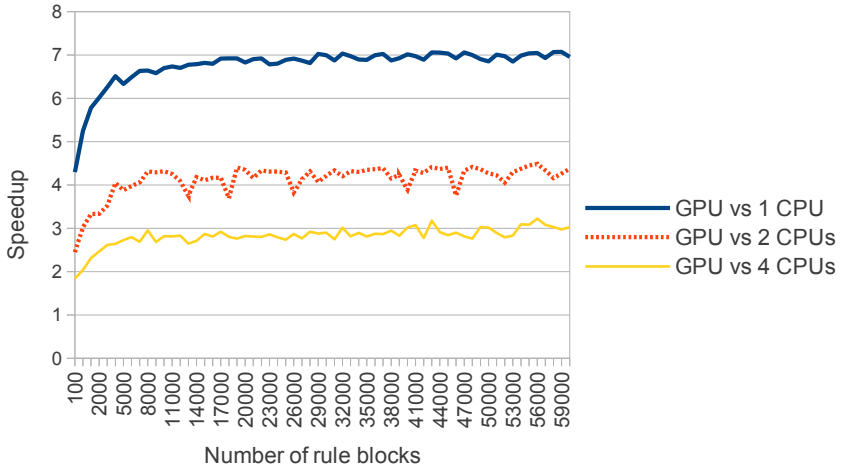


Figura 7.4: Escalabilidad de los simuladores cuando se incrementa el número de bloques de reglas.

comparativa de estos resultados con la GPU. Los bloques de reglas se fijan a 50000, los entornos a 20, los objetos a 5000 y la media de LHS a 1,5. Como se muestra en la Figura 7.5, la GPU obtiene mejor rendimiento, manteniéndose un 4,5x sobre un núcleo, 3,5x para 2, y 2,7x para 4.

Con respecto al segundo banco de pruebas, sus resultados se resumen en la Tabla 7.3. Este análisis (profile) se ha calculado ejecutando el simulador con 10000 bloques de reglas, 20 entornos, 50 simulaciones, 5000 objetos y dos diferentes longitudes medias de LHS, 1,5 (test A) y 3 (test B), respectivamente. La Fase 1 es la parte más compleja (tomando más del 50 % del tiempo de ejecución en la CPU). En el test A, la implementación GPU ofrece para la Fase 1 hasta un 14x de speedup. Por tanto, el porcentaje de ejecución se decrementa a 30 %.

Continuando con el Test A, la Fase 2 solo toma un 12 % del tiempo de CPU. Sin embargo, la GPU solo puede acelerar esta fase un 2x. Por tanto, esta fase se convierte en la más costosa cuando se ejecuta el simulador en la GPU (47 %). Nuestra implementación novel, *ph2-simorder-dyncomp*, es cercana en tiempo a la implementación secuencial. En efecto, esta fase es la más desafiante para ser paralelizada. Esfuerzos especiales deberían de considerarse a este respecto. Por otro lado, las Fases 3 y 4 son relativamente ligeras, y por tanto, aceleradas exitosamente (hasta un 13x y un 9,7x, respectivamente). Así, nuestra librería cuRNG_BINOMIAL está bien adaptada para la Fase 3.

Finalmente, como se comentó en el primer análisis del primer banco de

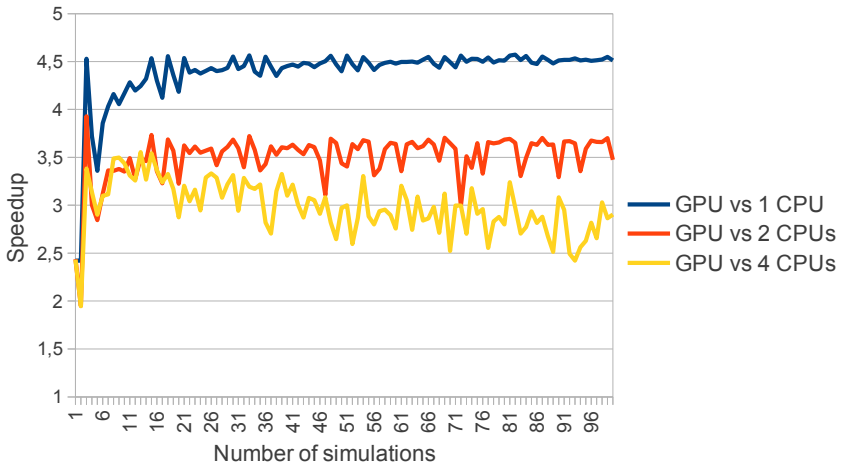


Figura 7.5: escalabilidad de los simuladores cuando se incrementa el número de simulaciones.

	Test A (Long. media LHS 1.5)			Test B (Long. media LHS 3)		
	% CPU	% GPU	Speedup	% CPU	% GPU	Speedup
Fase 1	53.7 %	30.1 %	14.23x	55.3 %	12 %	8.52x
Fase 2	12.6 %	47 %	2.13x	18.4 %	82.8 %	0.4x
Fase 3	22.6 %	13.7 %	13.2x	14 %	2.2 %	11.72x
Fase 4	11.1 %	9.2 %	9.7x	12.3 %	3 %	7.43x

Tabla 7.3: Analizando los simuladores para la GPU y una CPU.

pruebas, el rendimiento de *pdp-gpu-sim* decrementa cuando la longitud media de la parte izquierda incrementa. Para el Test B, el speedup total se decrementa desde 7.9x (Test A) a 1.8x (Test B). El porcentaje de tiempo consumido por la Fase 2 se ve incrementado por la GPU, tomando un 83 % del tiempo. Por tanto, el grado de competición de los bloques de reglas es un factor limitante para el rendimiento, el cual se interrelaciona con los resultados alcanzados.

En conclusión, las Fases 1, 3 y 4 se ejecutan eficientemente en la GPU, mientras que la Fase 2 es pobremente acelerada, ya que es inherentemente secuencial.

Parte IV

Resultados de la Tesis

"Las visiones que ofrecemos a nuestros hijos moldean el futuro. Estas visiones importan. A veces se auto-convierten en profecías. Los sueños son mapas."

Carl Sagan (1997)

8 Conclusiones

Este capítulo pone el punto y final del documento, resumiendo y analizando el trabajo presentado. Primero (Sección 8.1) ofreceremos un resumen del documento completo, capítulo a capítulo, destacando las principales contribuciones. En segundo lugar (Sección 8.2), se listarán y analizarán los resultados principales obtenidos durante el trabajo. Tercero (Sección 8.3), se mostrarán las conclusiones generales obtenidas de los resultados principales. Por último (Sección 8.4), analizaremos posibles líneas de trabajo futuro y líneas de investigación abiertas.

8.1. Resumen

Los sistemas P son dispositivos computacionales definidos en el área de la Computación con Membranas. Están basados en la abstracción de la estructura compartimentalizada y el procesamiento paralelo de información bioquímica en las células biológicas (Capítulo 1). Sin embargo, todavía no existe implementación alguna *in vivo*, *in vitro*, ni en *in silico*, principalmente por su naturaleza paralela, distribuida y no determinista. Por tanto, las computaciones prácticas de los sistemas P se llevan a cabo en simuladores basados en plataformas de silicio, aunque sus resultados potenciales se ven comprometidos por los límites físicos de dicha tecnología. Normalmente estos simuladores son ineficientes cuando manejan algunas características de los sistemas P, tales como la crea-

ción de un espacio de trabajo exponencial, el no determinismo y el paralelismo masivo.

Los simuladores para la Computación celular con Membranas han sido implementados tradicionalmente en arquitecturas CPUs orientadas a la latencia (Capítulo 2), los cuales carecen de la posibilidad de explotar la naturaleza masivamente paralela. Con el fin de mejorar su eficiencia, es necesario explotar la tecnología actual, llegando a soluciones en el área de la Computación de Alto Rendimiento (HPC), tales como los aceleradores o los procesadores de diversos núcleos (Capítulo 3). A este respecto, las Unidades de Procesamiento Gráfico (GPUs) se han consolidado como acelerados gracias a su arquitectura altamente paralela orientada al rendimiento. Por tanto, son buenas candidatas para disminuir la distancia entre las computaciones prácticas y teóricas de los sistemas P.

El objetivo de este trabajo es analizar las GPUs como plataformas paralelas para acelerar la simulación de modelos de la Computación celular con Membranas. Para este propósito es necesario definir nuevos algoritmos de simulación para sistemas P, que permitan reproducir mejor las semánticas de los modelos. Además, estos algoritmos deben de ser implementados eficientemente en la GPU. Las GPUs utilizadas para los procesos de desarrollo y experimentación son de la marca NVIDIA. Los simuladores son entonces implementados usando el modelo de programación CUDA, y la GPU específica usada para los experimentos es la NVIDIA Tesla C1060 (la cual contiene 240 núcleos y 4 GBytes de memoria). Este trabajo presenta los primeros intentos de simular sistemas P en CUDA. El documento se ha dividido en cuatro partes: preliminares (tres capítulos), simulación paralela aplicada a soluciones eficientes de problemas computacionalmente duros (dos capítulos), simulación paralela aplicada a modelos computacionales en la biología (dos capítulos), y conclusiones (un capítulo).

El Capítulo 1 ha introducido los campos interdisciplinares de la Computación Bioinspirada y Natural. La introducción se centró especialmente en la Computación celular con Membranas, detallándose los modelos definidos en el área (sistemas P). Además, se describieron dos modelos de sistemas P: sistemas P con membranas activas y sistemas P de tejido con división celular, junto a dos soluciones en tiempo linear del problema SAT usando las variantes descritas. El capítulo finalizó remarcando la necesidad de simular sistemas P en máquinas convencionales.

Esto dio lugar al Capítulo 2, el cual resumió el estado del arte en la simulación de sistemas P, con una discusión del proceso de desarrollo de simuladores. El capítulo se centró en el marco de simulación P-Lingua, y la librería Java

pLinguaCore, el cual es la semilla de nuestro trabajo. Después de discutir la necesidad de acelerar la simulación, se proveyó un análisis de la simulación paralela de sistemas P, con un pequeño resumen de los simuladores paralelos existentes.

El bloque de introducción concluyó con el Capítulo 3, el cual introdujo los conceptos correspondientes a la Computación de Alto Rendimiento y la Computación Paralela. Estos constituyen la base del proceso de desarrollo de simuladores paralelos en nuestro trabajo. El capítulo hizo especial mención a los aceleradores, especialmente a la GPU. De esta forma se introdujo el concepto de Computación con GPU, centrado en CUDA y la arquitectura de GPUs de NVIDIA.

El Capítulo 4 ha presentado el primer simulador implementado en CUDA, el cual está diseñado para sistemas P reconocedores con membranas activas. El algoritmo de simulación es el mismo que el existente en la librería pLinguaCore. Además, dicha librería construye la entrada del simulador paralelo mediante un fichero binario. El rendimiento de los simuladores creados se analizaron mediante dos casos de estudio: uno basado en una familia de sistemas P de testeo, y otro relacionado con la solución al problema SAT con membranas activas. Los experimentos mostraron que el primer caso de estudio reporta mejor aceleración que el segundo.

El Capítulo 5 ha introducido el siguiente paso natural en el trabajo, el cual es el desarrollo de simuladores *ad-hoc* para una familia de sistemas P que resuelven el problema SAT. El primer simulador fue creado para la solución de membranas activas. Aunque es un simulador menos flexible que el descrito en el Capítulo 4, la aceleración alcanzada con la GPU se multiplica. El simulador también fue mejorado para adaptarse a nuevas arquitecturas de GPUs, sistemas multi-GPU y supercomputadores. Finalmente, también se describió un simulador para la solución basada en sistemas de tejido con división celular, lo cual permitió hacer una comparativa con el otro simulador de membranas activas. Los experimentos mostraron que los sistemas a modo de células son más eficientes que los de sistemas de tejido para estas dos soluciones.

El Capítulo 6 ha descrito el trabajo en la simulación de sistemas P de Dinámica de Poblaciones (sistemas PDP). Primero se describió el marco de modelización basado en la variante de sistemas PDP, y las aplicaciones con modelos de ecosistemas reales. Después de analizar la variante, se introdujo el primer algoritmo de simulación, y punto de partida, BBB (Binomial Block Based). Acto seguido se presentaron los dos nuevos algoritmos de simulación contribuidos en este trabajo, denominados DNDP (Direct Non-Deterministic distribution with Probabilities) y DCBA (Direct distribution based on Con-

sistent Blocks Algorithm).

El Capítulo 7 finalizó la descripción del trabajo presentando los simuladores implementados para los sistemas PDP. Primero, se incluyeron implementaciones del DNDP y DCBA en la librería pLinguaCore. Después, el algoritmo DCBA fue implementado en un simulador independiente basado en C. Desde este punto de partida, se desarrollaron versiones en OpenMP y CUDA. En el capítulo se discutieron los principales retos de las implementaciones en las plataformas paralelas, como la creación de un generador de números aleatorios binomiales en la GPU. Los experimentos llevados a cabo reportan una mejor aceleración en la GPU que en una plataforma multinúcleo. Sin embargo, aún se deben de realizar más mejoras en los simuladores.

8.2. Resultados

En esta sección listaremos los principales resultados obtenidos en el trabajo desarrollado en la tesis. Los resultados que reportamos son los simuladores desarrollados, los algoritmos de simulación diseñados, y otros resultados transversales.

8.2.1. Simuladores paralelos

A continuación indicamos los modelos que fueron implementados en simuladores paralelos, junto a la tecnología correspondiente utilizada. Estos simuladores se incluyen en el proyecto software PMCGPU [12], y están disponibles bajo la licencia software GNU GPLv3. Cada simulador conforma un sub-proyecto en sí dentro de PMCGPU.

- *Sistemas P reconocedores con membranas activas en CUDA.*
- *Una familia de sistemas P con membranas activas que resuelve* `SAT` *en CUDA.*
- *Una familia de sistemas P de tejido con división celular que resuelve* `SAT` *en CUDA.*
- *Sistemas PDP mediante el algoritmo DCBA, tanto en OpenMP como en CUDA.*

8.2.2. Algoritmos de simulación

El desarrollo de algoritmos eficientes capaces de capturar fielmente la semántica descrita del marco de modelización de los sistemas PDP es una tarea difícil. Estos algoritmos deben de seleccionar reglas en el modelo de acuerdo a las probabilidades asociadas, mientras mantienen la semántica paralela y maximal. La forma en que las reglas se seleccionan en el modelo depende de algoritmo de simulación empleado. Estos algoritmos deben de manejar algunas cuestiones como la posible competición por objetos entre las reglas. Después de comprobar que el algoritmo BBB tiene una desventaja relacionada con una selección distorsionada de reglas, se introdujeron los siguientes algoritmos:

- *Algoritmo DNDP*: una primera aproximación a solucionar el problema del algoritmo BBB, inspirado en el algoritmo DND, y que realiza una distribución de objetos directa y no determinista, maximal consistente y calculando funciones de probabilidad.
- *Algoritmo DCBA*: una segunda aproximación que mejora al algoritmo DNDP, el cual estaba sesgada hacia aquellas reglas con mayor probabilidad. Este algoritmo afronta el no determinismo inherente mientras que realiza una distribución de recursos proporcional entre las reglas que eventualmente compitan por ellos.

8.2.3. Simuladores en pLinguaCore

Los dos algoritmos de simulación introducidos para los sistemas PDP, DNDP y DCBA, han sido implementados dentro de la biblioteca pLinguaCore, siendo parte de las nuevas características de la versión 3.0 de dicha biblioteca. Estas implementaciones tienen como objetivo servir para validar modelos de sistemas PDP, así como los correspondientes algoritmos:

- *DNDP en pLinguaCore*: el aspecto principal es que provee una solución paralela dentro de Java.
- *DCBA en pLinguaCore*: el aspecto principal es que mejora una fase del algoritmo, implementando una tabla en tabla hash.

8.2.4. Recursos computacionales y licencia software

La página web `http://sourceforge.net/p/pmcgpu` provee información y documentación técnica sobre los simuladores desarrollados, junto a todo el

código fuente de estos. Todo el software desarrollado en el trabajo está disponible bajo la licencia de software libre GNU GPL [2].

De forma adicional, el desarrollo del trabajo presentado en esta tesis ha requerido la instalación, configuración, administración y mantenimiento de un servidor GPU. Éste ha sido instalado dentro de una habitación habilitada para tal propósito. En ella se instaló el sistema operativo Ubuntu Linux Server edición 10.04, y ha sido expandido con 2 tarjetas gráficas NVIDIA Tesla C1060. Además, se ha instalado un administrador de colas de trabajo para permitir el acceso simultáneo de varios usuarios al servidor y sus GPUs. Una GPU se ha asignado para depuración y pruebas de ejecución rápidas, mientras que la otra para análisis de tiempo.

8.3. Conclusiones

Los sistemas P son una alternativa a la modelización de fenómenos biológicos en el campo de la Biología de Sistemas computacional y Dinámica de Poblaciones, basado en el funcionamiento de las células vivas, proveyendo un paralelismo masivo y compartimentalizado. Sin embargo, es necesario tener simuladores eficientes con el fin de validar experimentalmente los modelos con una alta productividad.

Las GPUs se están consolidando como un procesador masivamente paralelo donde los programadores pueden acelerar sus aplicaciones científicas. Son una buena alternativa a las CPUs para simular sistemas de membranas, dada la naturaleza doblemente paralela que ambos (GPUs y sistemas P) presentan. Su sistema de memoria compartida también permite realizar una sincronización eficiente de la simulación de los modelos.

El uso de la potencia y paralelismo de las GPUs para simular sistemas P es un nuevo concepto en el desarrollo de aplicaciones para la Computación celular con Membranas. Las características de la GPU pueden ayudar a los investigadores a acelerar sus simulaciones usando una arquitectura paralela, escalable y barata. Además, la Computación con GPU es lo suficientemente amplia como para adaptar los simuladores a sistemas P específicos. Por ejemplo, los sistemas P que explotan exponencialmente su espacio de trabajo alcanzan fácilmente los límites de memoria en la GPU, pero pueden ser escalados a sistemas multi-GPU o clusters de GPUs. Los sistemas P con comportamiento probabilístico también son simulados en la GPU sin perder rendimiento. Las bibliotecas como cuRAND, y la nueva cuRNG_BINOMIAL, pueden general números aleatorios a una gran velocidad.

Sin embargo, nuestros resultados muestran que la simulación de sistemas P está limitada por memoria: el rendimiento de la simulación es relativamente baja comparada con los recursos disponibles en la GPU. Las causas principales son: (1) la simulación de sistemas P requiere un alto grado de sincronización (por el reloj global de los modelos, la cooperación de reglas, la competición de reglas, etc.), y (2) el bajo número de instrucciones a ejecutar por porción de memoria. Este hecho restringe el diseño de los simuladores paralelos. Un diseñador debe de tener cuidado con la representación y manejo de cada ingrediente del sistema P. Un paso mal dado en la programación de la GPU puede romper fácilmente el paralelismo, y por tanto, el rendimiento.

Cabe recordar que tener simuladores flexibles afecta al rendimiento. Los simuladores paralelos flexibles están pensados para aprovechar el paralelismo de los sistemas P. Si el diseño de estos sistemas no tiene un gran grado de paralelismo, el rendimiento es por tanto reducido drásticamente. Por tanto, cuando se trabaja con simuladores altamente flexibles, el diseño de los sistemas P a simular debe de ser reconsiderado, de tal forma que ejecuten tantas reglas como sean posibles de manera simultánea. En cambio, cuanto menos flexibles sean, más eficientes serán. Por tanto, los simuladores *ad-hoc* pueden ejecutarse más eficientemente en GPUs.

Por último, es necesario definir nuevos algoritmos de simulación para reproducir mejor el comportamiento esperado de la computación de los sistemas P. Para sistemas PDP es muy importante alcanzar este hecho, de tan forma que los modelos de ecosistemas puedan reproducir mejor los procesos de dinámica de poblaciones. Además, los algoritmos de simulación deben de considerar el rendimiento con el fin de ser implementados fácilmente en plataformas paralelas.

Podemos concluir finalmente que la llegada de los acelerados en la Computación de Alto Rendimiento ofrece formas nuevas de desarrollar simuladores eficientes para sistemas P.

8.4. Trabajo futuro

Este trabajo presenta los primeros intentos en la simulación de sistemas P usando GPUs. Por tanto, se abren nuevas líneas de investigación para continuar y expandir el objetivo inicial: implementar el paralelismo de los sistemas P en plataformas de alto rendimiento. Destacamos las siguientes líneas:

- *Conectar los simuladores desarrollados con librerías de simulación:* esto permitirá usar de manera efectiva los simuladores por la comunidad

científica, requiriendo diseñar nuevos protocolos de comunicación eficientes.

- *Simular otros modelos con plataformas paralelas basadas en GPUs*: para este fin habría que analizar previamente la importancia del modelo para justificar la necesidad de simuladores paralelos.

- *Mejorar los simuladores existentes*: esfuerzos especiales deberían de tomarse mejorando el uso de memoria (mediante compresión de datos), adaptando la ejecución en CUDA a arquitecturas específicas de GPUs, ampliar la flexibilidad de los simuladores y aplicar heurísticas basadas en los modelos.

- *Mejorar los algoritmos de simulación*: a este fin habría que mejorar la forma de reproducir la semántica, a fin de mejorar el comportamiento y el rendimiento de los algoritmos. Por ejemplo, habría que reconsiderar la simulación de la aplicación maximal y paralela.

- *Usar nueva tecnología paralela*: las nuevas generaciones de tarjetas gráficas (como la arquitectura NVIDIA Kepler, y las APU de AMD) ofrecen mejoras de rendimiento que son interesantes para mejorar los simuladores desarrollados. De igual forma se podría analizar el escalado de los simuladores a plataformas más amplias como la computación grid y en nube.

- *Estudiar los modelos para simuladores paralelos*: más investigación se puede llevar a cabo con respecto a la simulación de características particulares de sistemas P. A fin de analizar los modelos que encajen bien con los simuladores flexibles, se puede estudiar previamente el grado de paralelismo del modelo con *sevilla carpets*. Por otro lado, sería interesante definir un tipo de sistema P (sistemas GP) que su simulación encaje perfectamente con la GPU, y que otros modelos se puedan adaptar a este otro fácilmente.

- *Modelar la GPU*: una línea de investigación teórica sería analizar los aspectos computacionales de la GPU mediante un modelo formal basado en sistemas P.

Bibliografía

[1] B. Barney. Introduction to Parallel Computing. https://computing.llnl.gov/tutorials/parallel_comp. Tutorial of "Using LLNL's Supercomputers" workshop.

[2] GNU GPL license. http://www.gnu.org/licenses/gpl.html.

[3] Inside HPC blog. http://insidehpc.org.

[4] Message Passing Interface forum. http://www.mpi-forum.org. Location containing the official MPI standards documents.

[5] NVIDIA CUDA C Programming Guide 4.2. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[6] OpenCL standard webpage. http://www.khronos.org/opencl.

[7] The Berkeley Open Infrastructure for Network Computing (BOINC) official website. http://boinc.berkeley.edu.

[8] The GPGPU organization. http://www.gpgpu.org.

[9] The NVIDIA CUDA Random Number Generation library (cuRAND). https://developer.nvidia.com/curand.

[10] The OpenMP API specification for parallel programming. http://www.openmp.org. Official website.

[11] The P-Lingua web page. http://www.p-lingua.org.

[12] The PMCGPU (Parallel simulators for Membrane Computing on the GPU) project website. http://sourceforge.net/p/pmcgpu.

[13] M. Qasem. WinSAT website. http://www.mqasem.net/sat/winsat, 2009.

[14] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the American Federation of Information Processing Societies Conference*, volume 30, pages 483–485, Atlantic City, NJ, April 1967.

[15] A. A. Aqrawi and A. C. Elster. Bandwidth reduction through multithreaded compression of seismic images. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1730–1739, may 2011.

[16] D. Besozzi, P. Cazzaniga, D. Pescini, and G. Mauri. Modelling metapopulations with stochastic membrane systems. *Biosystems*, 91(3):499–514, 2008.

[17] L. Bianco, F. Fontana, and V. Manca. P systems with reaction maps. *International Journal of Foundations of Computer Science*, 17(1):27–48, 2006.

[18] G. Bravo, L. Fernández, F. Arroyo, and M. A. Peña. Hierarchical master-slave architecture for membrane systems implementation. In *Thirteenth International Symposium on Artificial Life and Robotics 2008*, AROB 13th, 2008.

[19] F. G. Cabarle, H. N. Adorna, and M. A. Martínez-del-Amor. A spiking neural P system simulator based on CUDA. In M. Gheorghe, G. Păun, G. Rozenberg, A. Salomaa, and S. Verlan, editors, *Membrane Computing*, volume 7184 of *Lecture Notes in Computer Science*, pages 87–103. Springer Berlin Heidelberg, 2012.

[20] F. G. Cabarle, H. N. Adorna, M. A. Martínez-del-Amor, and M. J. Pérez-Jiménez. Improving GPU simulations of spiking neural P systems. *Romanian Journal of Information Science and Technology*, 15:5–20, 2012.

[21] M. Cardona, M. A. Colomer, A. Margalida, A. Palau, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and D. Sanuy. A computational modeling for real ecosystems based on P systems. *Natural Computing*, 10(1):39–53, 2011.

[22] M. Cardona, M. A. Colomer, A. Margalida, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and D. Sanuy. A P system based model of an ecosystem of some scavenger birds. In G. Păun, M. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume

5957 of *Lecture Notes in Computer Science*, pages 182–195. Springer Berlin Heidelberg, 2010.

[23] M. Cardona, M. A. Colomer, M. J. Pérez-Jiménez, D. Sanuy, and A. Margalida. Modeling ecosystems using P systems: the bearded vulture, a case study. In D. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 137–156. Springer Berlin Heidelberg, 2009.

[24] A. Castellini, V. Manca, and Y. Suzuki. Metabolic P system flux regulation by artificial neural networks. In G. Păun, M. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 196–209. Springer Berlin Heidelberg, 2010.

[25] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Implementing P systems parallelism by means of GPUs. In G. Păun, M. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 227–241. Springer Berlin Heidelberg, 2010.

[26] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Simulating a P system based efficient solution to SAT by using GPUs. *Journal of Logic and Algebraic Programming*, 79(6):317–325, 2010.

[27] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Simulation of P systems with active membranes on CUDA. *Briefings in Bioinformatics*, 11(3):313–322, 2010.

[28] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, M. J. Pérez-Jiménez, and M. Ujaldón. P systems simulations on massively parallel architectures. In *Third International Workshop on Parallel Architectures and Bioinspired Algorithms*, pages 17–26, Vienna, Austria, 2010.

[29] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del-Amor, M. J. Pérez-Jiménez, and M. Ujaldón. The GPU on the simulation of cellular computing models. *Soft Computing*, 16(2):231–246, 2012.

[30] J. M. Cecilia, G. D. Guerrero, J. M. García, M. A. Martínez-del-Amor, Ignacio Pérez-Hurtado, and M. J. Pérez-Jiménez. Simulation of P Systems with active membranes on CUDA. In *2009 International Workshop on High Performance Computational Systems Biology, HIBI'09*, pages 61–71, Trento, Italy, 2009. IEEE Computer Society.

[31] J. M. Cecilia, G. D. Guerrero, J. M. García, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. A massively parallel framework using P systems and GPUs. In *Symposium on Application Accelerators in High Performance Computing*, Illinois, USA, 2009.

[32] J. M. Cecilia, G. D. Guerrero, J. M. García, M. A. Martínez-del-Amor, M. J. Pérez-Jiménez, and M. Ujaldón. Enhancing the simulation of P systems for the SAT problem on GPUs. In *Symposium on Application Accelerators in High Performance Computing*, Knoxville, USA, July 2010 2010.

[33] S. Cheruku, A. Păun, F. J. Romero-Campero, M. J. Pérez-Jiménez, and O. H. Ibarra. Simulating FAS-induced apoptosis by using P systems. *Progress in Natural Science*, 17:424–431, 2007.

[34] G. Ciobanu and G. Wenyuan. A P system running on a cluster of computers. In *Lecture Notes in Computer Science*, WMC 2003, pages 123–150. Springer-Verlag, 2004.

[35] M. Colomer, A. Margalida, D. Sanuy, and M. J. Pérez-Jiménez. A bio-inspired computing model as a new tool for modeling ecosystems: The avian scavengers as a case study. *Ecological Modelling*, 222(1):33–47, 2011.

[36] M. Colomer, M. Martínez-del-Amor, I. Pérez-Hurtado, M. Pérez-Jiménez, and A. Riscos-Núñez. A uniform framework for modeling based on P systems. In *Bio-Inspired Computing: Theories and Applications (BIC-TA), 2010 IEEE Fifth International Conference on*, volume 1, pages 616–621, September 2010.

[37] M. A. Colomer, S. Lavín, I. Marco, A. Margalida, I. Pérez-Hurtado, M. J. Pérez-Jiménez, D. Sanuy, E. Serrano, and L. Valencia-Cabrera. Modeling population growth of Pyrenean Chamois (Rupicapra p. pyrenaica) by using P systems. In M. Gheorghe, T. Hinze, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 6501 of *Lecture*

Notes in Computer Science, pages 144–159. Springer Berlin Heidelberg, 2011.

[38] T. S. Crow. Evolution of the graphical processing unit. Master's thesis, University of Nevada Reno, `http://www.cse.unr.edu/~fredh/papers/thesis/023-crow/GPUFinal.pdf`, 2004.

[39] D. Díaz-Pernil, C. Graciani-Díaz, M. A. Gutiérrez-Naranjo, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. *Software for P systems*, chapter 17, pages 437–454. Oxford University Press, Oxford (U.K.), 2010.

[40] L. Diez Dolinski, R. Núñez Hervás, M. Cruz Echeandía, and A. Ortega. Distributed simulation of P systems by means of Map-Reduce: first steps with Hadoop and P-Lingua. In J. Cabestany, I. Rojas, and G. Joya, editors, *Advances in Computational Intelligence*, volume 6691 of *Lecture Notes in Computer Science*, pages 457–464. Springer Berlin Heidelberg, 2011.

[41] A. C. Elster. High-Performance Computing: past, present, and future. In J. Fagerholm, J. Haataja, J. Järvinen, M. Lyly, P. Raback, and V. Savolainen, editors, *Applied Parallel Computing*, volume 2367 of *Lecture Notes in Computer Science*, pages 433–444. Springer Berlin Heidelberg, 2006.

[42] R. Farber. *CUDA application design and development*. Elsevier, 2011.

[43] L. Fernández, V. J. Martínez, F. Arroyo, and L. F. Mingo. A hardware circuit for selecting active rules in transition p systems. In *SYNASC*, pages 415–418, 2005.

[44] F. Fontana, L. Bianco, and V. Manca. P systems and the modeling of biochemical oscillations. In R. Freund, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 3850 of *Lecture Notes in Computer Science*, pages 199–208. Springer Berlin Heidelberg, 2006.

[45] I. T. Foster. What is the Grid? - a three point checklist. *GRIDtoday*, 1(6), 2002.

[46] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. An overview of P-Lingua 2.0. In G. Păun, M. J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2010.

[47] M. García-Quismondo, L. F. Macías-Ramos, and M. J. Pérez-Jiménez. Implementing enzymatic numerical P systems for AI applications by means of graphic processing units. In J. Kelemen, J. Romportl, and E. Zackova, editors, *Beyond Artificial Intelligence*, volume 4 of *Topics in Intelligent Engineering and Informatics*, pages 137–159. Springer Berlin Heidelberg, 2013.

[48] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[49] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Pearson Education, Harlow, England, 2nd edition, 2003.

[50] G. D. Guerrero, J. M. Cecilia, J. M. García, M. A. Martínez-del-Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Analysis of P systems simulation on CUDA. In *XX Jornadas de Paralelismo*, pages 289–294, A coruña, Spain, September 2009. Servizo de Publicacións, Universidade da Coruña.

[51] J. L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[52] A. Gutiérrez and S. Alonso. *P systems: from theory to implementation*, chapter 17, pages 205–226. Concept Press Ltd, Hong Kong, 2010.

[53] A. Gutiérrez, L. Fernández, F. Arroyo, and S. Alonso. Hardware and software architecture for implementing membrane systems: A case of study to transition P systems. In M. Garzon and H. Yan, editors, *DNA Computing*, volume 4848 of *Lecture Notes in Computer Science*, pages 211–220. Springer Berlin Heidelberg, 2008.

[54] R. A. Juayong, F. G. Cabarle, H. N. Adorna, and M. A. Martínez-del-Amor. On the simulations of evolution-communication P systems with energy without antiport rules for GPUs. In *Tenth Brainstorming Week on Membrane Computing*, volume I, pages 267–290, Seville, Spain, February 2012. Fénix Editora.

[55] V. Kachitvichyanukul and B. W. Schmeiser. Binomial random variate generation. *Communications of the ACM*, 31(2):216–222, 1988.

[56] B. W. Kernighan and D. Ritchie. *The C programming language*. Prentice Hall, 2nd edition, 1988.

[57] D. B. Kirk and W. W. Hwu. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[58] K. Krewell. Cell moves into the limelight, 2005. Microprocessor Report.

[59] H. Li and L. R. Petzold. Efficient parallelization of stochastic simulation algorithm for chemically reacting systems on the graphics processing unit. Technical report, Dept. Computer Science, University of California, 2007.

[60] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: a unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[61] V. Manca. *Fundamentals of Metabolic P Systems*, chapter 19, pages 475–498. Oxford University Press, Oxford (U.K.), 2010.

[62] V. Manca, R. Pagliarini, and S. Zorzan. A photosynthetic process modelled by a metabolic P system. *Natural Computing*, 8(4):847–864, 2009.

[63] M. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, Á. Romero-Jiménez, C. Graciani-Díaz, A. Riscos-Núñez, M.A. Colomer, and M.J. Pérez-Jiménez. DCBA: Simulating population dynamics P systems with proportional object distribution. In *Proceedings of the 13th International Conference on Membrane Computing (CMC13)*, pages 291–310, Budapest, Hungary, August 2012.

[64] M. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, Á. Romero-Jiménez, C. Graciani-Díaz, A. Riscos-Núñez, M.A. Colomer, and M.J. Pérez-Jiménez. DCBA: Simulating Population Dynamics P systems with proportional object distribution. In *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, volume II, pages 27–56, Seville, Spain, February 2012. Fénix Editora.

[65] M. Martínez-del-Amor, I. Pérez-Hurtado, M. Pérez-Jiménez, A. Riscos-Núñez, and M. Colomer. A new simulation algorithm for multienvironment probabilistic P systems. In *IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2010)*, volume 1, pages 59–68, September 2010.

[66] M. Martínez-del-Amor, I. Pérez-Hurtado, M. Pérez-Jiménez, A. Riscos-Núñez, and F. Sancho-Caparrini. A simulation algorithm for multienvironment probabilistic p systems: a formal verification. *International Journal of Foundations of Computer Science*, 22(01):107–118, 2011.

[67] M. A. Martínez-del-Amor, J. M. Cecilia, G. D. Guerrero, and I. Pérez-Hurtado. An overview of P system simulation on GPUs. In *I Jornadas Jóvenes Investigadores*, pages 2–7, Cáceres, Spain, April 2010.

[68] M. A. Martínez-del-Amor, I. Karlin, R. E. Jensen, M. J. Pérez-Jiménez, and A. C. Elster. Parallel simulation of probabilistic P systems on multicore platforms. In *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, volume II, pages 17–26, Seville, Spain, February 2012. Fénix Editora.

[69] M. A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A. C. Elster, and M. J. Pérez-Jiménez. Population Dynamics P Systems on CUDA. In D. Gilbert and M. Heiner, editors, *Computational Methods in Systems Biology*, Lecture Notes in Computer Science, pages 247–266. Springer Berlin Heidelberg, 2012.

[70] M. A. Martínez-del-Amor, I. Pérez-Hurtado, M. J. Pérez-Jiménez, J. M. Cecilia, G. D. Guerrero, and J. M. García. Simulating active membrane systems using GPUs. In G. Paun, M. J. Pérez-Jiménez, and A. Riscos-Núñez, editors, *10th Workshop on Membrane Computing*, pages 369–384, Curtea de Arges, Rumania, August 2009. Marpapublicidad.

[71] M. A. Martínez-del-Amor, I. Pérez-Hurtado, M. J. Pérez-Jiménez, J. M. Cecilia, G. D. Guerrero, and J. M. García. Simulation of recognizer P systems by using manycore GPUs. In *7th Brainstorming Week on Membrane Computing*, volume II, pages 45–58, Sevilla, España, February 2009. Fénix Editora.

[72] C. Maxfield. *The Design Warrior's Guide to FPGAs*. Elsevier, 2004.

[73] V. Moya, C. González, J. Roca, A. Fernández, and R. Espasa. Shader performance analysis on a modern GPU architecture. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 355–364, Washington, DC, USA, 2005. IEEE Computer Society.

[74] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL programming guide*. Addison-Wesley, 1st edition, 2011.

[75] V. Nguyen, D. Kearney, and G. Gioiosa. Balancing performance, flexibility, and scalability in a parallel computing platform for membrane computing applications. In *Proceedings of the 8th Workshop on Membrane computing*, WMC'07, pages 385–413, Berlin, Heidelberg, 2007. Springer-Verlag.

[76] V. Nguyen, D. Kearney, and G. Gioiosa. An algorithm for non-deterministic object distribution in p systems and its implementation in hardware. In D. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2009.

[77] V. Nguyen, D. Kearney, and G. Gioiosa. An extensible, maintainable and elegant approach to hardware source code generation in reconfig-p. *Journal of Logic and Algebraic Programming*, 79(6):383–396, 2010.

[78] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[79] P. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, Burlington, USA, 1st edition, 2011.

[80] G. Păun and F. J. Romero-Campero. Membrane Computing as a modeling framework. Cellular systems case studies. In M. Bernardo, P. Degano, and G. Zavattaro, editors, *Formal Methods for Computational Systems Biology*, volume 5016 of *Lecture Notes in Computer Science*, pages 168–214. Springer Berlin Heidelberg, 2008.

[81] F. Peña-Cantillana, D. Díaz-Pernil, H. A. Christinal, and M. A. Gutiérrez-Naranjo. Implementation on CUDA of the smoothing problem with tissue-like P systems. *International Journal on Natural Computing Research*, 2(3):25–34, 2011.

[82] J. Pérez-Carrasco. Aceleración de simulaciones de sistemas celulares en soluciones del problema sat usando gpus. Master's thesis, Higher Technical School of Computer Engineering, July 2012.

[83] I. Pérez-Hurtado. *Desarrollo y aplicaciones de un entorno de programación para Computación Celular: P-Lingua*. PhD thesis, University of Seville, 2010.

[84] I. Pérez-Hurtado, L. Valencia-Cabrera, M. Pérez-Jiménez, M. Colomer, and A. Riscos-Núñez. MeCoSim: A general purpose software tool for simulating biological phenomena by means of P systems. In *2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA)*, volume 1, pages 637–643, September 2010.

[85] M. Pérez-Jiménez and F. Romero-Campero. P systems, a new computational modelling tool for systems biology. In C. Priami and G. Plotkin, editors, *Transactions on Computational Systems Biology VI*, volume 4220 of *Lecture Notes in Computer Science*, pages 176–197. Springer Berlin Heidelberg, 2006.

[86] M. J. Pérez-Jiménez, Á. Romero-Jiménez, and F. Sancho-Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing*, 2(3):265–285, 2003.

[87] D. Pescini, D. Besozzi, G. Mauri, and C. Zandron. Dynamical probabilistic p systems. *International Journal of Foundations of Computer Science*, 17(1):183–204, 2006.

[88] B. Petreska and C. Teuscher. A reconfigurable hardware membrane system. In C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 269–285. Springer Berlin Heidelberg, 2004.

[89] C. Prabhu. *Grid and Cluster Computing*. Prentice-Hall, 2008.

[90] G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143; Turku Center for CS–TUCS Report No 208 (1998), 2000.

[91] G. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing*. Oxford University Press, USA, 2010.

[92] F. J. Romero-Campero and M. J. Pérez-Jiménez. A model of the quorum sensing system in vibrio fischeri using P systems. *Artificial Life*, 14(1):95–109, 2008.

[93] F. J. Romero-Campero and M. J. Pérez-Jiménez. Modelling gene expression control using p systems: The lac operon, a case study. *Biosystems*, 91(3):438–457, 2008.

[94] A. Ruíz, M. Ujaldón, J. A. Andrades, J. Becerra, K. Huang, T. Pan, and J. H. Saltz. The GPU on biomedical image processing for color and phenotype analysis. In *7th IEEE International Conference on Bioinformatics and Bioengineering*, pages 1124–1128, Piscataway, NJ, USA, October 2007. IEEE Computer Society.

[95] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE Computer Society, May 2009.

[96] A. Syropoulos, L. Mamatas, P. C. Allilomes, and K. T. Sotiriades. A distributed simulation of transition P systems. In *Workshop on Membrane Computing*, pages 357–368, 2003.

[97] J. A. Tejedor, L. Fernández, F. Arroyo, and G. Bravo. An architecture for attacking the communication bottleneck in P systems. *Artificial Life and Robotics*, 12(1-2):236–240, 2008.

[98] G. Terrazas, N. Krasnogor, M. Gheorghe, F. Bernardini, S. Diggle, and M. Cámara. An environment aware P system model of quorum sensing. In S. Cooper, B. Löwe, and L. Torenvliet, editors, *New Computational Paradigms*, volume 3526 of *Lecture Notes in Computer Science*, pages 479–485. Springer Berlin Heidelberg, 2005.

[99] W. Voorsluys, J. Broberg, and R. Buyya. *Introduction to Cloud Computing*, pages 1–41. Wiley press, 2011.

[100] N. Whitehead and A. Fit-Florea. Precision & performance: floating point and IEEE 754 compliance for NVIDIA GPUs, 2011.

[101] C. Zandron, C. Ferretti, and G. Mauri. Solving np-complete problems using p systems with active membranes. In *Proceedings of the Second International Conference on Unconventional Models of Computation*, UMC '00, pages 289–301, London, UK, 2001. Springer-Verlag.

[102] X. Zeng, H. Adorna, M. A. Martínez-del-Amor, L. Pan, and M. J. Pérez-Jiménez. Matrix representation of spiking neural p systems. In

M. Gheorghe, T. Hinze, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 377–391. Springer Berlin Heidelberg, 2011.