UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

"DISEÑO Y OPTIMIZACIÓN DE IMPLEMENTACIONES DE MODELOS DE AUTÓMATAS CELULARES PARA ARQUITECTURAS PARALELAS CON ACCESO NO UNIFORME A MEMORIA."

TESIS

QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

PRESENTA:
MANUEL EDUARDO SÁNCHEZ SOLCHAGA

Director de tesis:
DRA. MARÍA ELENA LÁRRAGA RAMÍREZ
Instituto de Ingeniería, UNAM

Ciudad Universitaria, Ciudad de México. Enero, 2018

Dedicatoria.

A las mujeres más importantes en mi vida: mi esposa Selene, quien me ha dado fuerzas y me ha motivado siempre con todo su amor a lo largo de este tiempo para culminar este proyecto; mis hijas, Camila Sofía y Daniela Fernanda, quienes son mi motivación y orgullo; mi madre, por su cariño y apoyo incondicional.

A todas, las amo.

Agradecimientos.

Primero que nada, quiero agradecer a mi esposa Selene y a mis hijas Camila Sofía y Daniela Fernanda, por su todo su apoyo, paciencia y comprensión. A mi madre, a mi tía Gloria y a mis cuñadas, quienes siempre nos apoyaron incondicionalmente durante mi ausencia.

Agradezco inmensamente a la Dra. María Elena Lárraga Ramírez, por su invaluable apoyo, sabios consejos, paciencia y su constante dedicación para la realización de este proyecto. Además de ser una excelente tutora, es una persona muy humana. Ha sido un honor ser su estudiante.

También quiero agradecer a mis sinodales, al Dr. José David Flores Peñaloza, al Mtro. Juan Luciano Díaz González, al Mtro. José Luis Gordillo Ruíz y al Dr. Demetrio Fabián García Nocetti, quienes hicieron un gran trabajo al revisar esta tesis y brindaron comentarios para enriquecer su contenido. Muchas gracias por su cooperación, la cual aprecio profundamente.

Agradezco al Ing. José Fernando Maldonado Salgado del Instituto de Ingeniería, por las facilidades brindadas para el uso del cluster, las herramientas requeridas y el apoyo en conocimiento que hicieron posible este trabajo de tesis. Adicionalmente, agradezco al Ing. Juan Eduardo Murrieta León y al Mat. Enrique Palacios Boneta, del Instituto de Ciencias Nucleares, por su amistosa y productiva colaboración, ayudándome a resolver dudas y guiándome durante la elaboración de este trabajo.

En particular, quiero enfatizar fuertemente y expresar mi gratitud a la asistente de procesos del posgrado, Ma. del Lourdes González Lora, por su guía, paciencia y disposición para apoyarnos siempre. Así como a la técnico Juana González Bautista, quien desde el primer día que llegué a este posgrado me brindó su ayuda.

Quiero dar un agradecimiento especial a mis compañeros y amigos Pedro, Anuar, Osvaldo y Francisco; con quienes conviví durante mi estancia en este posgrado.

Finalmente, pero no menos importante, quiero expresar mi más profunda gratitud a las instituciones que hicieron posible este trabajo. A la Universidad Nacional Autónoma de México y a su Posgrado en Ciencia e Ingeniería de la Computación, por permitirme realizar mis estudios. Al CONACyT, por la beca otorgada para la realización de este proyecto, al Instituto de Ingeniería, por la beca para el uso de las instalaciones, equipo y recursos, así como al proyecto PAPIIT-DGAPA No IN112716.

Resumen.

Recientemente, los modelos de Autómatas Celulares (AC) han sido utilizados en una amplia gama de áreas de la ciencia, debido a que permiten generar un comportamiento emergente complejo, a partir de información obtenida a partir de las interacciones locales entre sus elementos que lo conforman a través de reglas relativamente simples. Por lo que los AC han sido utilizados en una amplia gama de áreas de aplicación, tales como: planeamiento urbano sustentable, análisis del impacto ecológico, evacuaciones e incendios, predicción del crecimiento de las ciudades y del uso de la tierra, tráfico vehicular, crecimiento de tumores, entre muchas otras. Sin embargo, cuando se modelan sistemas complejos muy grandes, con millones de partículas, el cálculo de las simulaciones puede ser una tarea que consume mucho tiempo para poder realizarlas. Para este tipo de sistemas, es indispensable el uso del cómputo de alto desempeño para la simulación y análisis de los modelos. En este trabajo, se realiza un análisis del desempeño de implementaciones paralelas de modelos de autómatas celulares a gran escala. La finalidad es determinar si al considerar la estructura topológica del sistema en donde se ejecutan las implementaciones, es posible establecer una estrategia que no solo brinde un mejor desempeño al minimizar los tiempos de ejecución, como la mayoría de los estudios existentes en la literatura, sino también de utilizar de manera adecuada los recursos existentes. Resultados de simulación computacional y su respectivo análisis indican que, el uso de afinidad de procesos y de memoria puede mejorar el rendimiento de las implementaciones paralelas de los modelos de AC a gran escala.

Abstract.

Recently, Cellular Automata (CA) models have been used in a wide range of areas of science, because they allow to generate a complex and emergent behavior, based on information obtained from local interactions among its elements that make it up through relatively simple rules. So that CAs have been used in a wide range of application areas, such as: sustainable urban planning, ecological impact analysis, evacuations and fires, prediction of the growth of the cities and of the use of the earth, vehicular traffic, tumor growth, among many others. However, when very big complex systems are modeled, with millions of particles, the calculation of simulations can be a task that consumes a lot of time to be able to perform them. For this type of systems, it is indispensable the use of high-performance computing for the simulation and analysis of models. In this work, it´s performed an analysis of the performance of parallel implementations of large-scale cellular automata models. The purpose is to determine if when considering the system topological structure where implementations are executed, it is possible to establish a strategy that not only provides better performance by minimizing execution times, like most existing studies in the literature, but also to use adequate way the existing resources. Computational simulation results and its respective analyzes indicate that, the use of process and memory affinity can improve the performance of parallel implementations of large-scale CA models.

Tabla de contenido.

Introducción.

El desarrollo de modelos a través de los cuales las computadoras pueden simular la evolución de sistemas artificiales y naturales es fundamental para el avance de la ciencia. En las últimas décadas, el incremento continuo del poder de las computadoras ha permitido la extensión de la aplicación de metodologías computacionales en la investigación y la industria, así como al estudio cuantitativo de fenómenos con comportamiento complejo. Uno de estos paradigmas son los Autómatas Celulares (AC), lo cuales han demostrado su efectividad para modelar sistemas complejos o propósitos de estudio específicos, cuando no es posible resolverlos de manera exacta o resulta muy complicado llevarlo a cabo, como por ejemplo a través de ecuaciones diferenciales.

Los AC son un paradigma de modelado de sistemas complejos muy conocido y explotado en una amplia gama de áreas de aplicación. Son sistemas dinámicos discretos en espacio y tiempo, formados por un gran número de componentes simples, idénticos e interconectados localmente. Recientemente, han ganado popularidad debido a que permiten generar un comportamiento emergente complejo, a partir únicamente de información obtenida de las interacciones locales entre sus elementos que lo conforman a través de reglas relativamente simples. Por lo que los AC han sido utilizados en una amplia gama de áreas de aplicación, tales como: planeación urbana sustentable, análisis del impacto ecológico, evacuaciones e incendios, predicción del crecimiento de las ciudades y del uso de la tierra, tráfico vehicular, crecimiento de tumores, entre muchas otras. El interés por el uso de los AC, se debe a que están caracterizados por ser capaces de generar un comportamiento emergente macroscópico y complejo, generado únicamente a partir de las interacciones locales entre el conjunto finito de elementos homogéneos que lo conforman, a través de reglas microscópicas relativamente simples.

Aun cuando los cálculos realizados por las reglas de transición de los AC para modelar la dinámica de los sistemas que simulan pueden ser extremadamente simples y no requieren de una gran capacidad de cómputo para poder realizarlas, cuando se modelan sistemas complejos para aplicaciones reales con un gran número de entidades esto no es así. Para estos casos, el número de iteraciones entre entidades en el tiempo y el espacio, se incrementa ampliamente y, por lo tanto, el número de operaciones derivadas. Por lo que el cálculo de las simulaciones puede llegar a ser una tarea que requiere mucho tiempo computacional para poder realizarla. Para este tipo de sistemas es conveniente el uso del cómputo de alto desempeño para la simulación y análisis de los modelos en un tiempo razonable que permita su uso en aplicaciones reales.

Así, en los últimos años se han realizado diversos estudios orientados a la implementación paralela de modelos basados en AC para procesar grandes cantidades de datos en un menor tiempo, en comparación con sus contrapartes secuenciales y/o debido a que permiten manejar grandes espacios de dominio, que sería imposible manejar en un solo equipo. Estos estudios se han realizado con base en el uso de diversas arquitecturas de cómputo paralelas, como clusters [47,53,56], procesadores multicore y

manycore [43,47,48,53,55], GPUs [41,43,45,46,49,55], FPGAs [41,44,49,50], etc., y su combinación [46,53].

Particularmente, en el trabajo realizado por Rybacky [55], se realiza una comparación de varias implementaciones de diferentes modelos de AC, evaluados en arquitecturas multicore y GPU. Ellos encuentran que en la mayoría de los casos los algoritmos basados en GPU superan a los algoritmos basados en multicore, pero que existen algunos problemas específicos donde ocurre lo contrario. Por lo que concluyen que a pesar de que los GPU son una buena alternativa para el desarrollo de implementaciones paralelas de AC, su utilidad depende del modelo a ser simulado. Por otra parte, Kalgin presenta un estudio comparativo de varias implementaciones de modelos de AC asíncronos para simular un proceso químico en varias arquitecturas paralelas (multicore, cluster y GPU) [54]. En ese trabajo, se estudiaron diversas técnicas para acelerar la simulación del juego de la vida con base en CUDA, logrando una implementación con tiempos de cómputo mejorados. Se concluye que la eficiencia de algunos de los algoritmos es alta en determinadas arquitecturas paralelas. Además, se sugiere como mejoras futuras enfocarse en reducir los costos de las comunicaciones. Bezbradica et al. [51] realizaron un estudio sobre estrategias de paralelización de autómatas celulares de gran escala para una aplicación farmacéutica. Para ello, utilizan implementaciones de memoria compartida desarrolladas con el API OpenMP, de memoria distribuida con la librería de paso de mensajes MPI y un híbrido de ambas implementaciones. Los resultados muestran que el enfoque híbrido ofrece un mejor rendimiento, seguido cercanamente de la solución MPI pura. Adicionalmente, sugieren realizar futuras mejoras al experimentar con diferentes métodos de balanceo de carga. Posteriormente, Millán et al. [42], presentaron un análisis del rendimiento de implementaciones paralelas tanto del modelo de AC conocido como el “Juego de la Vida” (GoL, por sus siglas en inglés), como de un modelo de Lattice Boltzman, en varias arquitecturas. En ese trabajo, realizaron un estudio utilizando contadores de hardware, con los cuales pueden verificar la eficiencia del algoritmo basado en los accesos a memoria local; además, hicieron un estudio sobre la escalabilidad de la implementación, en donde para el GoL con una malla de 64,000 × 64,000 y 1000 generaciones alcanzaron ∼56x de aceleración con ~87% de eficiencia. También realizaron, aunque de manera muy básica, un análisis de los costos de comunicación entre los procesos remotos al realizar un escalamiento; sin embargo, se limita a reducir el espacio de dominio hasta encontrar un porcentaje de comunicación pequeño, que no afecte el rendimiento.

A pesar de que todos los trabajos aquí mencionados utilizan un método de descomposición de dominio para realizar la paralelización de los modelos de AC, ninguno especifica detalladamente la estrategia para realizar la aglomeración y el mapeo de los datos a los procesos; ni tampoco, ninguno toma en cuenta las características de las arquitecturas en donde se ejecutan dichas implementaciones, a fin de utilizar de manera adecuada los recursos existentes.

De esta manera, la hipótesis que orienta este trabajo de tesis es la siguiente:

Dada una implementación paralela de un modelo de autómatas celulares, un espacio de dominio de grandes dimensiones para representar a un sistema a gran escala, y una cantidad variable de procesadores y procesos por procesador, ¿Es posible mejorar su rendimiento, al considerar la

estructura topológica de la arquitectura, de manera que al aplicar tanto la afinidad de procesos como la afinidad de memoria de manera adecuada se logre incrementar la eficiencia y a la vez algoritmos altamente escalables?

El objetivo de este trabajo de tesis se enfoca en realizar un análisis del desempeño de implementaciones paralelas híbridas de modelos de autómatas celulares a gran escala, utilizando diversas configuraciones del espacio de dominio, en términos de escalabilidad y eficiencia. La finalidad se enfoca en determinar si al considerar la estructura topológica del sistema en donde se ejecutan, permite una estrategia que no solo brinde un mejor desempeño al minimizar los tiempos de ejecución, como la mayoría de los estudios existentes en la literatura, sino también de utilizar de manera adecuada los recursos existentes. Para este propósito, se proponen tres implementaciones paralelas híbridas, las cuales se distinguen por la manera en que mapean los procesos y se colocan los datos en memoria. Mediante un análisis de escalamiento fuerte y débil, se realiza una evaluación de las implementaciones propuestas. Los resultados indican que, el uso de afinidad de procesos y memoria puede mejorar el rendimiento de las implementaciones paralelas de modelos de AC a gran escala. Aún más, los resultados indican que a medida que se considera un sistema más grande, con un mayor número de células, los beneficios son más notorios. A diferencia de la mayoría de los estudios existentes en la literatura que simplemente realizan una paralelización plana, en donde únicamente se limitan a dividir el trabajo entre las unidades de procesamiento de la plataforma utilizada sin considerar los accesos no uniformes a memoria, las implementaciones propuestas en este trabajo de tesis toman en cuenta aspectos específicos de las arquitecturas de hardware en donde se ejecutan.

Este trabajo de tesis se presenta organizado en cinco capítulos, de la siguiente manera:

- **Capítulo 1 – Los Autómatas Celulares como paradigma de modelado de sistemas complejos.** En este capítulo, se presenta una introducción al paradigma de modelado de los AC, así como su definición, propiedades y algunos conceptos relacionados. Así mismo, se describen brevemente algunas de sus aplicaciones, en particular, para la modelación de sistemas naturales y físicos con un comportamiento complejo.
- **Capítulo 2 – Conceptos básicos sobre el Cómputo de Alto Desempeño.** En este capítulo, se describen algunos conceptos básicos sobre el HPC, las principales arquitecturas, incluyendo algunas de sus principales características, las métricas para la evaluación del desempeño de aplicaciones paralelas, a fin de tener una base para comprender este trabajo de tesis. Así mismo, se describe una metodología para el diseño de algoritmos paralelos, con la cual se pretende fortalecer el desarrollo de aplicaciones paralelas escalables, concurrentes y con un alto desempeño.
- **Capítulo 3 – Trabajos relacionados.** En este capítulo, se analizan algunos trabajos de interés encontrados en la literatura y que se relacionan con este trabajo de tesis. Estos trabajos fueron considerados, debido a que son clasificados como entre algunos de los más relevantes, así como por ser estudios recientes en el área.

- **Capítulo 4 – Desarrollo de modelos de AC basados en arquitecturas paralelas con diseño NUMA.** En este capítulo, se presenta una propuesta para el diseño de implementaciones paralelas de modelos de AC. El objetivo general es realizar un análisis del desempeño en términos de escalabilidad y eficiencia, de implementaciones paralelas híbridas de modelos de autómatas celulares a gran escala, utilizando afinidad de procesos y afinidad de memoria diversas configuraciones del espacio de dominio; a fin de determinar sí esta estrategia no solo brinda un mejor desempeño al minimizar los tiempos de ejecución, como la mayoría de los estudios existentes en la literatura, sino también de utilizar de manera adecuada los recursos existentes. Para lograr este objetivo, se hace uso de una metodología orientada al desarrollo de programas paralelos eficientes y altamente escalables. Con la finalidad de evaluar la propuesta, se implementan un modelo de AC conocido, que se usan como casos de estudio. Finalmente, se detallan las implementaciones paralelas para el caso de estudio, así como de las optimizaciones realizadas.
- **Capítulo 5 – Resultados obtenidos.** En este capítulo, se presenta una evaluación de la propuesta para el diseño de implementaciones paralelas del modelo de AC presentada en el capítulo previo. Pare este propósito, se realizan y analizan los resultados que se obtienen de varios experimentos de las implementaciones sobre el modelo de AC bajo estudio, en varias plataformas. Con base en los resultados que se obtienen, se corrobora el cumplimiento de la hipótesis planteada en la introducción de este trabajo de tesis.
- **Conclusiones y trabajo futuro.** Finalmente, se presentan las conclusiones obtenidas en base a los resultados presentados en el capítulo anterior. Así mismo, se presentan las limitaciones del trabajo, así como algunas propuestas para ampliar en un futuro próximo este trabajo de tesis.

Capítulo 1.

Los Autómatas Celulares como paradigma de modelado de sistemas complejos.

Profundamente arraigados en la investigación fundamental en Matemáticas e Informática, los Autómatas Celulares (AC) son reconocidos como un paradigma de modelado intuitivo para los Sistemas Complejos. En los últimos años, más allá del ámbito original de las aplicaciones - Física, Informática y Matemáticas - AC también se han convertido en herramientas fundamentales en disciplinas muy diferentes como la epidemiología, la inmunología, la sociología, las finanzas y la medicina. Dado que son el centro de estudio de este trabajo de tesis, en este capítulo, se introduce la definición de los AC, sus propiedades y algunos conceptos relacionados. Así mismo, se describen brevemente algunas de sus aplicaciones, en particular, para la modelación de sistemas naturales y físicos con un comportamiento complejo.

1.1 Origen de los autómatas celulares.

Los AC fueron concebidos a principios de los 50's por los matemáticos John von Newmann y Stanisław Ulam [1], cuya motivación era la de modelar una máquina capaz de auto-reproducirse [2]. Su objetivo se enfocó en abstraer un conjunto de primitivas de interacciones locales necesarias para la evolución de formas complejas de organización esenciales para la vida. Posteriormente, a principios de los 70's, Martin Gardner escribió un artículo en su columna *Mathematical Games* de la revista *Scientific American*, el cual, documentaba el trabajo realizado por el matemático John H. Conway titulado "El juego de la vida" (GoL, por sus siglas en inglés) [3]. A diferencia del modelo realizado por von Neumann, Conway logra un resultado parecido al comportamiento real de la vida pero con un conjunto de reglas muy simple; por lo que hasta la fecha, ha recibido un amplio interés por parte de investigadores, ya que además, una de sus características más importantes es su capacidad de realizar cómputo universal, es decir, que con una configuración inicial apropiada, el GoL se puede convertir en una computadora de propósito general (máquina de Turing) [4]. Sin embargo, fue hasta mediados de los 80's que Stephen Wolfram comenzó a estudiar a gran detalle una familia de reglas de AC unidimensionales [5, 6], y mostró que aun estas reglas muy simples son capaces de emular comportamientos complejos. Desde entonces, Wolfram ha continuado demostrando la importancia de los AC en diferentes áreas. Su trabajo científico más grande y significativo sobre AC fue publicado en 2002 en su libro titulado "A New Kind of Science" [7]. Por lo que, en las últimas décadas, los AC han ganado popularidad, debido a su capacidad de lograr una serie de propiedades que surgen de la propia dinámica local a través del paso del tiempo y no desde un inicio.

1.2 Definición de los autómatas celulares.

Definición 1. *De manera informal, los AC son sistemas dinámicos, discretos tanto en el espacio como en el tiempo, formados por un gran número de elementos simples y homogéneos llamados celdas, dispuestas uniformemente generando una estructura topológica regular d-dimensional denominado enmallado; en donde cada una de sus celdas, puede representar, no simultáneamente, un número finito k de estados específicos. Dichas celdas, interactúan entre sí, con las celdas dentro de su vecindad, a través de un conjunto de reglas locales que representan la dinámica del sistema que se está modelando.*

Definición 2. Formalmente, los AC clásicos se pueden definir como la 4-tupla:

$$A = (L, S, N, f)$$

En donde:

- L (Lattice, en inglés para referencia el pseudocódigo), es el conjunto de celdas identificadas por los puntos con coordenadas enteras en un espacio Euclidiano con dimensiones $d \in Z^+$.
- S (States, en inglés), es el conjunto finito de estados de las celdas de L, formado por los k posibles valores que una celda puede tomar en un instante de tiempo (t = 0, 1, 2, …) determinado. Donde $S = \{s_1, s_2, \ldots, s_k\}$
- N (Neighborhood, en inglés), es el conjunto finito de celdas que definen la vecindad de la celda Ci, a partir de las cuales se obtiene el estado de esta celda para el tiempo t+1, según la regla de transición que se considere.
- f (function, en inglés), es la función de transición local $f: S^n \rightarrow S$ para cada celda, la cual se representa mediante un conjunto finito de reglas, entre las que destaca una regla de actualización. S^n es un conjunto finito, el cual se integra por todas las combinaciones posibles de los estados, tanto del elemento en consideración como de aquellos n-1 sitios que conforman su vecindad. El objetivo de esta función de transición es el de cambiar la configuración de una celda a cada paso de tiempo t.

Este formalismo describe a los AC homogéneos, en donde la función de transición es idéntica para todas las celdas, así como la vecindad es la misma para todas las celdas en cada paso de tiempo.

1.3 Elementos principales de los autómatas celulares.

La definición de un AC requiere mencionar sus elementos básicos:

Un espacio celular regular. La estructura topológica del enmallado depende tanto de la cantidad de dimensiones del espacio de dominio (ver Figura 1.1), como de la forma de las celdas (ver Figura 1.2), las cuales pueden tener una de múltiples formas, donde la elección de la misma, se basa en el objetivo del estudio a realizar.

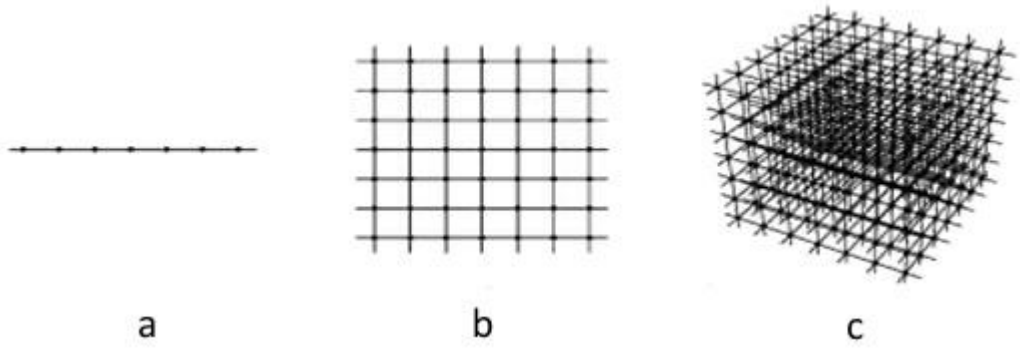


Figura 1.1. Espacio celular en 1, 2 y 3 dimensiones en el espacio celular de un AC con celda cuadrada. [7]

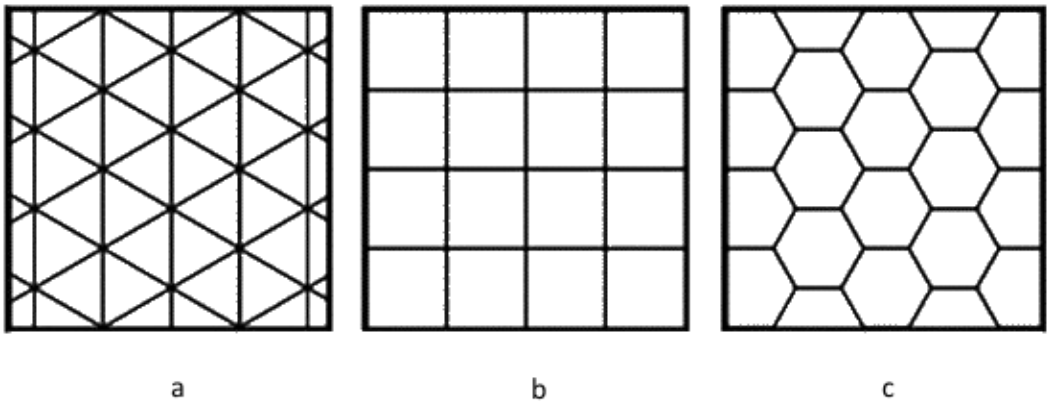


Figura 1.2. Algunas formas de celdas y la estructura topológica generada por las mismas en el espacio celular de un AC bidimensional. a) Triangular, b) Cuadrada, c) Hexagonal

Conjunto de Estados. Es finito y cada elemento o célula del espacio toma un valor de este conjunto de estados. También se denomina alfabeto. Puede ser expresado en valores o colores.

Configuración Inicial. Es la asignación inicial de un estado a cada una de las células del espacio.

Vecindades. Así mismo, esta estructura topológica induce las conexiones entre las celdas, definiendo las celdas adyacentes que son capaces de influenciar la evolución de una celda específica (usualmente incluyendo a la misma celda), las cuales se denominan su vecindad. Una vecindad no tiene restricción en tamaño o ubicación, excepto que debe ser la misma para todas las celdas en el AC.

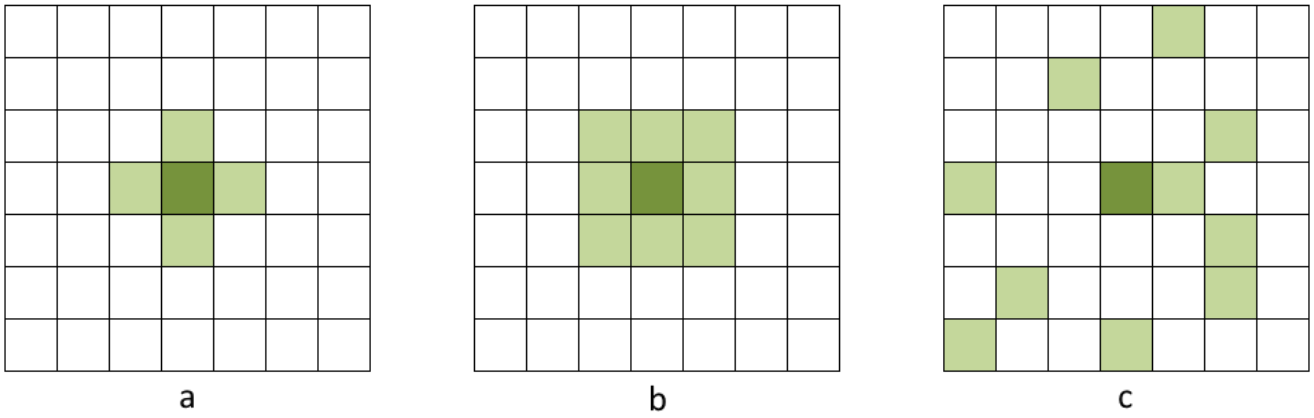


Figura 1.3. Algunos tipos de vecindades en el espacio celular de un AC con celda cuadrada.

a) Von Neumann, b) Moore, c) Aleatoria

Algunas de las vecindades más comúnmente utilizadas son: la vecindad de von Neumann (ver Figura 1.3 a), la cual consiste en una celda especifica junto con las celdas de arriba, abajo, a la derecha y a la izquierda respecto a esta (N, S, E, O); y la vecindad de Moore (ver Figura 1.3 b), que está formada por la celda especificada junto con las celdas alrededor de esta (N, S, E, O, NE, NO, SE, SO).

Función de Transición Local. La evolución de los estados de cada una de las celdas en el AC se define por un conjunto de reglas de transición. En cada paso de tiempo (generación), cada una de las celdas en el espacio de dominio utiliza la misma función de transición para actualizar su estado al siguiente tiempo, con respecto al estado actual de las celdas en su vecindad.

Condiciones de frontera. En teoría el enmallado es infinito. Sin embargo, en la práctica, el espacio de dominio de un AC siempre es acotado. En vez de tener una regla diferente para las celdas en los límites del espacio de dominio, se extiende la vecindad de estas celdas utilizando celdas virtuales que permitan completar la vecindad. Para definir los estados que se utilizarán en las celdas virtuales para completar la vecindad de las celdas al borde del espacio de dominio que utilizará la regla de evolución, se utilizan varias condiciones de frontera. La figura 1.4 muestra varios tipos de condiciones de frontera utilizadas para extender el espacio de dominio.

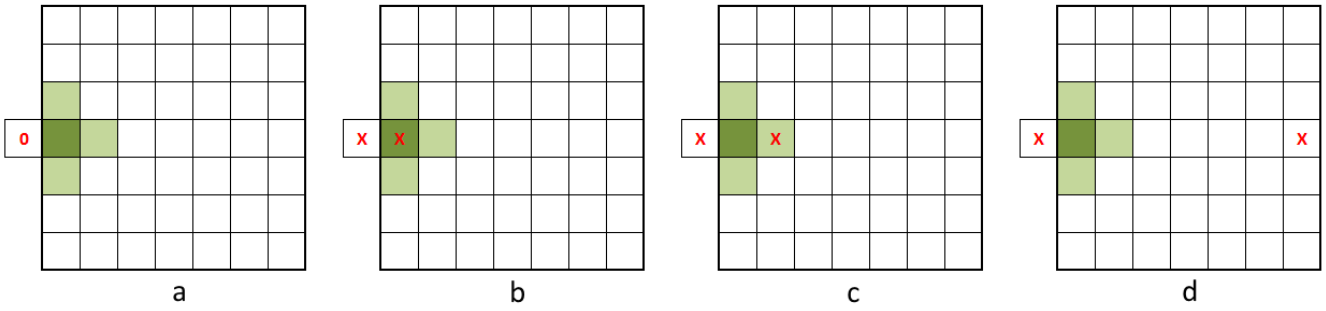


Figura 1.4. Condiciones de frontera de un AC.

a) Fija, b) Adiabática, c) Reflectiva, d) Periódica

Adicionalmente, es necesario especificar el *tipo de límite o frontera del espacio*, entre los que podemos destacar: Una condición de frontera fija (Figura 1.4 a) es definida al completar la vecindad con celdas virtuales con un estado fijo pre-asignado. Una condición de frontera adiabática (Figura 1.4 b) se obtiene al duplicar el estado de la celda en la celda virtual. En una condición de frontera reflectiva (Figura 1.4 c), el estado del vecino opuesto es replicado en la celda virtual. Una condición de frontera periódica (Figura 1.4 d) se obtiene al utilizar el estado de la celda ubicada en el extremo opuesto del espacio de dominio, en la celda virtual, formando un toroide. Dependiendo de la condición de frontera utilizada, se obtendrán comportamientos completamente diferentes del AC.

Condición inicial. El modelo requiere de datos de entrada, a través de los cuales se defina el estado inicial de cada una de las celdas, a esto se le conoce como condiciones iniciales, y ésta define el comportamiento inicial del autómata celular y, por lo tanto, afectan el comportamiento del modelo.

A pesar de la simplicidad de los AC, no es fácil analizar su comportamiento global desde el comienzo, a no ser por vía de la simulación, partiendo de un estado o configuración inicial de células y cambiando en cada instante los estados de todas ellas de forma síncrona; lo que lo hace altamente complejo y difícil de predecir.

1.4 Algunas variaciones de los autómatas celulares.

Para soportar una mayor diversidad de aplicaciones, varias extensiones y modificaciones al modelo básico han sido propuestas. Los principales cambios se refieren a la posibilidad de tener heterogeneidad temporal y espacial tanto en la función de transición y en la vecindad; asincronía en la función de transición, para que cada celda pueda escoger de manera no determinística entre cambiar su estado actual o mantenerlo en cada paso de tiempo; vecinos complejos dependientes del tiempo; funciones de transición probabilísticas y jerárquicas; o diferentes conjuntos de k estados que una celda puede tomar en un instante de tiempo dado.

1.5 Aplicaciones de los autómatas celulares.

En las últimas décadas, los AC han sido ampliamente utilizados como una de las herramientas más importantes en la modelación de sistemas complejos en áreas tan diversas que van desde física [8-10], geografía [11-13] hasta química [14-16], entre muchas otras. Esto se debe a que los AC permiten realizar cómputos únicamente a partir de información obtenida a partir de las interacciones locales entre sus elementos, exhibiendo un comportamiento emergente y de auto-organización; lo cual imita el comportamiento de un sistema complejo real, permitiendo una verdadera comprensión de las propiedades fundamentales del sistema bajo estudio [17].

Una de las aplicaciones donde los AC tienen mayor auge y utilidad por el nivel de detalle que describen el sistema, es en la modelación de tumores cancerígenos.

1.5.1. Modelo de AC para la simulación del crecimiento de tumores.

En esta dirección, diversos modelos se han propuesto a la fecha [19-28]. Estos modelos se enfocan en modelar, analizar, entender y aún más, pronosticar el desarrollo de esta enfermedad en un organismo real, mientras que con enfoques matemáticos o modelos in vivo o in vitro no es posible realizar. Por lo que son una alternativa prospera para estudiar nuevos tratamientos, analizar cuál es el intervalo óptimo de su aplicación, así como la intensidad en cada aplicación. Sin embargo, dado que estos modelos suelen ser bastante complejos en relación al número de entidades y el rango que cubren a una escala espacial y temporal, en ocasiones no es posible representar la cantidad necesaria de elementos para el modelo en un mismo equipo, debido a que sobrepasa la cantidad de memoria disponible. Aunque esto fuera posible, el cálculo de las simulaciones puede ser una tarea que consume mucho tiempo; a pesar de que los cálculos realizados por las reglas de transición de los autómatas celulares pueden ser extremadamente simples y no requieren de una gran capacidad de cómputo para poder realizarlas.

Particularmente, Ángel Monteagudo y José Santos desarrollaron en [20-24] un modelo de AC para simular el crecimiento de tumores, basados en las diferencias fenotípicas entre las células sanas y cancerígenas, descritas por Hanahan and Weinberg en el artículo titulado “The Hallmarks of Cancer” [18], en el cual, se describen seis alteraciones esenciales en la fisiología celular que dictan colectivamente el crecimiento maligno, las cuales son: autosuficiencia en las señales de crecimiento, insensibilidad a las señales inhibidoras del crecimiento (antirretroceso), evasión de la muerte celular programada (apoptosis), potencial replicativo ilimitado, angiogénesis sostenida e invasión tisular y metástasis.

Ellos utilizan un modelo de eventos discretos, similar al utilizado por Abbott et al. [18], que tiene en cuenta los principales aspectos del ciclo celular desde el punto de vista de la aplicación. Una mitosis está programada varias veces en el futuro, siendo una variable aleatoria distribuida

uniformemente entre 5 y 10 tiempos, simulando la duración variable del ciclo de vida celular (entre 15 y 24 horas). Finalmente, una malla con 10^6 sitios representa aproximadamente 0,1 mm^3 de tejido. Cada célula reside en un punto en una red cúbica y tiene un "genoma" asociado con diferentes marcas de cáncer [18].

En este modelo, cada célula tiene un “genoma” asociado con diferentes características distintivas de cáncer (hallmarks). Las alteraciones esenciales en la fisiología celular que dictan colectivamente el crecimiento maligno son (ver Tabla 1.1):

Alteración.	**Descripción.**
SG. Auto-Crecimiento (SG):	Crecimiento incluso en ausencia de señales normales. La mayoría de las células normales esperan un mensaje externo (señales de crecimiento de otras células) antes de dividirse. Las células cancerosas a menudo falsifican sus propios mensajes de crecimiento.
Ignorar la inhibición del crecimiento (IGI):	A medida que el tumor se expande, aprieta el tejido adyacente, lo que envía mensajes químicos que normalmente harían interrumpir la división celular. Las células malignas ignoran los comandos, proliferando a pesar de las señales anti-crecimiento emitidas por las células vecinas.
Evasión de la apoptosis (EA):	En las células sanas, el daño genético por encima de un nivel crítico suele activar un programa de suicidio (muerte celular programada o apoptosis). Las células cancerosas evitan este mecanismo.
Capacidad para estimular la construcción de los vasos sanguíneos (AG):	Los tumores necesitan oxígeno y nutrientes para sobrevivir, estos se obtienen a través de inducir la formación de nuevas ramas de vasos sanguíneos a partir de vasos cercanos existentes (angiogénesis).
Inmortalidad efectiva (EI):	Las células sanas pueden dividirse no más de varias veces (< 100). El limitado potencial de replicación surge porque, con la duplicación, hay una pérdida de pares de bases en los telómeros (extremos de los cromosomas que protegen las bases), por lo que cuando el ADN está desprotegido, la célula muere. Las células malignas sobreproducen la enzima telomerasa, evitando el acortamiento del telómero, por lo que estas células superar el límite reproductivo.
Capacidad para invadir otros tejidos y propagarse a otros órganos (MT):	Los cánceres generalmente se convierten en una amenaza para la vida sólo después de que de alguna manera desactivan los circuitos celulares que los confina a una parte específica del órgano en el que surgieron. Nuevos crecimientos aparecen y eventualmente interfieren con los

	sistemas vitales.
Inestabilidad genética (GI):	Representa la alta incidencia de mutaciones en las células cancerosas, lo que permite una rápida acumulación de daño genético. Es una característica que permite el cáncer, ya que, aunque no es necesario en la progresión de la neoplasia al cáncer, hace que la progresión mucho más probable. La simulación implica que las células con este factor aumentarán su tasa de mutación.

Tabla 1.1. Alteraciones esenciales en la fisiología celular que dictan colectivamente el crecimiento maligno.

Cada genoma de la célula indica si algunos de estos signos se activan como consecuencia de las mutaciones. Además, se consideró una "inestabilidad genética", que explica la alta incidencia de mutaciones en las células cancerosas, lo que permite una rápida acumulación de daño genético. La simulación implica que las células con este factor aumentarán su tasa de mutación. Dependiendo de la activación de los sellos en cada una de las celdas, el sistema puede evolucionar a dinámicas diferentes. Metástasis (MT) y angiogénesis (AG) no son consideradas, ya que este trabajo se enfoca en la fase avascular de la tumorigénesis. Por lo tanto, cada célula tiene su genoma que consiste en cinco características (SG, IGI, EA, EI, GI), más algunos parámetros particulares de cada célula (t_l, e, m, i, g, a) que se especifican en la siguiente tabla (ver Tabla 1.2):

Parámetro.	**Valor predeterminado**	**Descripción.**
Longitud del telómero (t_l)	100	Longitud inicial del telómero. Cada vez que una célula se divide, la longitud se decrementa en una unidad. Cuando llega a 0, la célula muere, a menos que el sello “Inmortalidad Efectiva” (EI) esté activo.
Evade apoptosis (e)	10	Una célula con n marcas mutadas tiene una probabilidad adicional de morir cada ciclo celular, a menos que la marca "Evade apoptosis" (EA) esté activada.
Tasa de mutación base (m)	100,000	Cada gen (sello) está mutado (cuando la célula se divide) con una probabilidad de 1/m de mutación.
Inestabilidad genética (i)	100	Hay un aumento de la tasa de mutación de base por un factor de i para las células con esta mutación (GI).

Ignora (*g*)	10	Las células con el sello "Ignorar inhibición del crecimiento" (IGI) activado tienen una probabilidad 1/*g* de matar a un vecino para dejar espacio para la mitosis.
Muerte celular aleatoria (*a*)	1,000	En cada ciclo celular cada célula tiene una probabilidad de 1/*a* de muerte por varias causas.

Tabla 1.2. Parámetros predeterminados asociados con las características del cáncer, utilizados por los autores.

Los parámetros longitud de telómero y tasa de mutación base pueden cambiar sus valores en una celda en particular a lo largo del tiempo. El genoma de la célula es heredado por las células hijas cuando se produce una división mitótica.

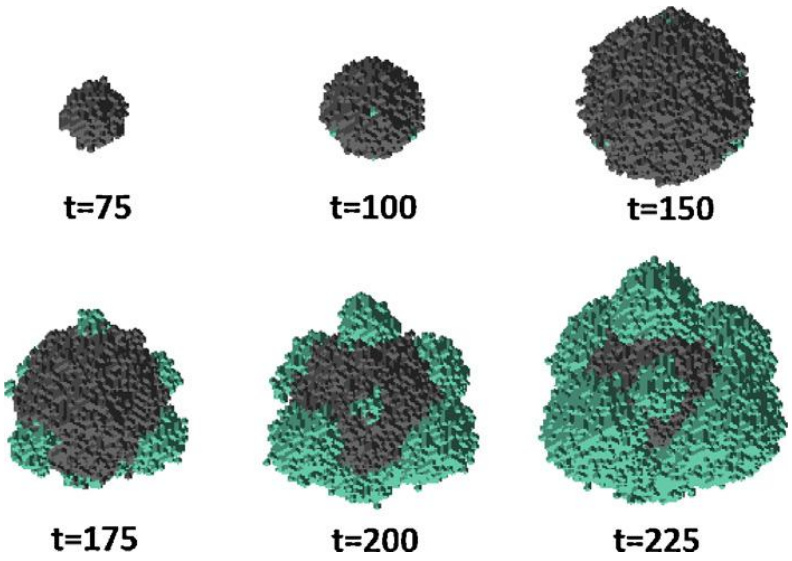


Figura 1.6. Sistema celular en diferentes pasos de tiempo. Células sanas (gris), células cancerosas (verde) [23]

En la simulación del ciclo de vida de la célula, la mayoría de los elementos no cambian observablemente cada paso del tiempo. Los únicos cambios observables en las células son la apoptosis y la mitosis (ver Figura 1.6). En un tejido, sólo una fracción de todas las células está sufriendo tales transiciones en un momento dado.

En lo siguiente, nos enfocamos a la descripción de una de estas aplicaciones de los autómatas celulares, la cual será de gran utilidad en la evaluación de la implementación que se propone en este trabajo de tesis: el juego de la vida.

1.5.2 El juego de la vida.

El “Juego de la Vida” [3] (GoL) es el AC más conocido, formulado por el matemático británico John H. Conway en 1970, a través del cual se simulan celdas vivas y muertas en una malla bidimensional, en donde cada celda (i, j) puede tener uno de 2 estados (vivo o muerto).

El estado de cada una de las celdas cambiará en el siguiente paso de tiempo (de manera síncrona), dependiendo del estado actual de las celdas en su vecindad (8 vecinos al utilizar una vecindad de Moore de radio 1) y del de ella misma, de acuerdo a un conjunto definido de reglas simples, que se enumeran a continuación:

- Una celda viva con menos de 2 vecinos vivos, muere (por soledad).
- Una celda viva con 2 o 3 vecinos vivos, permanece viva para la siguiente generación (por sobrevivencia).
- Una celda viva con más de 3 vecinos vivos, muere (por sobrepoblación).
- Una celda muerta con exactamente 3 vecinos vivos, nace (es decir, al siguiente paso de tiempo estará viva, por reproducción).

La configuración inicial constituye la primera generación del sistema. La segunda generación se forma al aplicar las reglas mencionadas anteriormente de manera simultánea a todas las celdas en la generación anterior. Las reglas continúan aplicándose repetidamente para crear generaciones posteriores. Teóricamente, el enmallado del GoL es infinito, pero en la práctica, dado que las computadoras no pueden representar estructuras de este tipo, se utiliza una malla finita con una condición de frontera periódica. Lo cual significa que, para completar la vecindad de las celdas al borde del enmallado, se considerarán como sus vecinos a las celdas en la posición opuesta del enmallado.

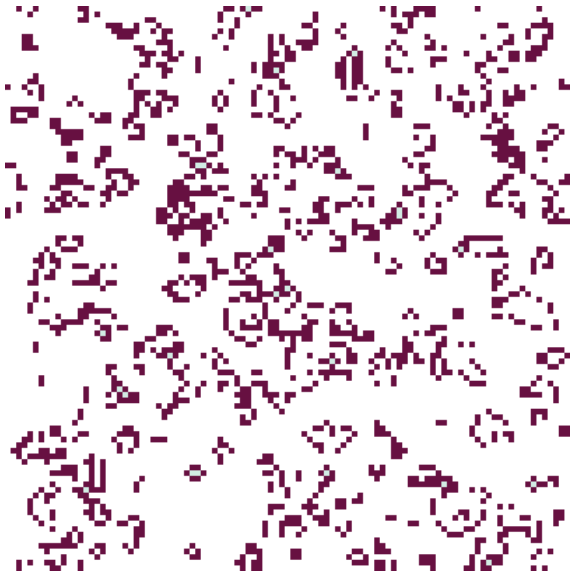


Figura 1.5. Configuración de un estado de tiempo del GoL.

A pesar de su simplicidad, el GoL es un modelo que genera patrones interesantes como se muestran en la Figura 1.5. Este modelo es importante, ya que provee los fundamentos básicos para crear simulaciones que muestran las características y comportamientos reproductivos de sistemas biológicos.

En este trabajo de tesis, se presenta una propuesta que permita implementar eficientemente modelos de AC para sistemas a gran escala, a través del uso del cómputo de alto desempeño (HPC, por sus siglas en inglés). Para evaluar la propuesta, se utilizan como caso de estudio el juego de la vida escrito en esta sección.

En el siguiente capítulo, se introducen conceptos relacionados al HPC, necesarios para el buen entendimiento de este trabajo.

Capítulo 2.

Conceptos básicos sobre el cómputo de alto desempeño.

El objetivo principal del cómputo de alto desempeño, aquí referido como HPC, es acelerar la ejecución de un código para permitir la simulación de sistemas con resultados más rápidos, con mayor precisión y mucho más grandes, de lo que es posible en equipos secuenciales.

En este capítulo, se describen algunos conceptos básicos sobre el HPC, las principales arquitecturas, incluyendo algunas de sus principales características, las métricas para la evaluación del desempeño de aplicaciones paralelas, a fin de tener una base para comprender este trabajo de tesis. Así mismo, se describe una metodología para el diseño de algoritmos paralelos, con la cual se pretende fortalecer el desarrollo de aplicaciones paralelas escalables, concurrentes y con un alto desempeño.

2.1 Introducción.

La computación paralela es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente [36], operando sobre el principio de que problemas grandes, a menudo se pueden dividir en unos más pequeños, que luego son resueltos simultáneamente (en paralelo). Así, la computación paralela, se aplica principalmente para permitir la simulación de sistemas con resultados más rápidos, con mayor precisión y mucho más grandes, de lo que es posible en equipos secuenciales. Para ello, existen varias técnicas diferentes de paralelismo: a nivel de bit, donde el aumento del tamaño de la palabra reduce el número de instrucciones que el procesador debe ejecutar para realizar una operación en variables cuyos tamaños son mayores que la longitud de la palabra; a nivel de instrucción, donde se realiza un pipeline de instrucciones de varias etapas; a nivel de datos, en la cual se realiza el mismo cálculo en distintos o mismos grupos de datos; y a nivel de tareas, donde cálculos completamente diferentes se pueden realizar en cualquier conjunto igual o diferente de datos.

Aunque el paralelismo ha sido utilizado desde hace muchos años, el interés por este se ha incrementado en las últimas décadas, debido a las limitaciones físicas que impiden incrementar la frecuencia del reloj; tales como el consumo energético y por consiguiente producción de calor debido a la fuga de corriente, lo que requiere de un gran esfuerzo adicional para realizar el enfriamiento, así como que no es posible reducir los tiempos de acceso a la memoria al mismo ritmo que como se incrementa la frecuencia del reloj.

Actualmente, gracias al HPC es posible reemplazar, acelerar o ampliar experimentos que por medios convencionales podrían resultar peligrosos, costosos, o incluso imposibles de realizar. No obstante, las aplicaciones paralelas son más difíciles de implementar que las secuenciales, porque la concurrencia da lugar a posibles errores que es necesario considerar. Además, la comunicación y

sincronización entre los diferentes procesos son algunos de los más grandes obstáculos que es necesario manejar para obtener un buen rendimiento. Es por ello que, realizar simulaciones eficientes, con un alto grado de escalabilidad, y que brinden un alto desempeño, no es trivial.

En las últimas décadas, la investigación del cómputo de alto rendimiento incluyó nuevos desarrollos en tecnologías paralelas de hardware y software, cada una con características específicas y ventajas que las hacen especiales para resolver cierto tipo de problemas; las cuales se clasifican de diferentes maneras: algunas en base al número de instrucciones concurrentes y en los flujos de datos disponibles en la arquitectura, otras según el nivel de paralelismo que admite el hardware, o en base a su organización de memoria.

A continuación, se presentan algunas clasificaciones, así como algunas de las plataformas convencionales actuales de interés para este trabajo de tesis, de las cuales es necesario conocer las características que estas tecnologías brindan.

2.2 Arquitecturas de cómputo paralelas.

Michael J. Flynn propuso en 1972 una de las primeras clasificaciones de las arquitecturas de cómputo paralelas conocida como la *taxonomía de Flynn* [28], la cual clasifica las arquitecturas de cómputo en base al flujo de instrucciones que es ejecutado por una plataforma y en la secuencia de datos disponibles en la arquitectura. Si bien, esta clasificación es antigua, ha sobrevivido debido a su fácil comprensibilidad y brinda una primera aproximación clara:

SIMD - Single Instruction, Multiple Data. Un flujo de instrucciones único, ya sea en un solo procesador (núcleo), proporciona paralelismo operando en múltiples flujos de datos simultáneamente. Ejemplos de SIMD son las unidades de procesamiento gráfico (GPU) y las capacidades SIMD de los microprocesadores superescalares modernos.

MIMD - Multiple Instruction, Multiple Data. Varias secuencias de instrucciones en varios procesadores (núcleos) operan en diferentes elementos de datos simultáneamente. La memoria compartida y los equipos paralelos de memoria distribuida descritos en este capítulo son ejemplos típicos del paradigma MIMD.

Existen dos categorías más, denominadas SISD (Single Instruction, Single Data) y MISD (Multiple Instruction, Single Data), la primera describiendo la ejecución convencional, no paralela, de un solo procesador; mientras que la última no se considera un paradigma útil en la práctica.

2.3 Arquitecturas de memoria compartida.

Este tipo de arquitectura consiste en un conjunto de Unidades de Procesamiento (UP) que comparten una única memoria global. Físicamente, la memoria global está constituida por varios módulos de memoria independientes que brindan un espacio de direcciones común, el cual puede ser

accedido por todas las UP del sistema. Los datos pueden ser intercambiados entre las UP, a través de la memoria, al escribir y leer en variables compartidas.

2.3.1 Plataformas multicore.

Estos procesadores surgieron como una solución alternativa para incrementar el rendimiento, ya que el enfoque tradicional fue acotado, debido a las limitaciones físicas en la fabricación de procesadores mencionadas anteriormente. Debido a estas y otras cuestiones, en vez de incrementar la complejidad de la organización interna de un procesador, los principales fabricantes de procesadores decidieron utilizar un enfoque diferente, integrando múltiples UP independientes en un mismo procesador.

Particularmente, un procesador multicore consiste en un conjunto de UP conocidas como núcleos, en donde cada uno de ellos tiene un control independiente y pueden acceder a la misma memoria de manera concurrente. Estos procesadores permiten ejecutar de manera simultánea múltiples instrucciones en los distintos núcleos, aumentando la velocidad global de los programas susceptibles a ser paralelizados. Estos procesadores contienen múltiples niveles de memorias cache (algunas independientes entre los núcleos y otras compartidas), en donde cada nivel brinda diferentes capacidades de almacenamiento con diferentes velocidades de acceso, que permiten reducir la latencia generada por el problema de la diferencia de velocidad entre el procesador y la memoria. Actualmente, existen diferentes diseños de procesadores multicore con diferente número de núcleos, estructura y tamaño de las cache, tiempos de acceso a las cache, etc.; uno de estos diseños se presenta en la Figura 2.1.

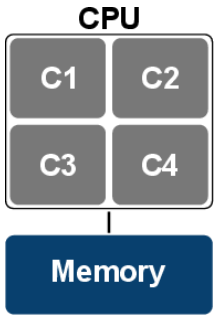


Figura 2.1. Diagrama esquemático de procesador multicore.

Para el sistema operativo, cada núcleo representa un procesador lógico con recursos de ejecución independientes. Así, cada núcleo es controlado de manera independiente y el sistema operativo puede asignar diferentes aplicaciones a los diferentes núcleos. Utilizando técnicas de programación paralela, es posible ejecutar aplicaciones computacionalmente intensivas de manera paralela en un conjunto de núcleos, reduciendo el tiempo de ejecución comparado con la ejecución en un mismo núcleo u obteniendo resultados más precisos que llevando a cabo una mayor cantidad de cálculos en el caso secuencial.

En 2017, Intel lanzó la 7ª generación del procesador Intel i7, el cual cuenta con hasta 4 núcleos capaz de 8 hebras simultaneas, así como una Serie-X i9, con hasta 18 núcleos capaz de 36 hebras simultaneas; mientras que, por su parte, AMD lanzó el procesador AMD Ryzer 7 1800X, el cual cuenta con 8 núcleos (16 hebras). No obstante, Intel y AMD, cuentan adicionalmente con procesadores de hasta 32 núcleos, cada uno, para la industria. Por lo que, de acuerdo a la ley de Moore, es posible predecir que en 2025 un procesador típico podría consistir de docenas o incluso hasta cientos de núcleos, brindando un gran potencial de rendimiento.

2.3.2 Multithreading simultáneo.

El multithreading simultáneo (SMT) permite que varios hilos compartan las unidades funcionales del procesador y se ejecuten simultáneamente. Para permitir el SMT, se realiza una copia de los recursos necesarios para almacenar el estado de cada hilo (esto incluye el contador de programa, los registros de usuario y de control, así como el controlador de interrupción junto con sus registros) [30]. Con esta replicación, el procesador aparece al sistema operativo y al programa de usuario como un conjunto de procesadores lógicos a los que se pueden asignar procesos para su ejecución. Cada procesador lógico almacena su estado de procesador en un recurso de procesador separado. Esto evita la sobrecarga para guardar y restaurar los estados del procesador al cambiar a otro procesador lógico. Todos los demás recursos del chip del procesador como cachés, sistema de bus y unidades de función y control son compartidos por los procesadores lógicos. Por lo tanto, la implementación de SMT sólo conduce a un pequeño aumento en el tamaño del chip. Cuando un procesador lógico debe esperar por un evento, los recursos pueden ser asignados a otro procesador lógico. Esto conduce a un uso continuo de los recursos desde el punto de vista del procesador [30].

Entre algunos ejemplos de procesadores que soportan SMT están los procesadores Intel Core i3, i5 e i7 (2 procesadores lógicos), los procesadores IBM Power7 (4 procesadores lógicos) y el procesador Sun/Oracle T4 (8 procesadores lógicos) [32].

2.3.3 Plataformas multiprocesador.

Inicialmente, los sistemas multiprocesador fueron diseñados para que cada procesador accediera a cualquier localidad de memoria en la misma cantidad de tiempo que cualquier otro en el sistema, este tipo de arquitectura se conoce como multiprocesador simétrico (SMP). Los sistemas SMP acceden a la memoria compartida utilizando un bus compartido común [31], como se muestra en la Figura 2.2. La principal desventaja y cuello de botella de escalar la arquitectura SMP, es el ancho de banda del bus común, ya que a medida que aumenta el número de procesadores, aumenta la demanda del bus común. De acuerdo con [33,34], el límite máximo para una configuración SMP basada en bus está entre 32 y 64 procesadores.

Para superar los problemas de escalabilidad de la arquitectura SMP, se desarrolló el acceso no uniforme a memoria (NUMA). Una plataforma con diseño NUMA consiste en un conjunto de

procesadores, en donde sus unidades de procesamiento se agrupan en diferentes dominios denominados nodos NUMA o Dominios Locales (LD) [29], en donde cada uno de ellos tienen acceso a toda la memoria disponible, al igual que en la arquitectura SMP; pero a diferencia de esta, cada LD se encuentra directamente conectado a una porción de la memoria física disponible, así como a otros dispositivos de E/S (formando grupos de sistemas SMP). Normalmente, cada LD es representado por un procesador físico, sin embargo, existen arquitecturas con más de un LD por procesador físico.

Como se puede apreciar en la Figura 2.3, estos procesadores se encuentran interconectados por una red especializada de alta velocidad (Hyper Transport (HT) en arquitecturas AMD y Quick Path Interconnect (QPI) en arquitecturas Intel). Como consecuencia, el tiempo para acceder a los datos se condiciona por la distancia entre el procesador y el banco de memoria en donde los datos fueron colocados. Los accesos remotos conducen a una mayor latencia, generando problemas de desbalanceo de carga y tiempos de espera por sincronización, contención de acceso a memoria, etc. Por lo que es importante minimizar los accesos remotos a memoria y garantizar un buen desempeño de las aplicaciones al asegurar una afinidad de procesos y memoria, generando una alta localidad. Cabe mencionar que, ignorar este diseño en el desarrollo de aplicaciones acotadas por la memoria, puede generar un desempeño deficiente, así como un mal aprovechamiento de los recursos de cómputo disponibles.

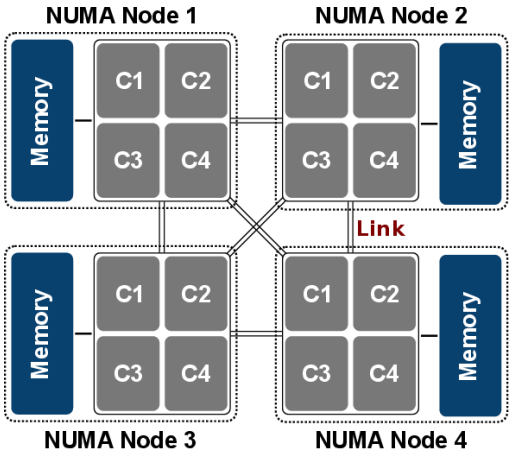


Figura 2.3. Diagrama esquemático de una plataforma multiprocesador con diseño NUMA.

A diferencia de la arquitectura SMP, la arquitectura NUMA ofrece ventajas de escalabilidad y rendimiento conforme se incrementa el número de procesadores. A su vez, cada procesador puede tener más de una UP tanto física como lógica.

2.4 Arquitecturas de memoria distribuida.

Consisten de un conjunto de elementos de procesamiento independiente (llamados nodos de procesamiento) y de una red de interconexión la cual conecta los nodos y permite la transferencia de

datos entre los mismos. Existen varias plataformas dentro de esta categoría, en donde los llamados clusters son la más conocida.

2.4.1 Cluster de computadoras.

Un cluster está constituido por un conjunto de computadoras, en donde cada una de ellas maneja su memoria de manera independiente. Estas computadoras se encuentran interconectadas, a través de una red de alta velocidad (FCS (Fibre Channel Standard), SCI (Scalable Coherent Interface), Switched Gigabit Ethernet, Myrinet, o InfiniBand) [30]. Las plataformas de cómputo presentes en los nodos de procesamiento que constituyen los clusters modernos cuentan con computadoras multiprocesador y multicore con memoria compartida. En la figura 2.4, se muestra el ejemplo de un cluster con nodos multiprocesador, que utilizan una red de interconexión para comunicarse entre sí.

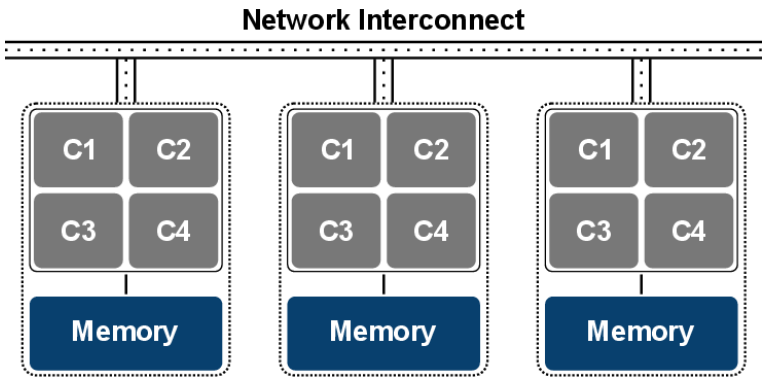


Figura 2.4. Diagrama de arquitectura con memoria distribuida.

Un modelo de programación natural de las arquitecturas de memoria distribuida es el modelo de paso de mensajes, el cual es soportado por librerías de comunicación como MPI o PVM. Estas librerías se basan frecuentemente en protocolos standard como TCP/IP.

2.5 Métricas para la evaluación del desempeño.

En los sistemas paralelos, uno de los principales problemas es la degradación del rendimiento. Idealmente, el rendimiento a partir de la paralelización crecería de manera lineal; por ejemplo, cada vez que se duplicara el número de elementos de procesamiento se reduciría a la mitad el tiempo de ejecución. Sin embargo, esto no sucede en la realidad y muy pocos algoritmos paralelos logran una aceleración óptima. La mayoría tienen una aceleración casi lineal para un pequeño número de elementos de procesamiento, posteriormente pasa a ser constante, o en el peor caso, decrece. Esto se debe principalmente a la comunicación excesiva entre los procesos, lo que provoca un efecto de saturación del sistema y a su vez, degrada de manera automática el rendimiento. Es por ello, que analizar el desempeño de un programa paralelo es importante, ya que permite evaluar la plataforma de

hardware, determinar la efectividad y escalabilidad del algoritmo, y evaluar los beneficios del paralelismo. Para facilitar el desarrollo y análisis de los programas que se ejecutan en paralelo, se analizan algunas métricas para analizar el desempeño, las cuales se describen a continuación.

2.5.1 Tiempo de ejecución (Wallclock time).

Se considera como la medida principal de desempeño que un programa es capaz de exhibir. Éste se define como el tiempo transcurrido entre el inicio y fin de una ejecución. En el caso de los programas paralelos, el tiempo de ejecución T_p, es el tiempo que tarda desde que inicia el primer proceso, hasta que termina el último proceso en ejecución. En este caso de manera intrínseca, se toman en cuenta el tiempo de procesamiento, el tiempo de inactividad y el tiempo de comunicación. Debido al no determinismo del ambiente de ejecución de los programas paralelos, si estos se ejecutan varias veces, es poco probable que se generen exactamente los mismos tiempos de ejecución. Por lo que comúnmente, el tiempo de ejecución de un programa, se obtiene a partir del promedio de un conjunto de ejecuciones realizadas en una misma plataforma.

2.5.2 Costo.

Cuantifica la cantidad total de trabajo que realiza cada uno de los procesadores que participan en la ejecución de un programa [29]. El costo de un programa paralelo que tiene tiempo de ejecución $T_p\ (n)$, con tamaño de entrada n y ejecutado en p procesadores se expresa como:

$$C_p\ (n) = p \cdot T_p\ (n)$$

2.5.3 Aceleración (Speedup).

Es una medida que indica la ganancia en velocidad al paralelizar una aplicación, en comparación con la implementación secuencial de la misma [29]. La aceleración $S_p\ (n)$ de un programa paralelo con tiempo de ejecución paralelo $T_p\ (n)$ se define como:

$$S_p\ (n) = T_s(n) / T_p\ (n)$$

donde p representa al número de procesadores utilizados para resolver un problema de tamaño n, y $T_s(n)$ es el tiempo de ejecución de la mejor implementación secuencial para resolver el mismo problema (aceleración absoluta).

2.5.4 Eficiencia.

Mide la porción útil del trabajo total realizado por n procesadores y puede expresarse como:

$$E_p\ (n) = S_p(n) / p = T_s(n) / p.T_p(n)$$

en donde $T_s(n)$ es el tiempo de ejecución secuencial de la mejor implementación secuencial y T_p (n) es el tiempo de ejecución paralelo en *p* procesadores. La eficiencia indica que tan bien se están utilizando los recursos computacionales del sistema. Un programa con aceleración lineal $S_p(n) = p$ tiene una eficiencia $E_p\ (n) = 1$ [29].

2.5.5 Ley de Amdahal.

En 1967, Gene Amdahl desarrolló una ley conocida como la Ley de Amdahl [34], la cual determina la aceleración potencial de un algoritmo en una plataforma paralela. Esta ley supone que todo el problema es de tamaño fijo, por lo que la cantidad total de trabajo que se hará en paralelo también es independiente de la cantidad de recursos de cómputo paralelos que se utilicen. Además, señala que la porción más pequeña que no pueda paralelizarse de un programa limita la aceleración que se logra con la paralelización, es decir, la parte secuencial del programa no cambia con respecto a la cantidad de procesadores utilizados, mientras que la porción paralela se ejecuta equitativamente por los *P* procesadores reduciéndose el tiempo de ejecución para esta parte como lo muestra la figura 2.5.

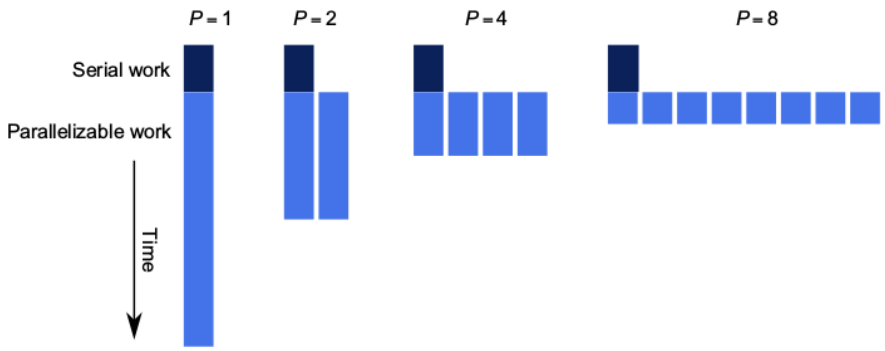


Figura 2.5. La aceleración está limitada por la sección de trabajo secuencial [36].

El tiempo de ejecución de un programa paralelo está compuesto por el tiempo de ejecución de la fracción secuencial *f* (en donde, $0 \leq f \leq 1$), y del tiempo de ejecución de la fracción paralela $(1 - f) / p$ [29]. Asumiendo que se utiliza el mejor programa secuencial y que la parte paralela del programa puede ser perfectamente paralelizada, de acuerdo a la Ley de Amdahl, la máxima aceleración alcanzable es [36]:

$$S_p\ (n) \ = \ T_s(n) / [f * T_s(n) + ((1 - f) / p) * T_s\ (n)] \ = \ 1 / (f + (1 - f) / p) \ \leq \ 1 / f$$

Si la parte secuencial *f* del programa abarca el 15% del tiempo de ejecución, se puede obtener un límite superior a no más de 6.67x de aceleración, independientemente de la cantidad de unidades de procesamiento que se utilicen.

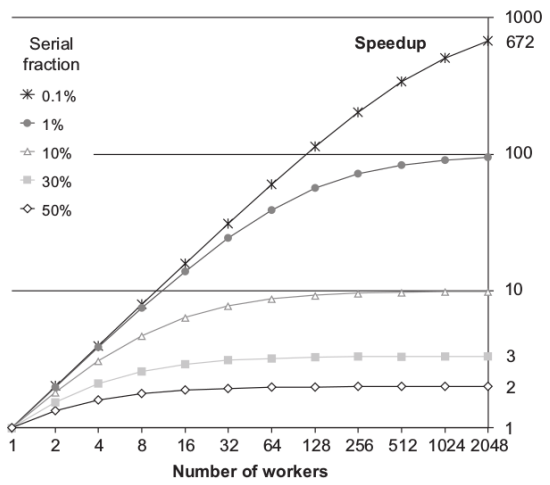


Figura 2.6. Aceleración – La escalabilidad de la paralelización está limitada por la fracción serial de la carga de trabajo [36].

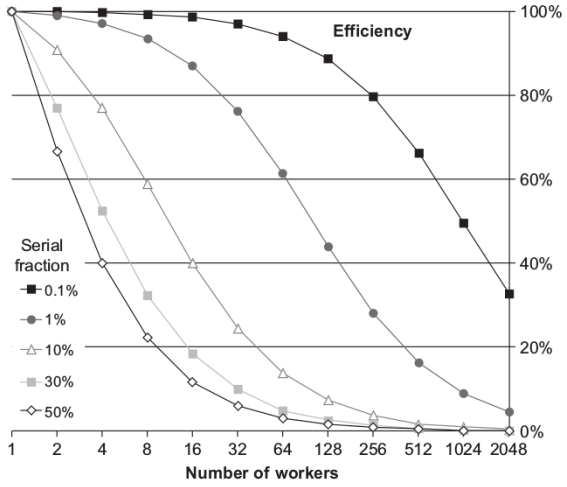


Figura 2.7. Eficiencia – Aun cuando mayores aceleraciones son posibles, la eficiencia puede fácilmente llegar a ser deficiente [36].

La Figura 2.6 muestra la cota de aceleración para varios valores de *f* y *p*. Aún con *f* = 0.1 % y *p* = 2048, la aceleración de un programa está limitada a 672x [36]. Por su parte, la figura 2.7 muestra el uso ineficiente de los recursos de hardware paralelos para las diferentes fracciones seriales.

2.5.6 Ley de Gustafson.

En 1988, John L. Gustafson y Edwin H. Barsis enuncian la actualmente conocida ley de Gustafson [35]. Como se observa en la figura 2.8, esta ley toma en consideración un incremento en el tamaño de los datos en proporción al incremento en el número de recursos paralelos y calcula la máxima aceleración de la aplicación. Gustafson y Barsis observaron que conforme el tamaño del problema crece, el trabajo requerido para la parte paralela del problema frecuentemente crece más rápido que la parte secuencial, por consiguiente, la fracción serial decrece y la aceleración mejora.

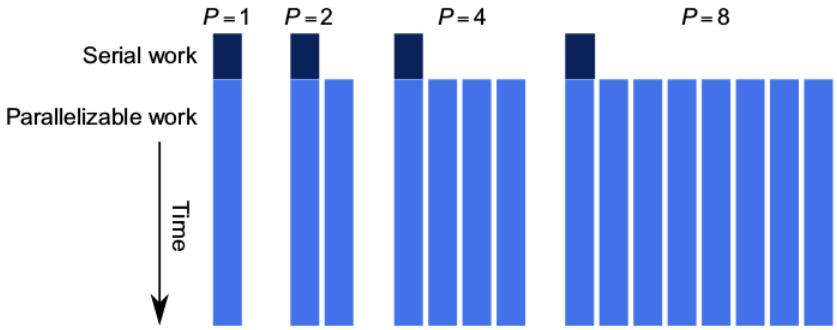


Figura 2.8. Ley de Gustafson-Barsis . Sí el tamaño del problema se incrementa con respecto a p mientras la porción

serial crece lentamente o permanece fija, la aceleración crece conforme se agregan recursos [36].

Mientras que la ley de Amdahl predice la máxima aceleración que se puede alcanzar al paralelizar un código en serie, la ley de Gustafson-Barsis se utiliza para calcular la aceleración de un código paralelo existente. La fórmula para la aceleración escalada es [37]:

$$S_p(n) \leq p + (1 - p)\, s$$

en donde p es el número de procesadores, y s es el porcentaje de tiempo que la aplicación paralela gasta en la ejecución serial. Por ejemplo, sí el tiempo de ejecución total de una aplicación paralela es de 1040 segundos en 32 procesadores, de los cuales 14 segundos son de la ejecución secuencial en 1 de esos 32 procesadores, la aceleración de esta aplicación sería [37]:

$$S_p(n) \leq p + (1 - p)\, s = 32 + (1 - 32)(0.0135) = 32 - 0.4173 = 31.5826$$

Tanto la ley de Gustafson como la ley de Amdahl asumen que el tiempo de funcionamiento de la parte secuencial del programa es independiente del número de procesadores. Sin embargo, a diferencia de la ley de Amdahl, la ley de Gustafson supone que la cantidad total de trabajo que se hará en paralelo varía linealmente con el número de procesadores. Ambas leyes no se contradicen, ya que utilizan razonamientos distintos, mientras que una se refiere a que un programa corra más rápido con la misma carga de trabajo, la otra se refiere a que un programa corra en el mismo tiempo con una carga de trabajo más grande.

2.5.7 Escalabilidad.

La escalabilidad es una medida que describe si se puede alcanzar una mejora del rendimiento que sea proporcional al número de procesadores empleados. La escalabilidad depende de varias propiedades de un algoritmo y su ejecución paralela [29]. Un sistema se dice que es escalable para un determinado rango de procesadores [1... n], si la eficiencia E(n) del sistema se mantiene constante y en todo momento por encima de un factor 0.5. Normalmente, todos los sistemas tienen un determinado número de procesadores a partir del cual la eficiencia empieza a disminuir considerablemente. Un sistema es más escalable que otro si este número de procesadores, a partir del cual la eficiencia disminuye, es menor que el otro.

2.6 Metodología para el diseño de algoritmos paralelos.

Existen muchas metodologías para el diseño de aplicaciones paralelas propuestas en la literatura. El objetivo de utilizar una de ellas es la de contar con una guía que contemple las implicaciones básicas necesarias para el desarrollo de implementaciones paralelas, independientemente de la arquitectura o modelo de programación que se esté utilizando.

Ian Foster sugiere en [38], que el diseño de un programa paralelo debe partir de una solución algorítmica existente (posiblemente secuencial) a un problema computacional, particionando el mismo en muchas pequeñas tareas e identificando las dependencias entre ellas (comunicación y sincronización), para lo cual se deben seleccionar estructuras adecuadas. Estas dos primeras fases de diseño, partición y comunicación, se utilizan para un modelo que no pone ninguna restricción en el número de procesadores. Además, la granularidad de la tarea debe ser lo más fina posible para no restringir artificialmente las fases de diseño posteriores. Como resultado, se logra un algoritmo paralelo (más o menos) escalable en un modelo de programación abstracto que es en gran medida independiente de una arquitectura paralela en particular. A continuación, se incrementa la granularidad aglomerando los elementos particionados en tareas para reducir la comunicación, preservando los canales de comunicación entre tareas dentro de la vecindad definida y eliminando comunicaciones innecesarias para aquellas celdas dentro de una misma tarea. Posteriormente, las tareas son asignadas a las unidades de procesamiento, para ser ejecutados. Esta asignación se puede especificar de manera estática o se puede determinar en tiempo de ejecución, con la finalidad de equilibrar la carga y reducir aún más los costos de comunicación. Estas dos últimas etapas, la aglomeración y el mapeo, son más dependientes de la plataforma de hardware; ya que requieren utilizar información sobre la estructura topológica de la plataforma de hardware, en donde se ejecuta la aplicación, para optimizar el rendimiento.

En el siguiente capítulo, se analizan algunos trabajos de interés, relacionados con este trabajo de tesis. Algunos de ellos tratan sobre implementaciones de modelos de AC en plataformas paralelas, mientras que otros abordan el problema de la localidad.

Capítulo 3.

Trabajos relacionados.

En este capítulo, se analizan algunos trabajos de interés encontrados en la literatura y que se relacionan con este trabajo de tesis. Estos trabajos fueron considerados, debido a que son clasificados como entre algunos de los más relevantes, así como por ser estudios recientes en el área.

3.1 Introducción

Aun cuando los cálculos realizados por las reglas de transición de los AC para modelar la dinámica de los sistemas que simulan pueden ser extremadamente simples y no requieren de una gran capacidad de cómputo para poder realizarlos, cuando se modelan sistemas complejos para aplicaciones reales con un gran número de entidades esto no es así. En estos casos, el número de iteraciones entre entidades en el tiempo y el espacio, se incrementa ampliamente y, por lo tanto, el número de operaciones derivadas. Particularmente, el cálculo de las simulaciones derivadas del incremento de las operaciones, puede ser una tarea que puede llegar a consumir mucho tiempo de cómputo. Por lo que recientemente, ha surgido un interés por el desarrollo de implementaciones de modelos de AC mediante el uso de arquitecturas paralelas [41-56], que muestren resultados positivos al procesar grandes cantidades de datos en un menor tiempo en comparación con sus contrapartes secuenciales, o debido a que permiten manejar grandes espacios de dominio, que sería imposible manejar en un solo equipo.

3.2 Trabajos relacionados

Diversos trabajos se han realizado, orientados al desarrollo de implementaciones paralelas de AC en diferentes arquitecturas, como clusters [47,53,56], procesadores multicore y manycore [43,47,48,53,55], GPUs [41,43,45,46,49,55], FPGAs [41,44,49,50], etc., y su combinación [46,53]. En el trabajo realizado por Rybacky [55], se realiza una comparación de varias implementaciones de diferentes modelos de AC, evaluados en arquitecturas multicore y GPU. Ellos encuentran que en la mayoría de los casos los algoritmos basados en GPU superan a los algoritmos basados en multicore, pero existen algunos problemas específicos donde ocurre lo

contrario; por lo que concluyen que a pesar de que los GPU son una buena alternativa para el desarrollo de implementaciones paralelas de AC, su utilidad depende del modelo a ser simulado. Por otra parte, Kalgin [54] presenta un estudio comparativo de varias implementaciones de modelos de AC asíncronos para simular un proceso químico en varias arquitecturas paralelas (multicore, cluster y GPU). En ese trabajo se estudiaron diversas técnicas para acelerar la simulación del juego de la vida con base en CUDA, logrando una implementación con tiempos de cómputo mejorados. se concluye que la eficiencia de algunos de los algoritmos es alta en determinadas arquitecturas paralelas. Además, se sugiere como mejoras futuras enfocarse en reducir los costos de las comunicaciones. Bezbradica et al. [51] realizaron un estudio sobre estrategias de paralelización de autómatas celulares de gran escala para una aplicación farmacéutica. Para ello utilizan implementaciones de memoria compartida desarrolladas con el API OpenMP, de memoria distribuida con la librería de paso de mensajes MPI y un híbrido de ambas implementaciones. Los resultados muestran que el enfoque híbrido ofrece un mejor rendimiento, seguido cercanamente de la solución MPI pura. Adicionalmente, sugieren realizar futuras mejoras al experimentar con diferentes métodos de balanceo de carga.

3.2.1. El trabajo de Millán

Posteriormente, Millán et al. [42], presentaron un análisis del rendimiento de implementaciones paralelas tanto del modelo de AC conocido como el "Juego de la Vida" (GoL, por sus siglas en inglés), como de un modelo de Lattice Boltzman, en varias arquitecturas. En este trabajo, realizaron un estudio utilizando contadores de hardware, con los cuales pueden verificar la eficiencia del algoritmo basado en los accesos a memoria local; además, hicieron un estudio sobre la escalabilidad de la implementación, en donde para el GoL con una malla de 64,000 × 64,000 y 1000 generaciones alcanzaron ~56x de aceleración con ~87% de eficiencia. Para ello, se optimizaron implementaciones paralelas del GoL [3] para ambientes HPC con múltiples CPUs, utilizando MPI (Memoria Distribuida) para simular espacios de dominio de grandes dimensiones. Particularmente, basándose en un estudio previo presentado por Millán et al. [46], desarrollaron varias versiones secuenciales del GoL, y posteriormente una versión MPI paralela. Aquí, las versiones de secuenciales y MPI fueron mejoradas significativamente al realizar varias optimizaciones, usando contadores de hardware para recopilar información sobre el rendimiento del código y detectar cuellos de botella.

Así, los autores desarrollan cuatro implementaciones secuenciales: *baseline, swap, one_grid* y *one_grid_ch*. a) La implementación "*baseline"* se basa en el trabajo previo de Millán et al. [3], la cual se puede considerar una implementación intuitiva no optimizada; cada iteración inicia borrando el arreglo que almacena el próximo estado en el tiempo, posteriormente se evalúa el estado siguiente de cada celda en base al estado actual de las celdas en la vecindad, para lo cual se determinan los índices de las celdas vecinas utilizando 4 sentencias *if-like*, las cuales a su vez, controlan las condiciones de frontera periódicas, finalmente se copia por completo el arreglo que almacena los estados del siguiente estado en el tiempo al arreglo del estado del tiempo actual. b)

La implementación "*swap*" se caracteriza por que, a diferencia de la implementación previa, realiza el intercambio de las referencias a los arreglos que contienen el estado siguiente y el estado actual del AC; por lo que no hay necesidad de realizar copias innecesarias o el borrado continuo del arreglo al iniciar cada iteración, las condiciones de frontera funcionan de la misma manera. c) La implementación denominada "*one_grid*", utiliza un solo arreglo para llevar a cabo la evolución del AC, lo que significa que los estados actual y siguiente se almacenan en el mismo. Esta implementación también agrega filas y columnas fantasma (halo), con las cuales las cuatro sentencias *if-like* para controlar las condiciones de frontera de la implementación "*baseline*", ya no son necesarias. La cuadrícula se incrementa en 3 en cada dirección (N + 3) × (M + 3). El resultado de la evolución de una célula se almacena en la célula izquierda diagonal. Las celdas de la cuadrícula se leen de arriba a abajo y de izquierda a derecha. Al leer los estados del vecindario de una celda, la célula diagonal ascendente izquierda sólo es necesaria para la celda actual. Cuando se calculan todas las celdas de la primera iteración, la siguiente iteración debe comenzar desde la cuadrícula desplazada almacenada en la iteración anterior, pero cambiando el punto de inicio y la dirección. La segunda iteración comienza desde la esquina inferior derecha y las celdas se leen de abajo hacia arriba y de derecha a izquierda. Los resultados de la segunda iteración se almacenan en la diagonal derecha-abajo de la celda. d) La última implementación secuencial "*one_grid_ch"*, es la misma que "*one_grid*", pero utiliza el tipo de dato char en lugar de int, con lo cual utiliza 4 veces menos memoria.

También presentan tres implementaciones paralelas: *baseline, one_grid* y *one_grid_ch*. El código utiliza la descomposición del dominio. Para el caso de la implementación paralela "*baseline",* cada proceso MPI recibe un bloque de la cuadrícula para ser procesado, luego comunica asincrónicamente las filas y columnas del borde de cada bloque a los procesos vecinos mientras procesa las celdas internas (las comunicaciones con los vecinos se traslapan con el procesamiento de las celdas internas del bloque). Cuando el proceso MPI termina con las celdas internas, espera en caso que las filas y columnas del borde no hayan sido completamente recibidas de procesos vecinos y luego procesa las celdas externas utilizando las celdas recibidas para calcular el estado siguiente. La implementación paralela "*one_grid*", utiliza el mismo concepto que su versión secuencial, con un único arreglo utilizado para almacenar tanto el estado actual como el estado siguiente del AC, pero a diferencia con la implementación paralela "*baseline*", este código no puede traslapar la comunicación con el procesamiento. La tercera implementación paralela "*one_grid_ch",* es similar a "*one_grid"*, pero utiliza el tipo de datos char en lugar del tipo de datos int.

Finalmente, aunque de manera muy básica, lo autores presentan un análisis de los costos de comunicación entre los procesos remotos al realizar un escalamiento; sin embargo, se limita a reducir el espacio de dominio hasta encontrar un porcentaje de comunicación pequeño, que no afecte el rendimiento.

3.2.2. El trabajo de Alves et. al.

Más tarde, Alves et. al, [40] sugieren una manera de mejorar la localidad y balancear los accesos a memoria en arquitecturas de memoria compartida con diseño NUMA, a través de técnicas que mapeen procesos a núcleos y datos a controladores de memoria, basados en la afinidad entre procesos y datos. Dichas técnicas de mapeo, pueden operar a diferentes niveles de hardware y software, lo cual afecta su complejidad, aplicabilidad, y ganancia en rendimiento resultante y de consumo energético. Para ello, los autores introducen una taxonomía para clasificar diferentes mecanismos de mapeo y brindar una visión global de las soluciones existentes. El objetivo de este trabajo es investigar el impacto en rendimiento al utilizar diversos mecanismos de mapeo basados en la afinidad entre procesos y datos, en aplicaciones con diferente comportamiento de acceso a la memoria

Para ilustrar la operación y beneficios de varios tipos de mecanismos de mapeo, los autores presentan el caso de estudio de una aplicación científica, Ondes3D, el cual, simula la propagación de ondas sísmicas generadas por los terremotos usando un método numérico de diferencias finitas y es implementado con OpenMP. Debido a que ésta tiene un comportamiento de acceso a memoria estático, por lo que es adecuado para un amplio rango de técnicas de mapeo, incluyendo aquellas que requieren trazas de acceso a memoria. Así, su patrón de comunicación lo determinan por la descomposición de dominio, llevando a grandes cantidades de comunicación entre procesos vecinos. En la versión base de Ondes3D, todos los datos de entrada son inicializados por el proceso maestro, llevando a un mapeo de datos desfavorable con una política *first-touch*, que causa un alto número de accesos NUMA remotos, así como un alto desbalance. Además, utilizan dos conjuntos de entrada (uno pequeño y uno grande para mostrar la influencia del tiempo de ejecución en las ganancias que pueden ser alcanzadas por cada tipo de mecanismo. De tal manera que ambos conjuntos usan los mismos datos de entrada, pero difieren en el número de iteraciones.

Los resultados de simulación, indican que el mapeo puede alcanzar grandes mejoras en rendimiento de más de 200% comparado con la versión base. Aunque los autores encuentran que los cambios al código fuente y el perfilado fuera de línea brindan las más grandes mejoras, también concluyen que estos mecanismos también son los que tienen la más grande sobrecarga inicial, al requerir cambios en el código fuente y la recompilación del código, o una generación de trazas de memoria que consume mucho tiempo (para esta aplicación, el trazado causo una ralentización de 120x comparada a la ejecución normal). Concluyendo así, que los mecanismos de mapeo tienen diferentes características, así como ventajas y desventajas, dependiendo de en donde y como sean implementados. Estas diferencias afectan su aplicación, ganancia y sobrecarga.

3.3 Análisis del estado actual.

A pesar que los trabajos actuales sobre implementaciones paralelas de modelos de AC son importantes, puesto que muestran resultados positivos al procesar grandes cantidades de datos en un menor tiempo, en comparación con sus contrapartes secuenciales, o debido a que permiten manejar grandes espacios de dominio que sería imposible manejar en un solo equipo; ninguno toma en consideración aspectos específicos de las arquitecturas de hardware en donde se ejecutan, y simplemente realizan una paralelización plana, en donde únicamente se limitan a dividir el trabajo entre las unidades de procesamiento de la plataforma utilizada.

Además, ninguno de los trabajos existentes especifica detalladamente la estrategia para realizar la aglomeración y el mapeo de los datos a los procesos; ni tampoco, para el caso de las implementaciones basadas en arquitecturas de memoria distribuida, cuáles son los costos de comunicación y sincronización entre los procesos remotos y como estos afectan al desempeño al aumentar la cantidad de unidades de procesamiento.

Es por ello que, en el siguiente capítulo se presenta una propuesta para el diseño y desarrollo de implementaciones paralelas de modelos de AC, que utilicen el mecanismo de mapeo adecuado, basado en su comportamiento de acceso a memoria, que permita mejorar el rendimiento en plataformas de HPC híbridas, a fin de permitir el manejo de sistemas a gran escala.

Capítulo 4.

Desarrollo de modelos de AC basados en arquitecturas paralelas con diseño NUMA.

En este capítulo, se presenta una propuesta para el diseño de implementaciones paralelas de modelos de AC. El objetivo general es realizar un análisis del desempeño en términos de escalabilidad y eficiencia, de implementaciones paralelas híbridas de modelos de autómatas celulares a gran escala, utilizando diversas configuraciones del espacio de dominio. Con la finalidad de determinar si al considerar la estructura topológica del sistema en donde se ejecutan, permite una estrategia que no solo brinde un mejor desempeño al minimizar los tiempos de ejecución, como la mayoría de los estudios existentes en la literatura, sino también de utilizar de manera adecuada los recursos existentes. Para lograr este objetivo, se hace uso de una metodología orientada al desarrollo de programas paralelos eficientes y altamente escalables. Con la finalidad de evaluar la propuesta, se implementa un modelo de AC conocido, que se usa como caso de estudio. Finalmente, se detallan las implementaciones paralelas para el caso de estudio, así como de las optimizaciones realizadas.

4.1 Diseño de implementaciones de modelos de AC.

Para realizar el diseño de implementaciones paralelas, es necesario contar con una guía que contemple las implicaciones básicas necesarias, independientemente de la arquitectura o modelo de programación que se esté utilizando. Para ello, existen muchas metodologías para el diseño de software paralelo propuestas en la literatura. En este trabajo de tesis se utiliza la metodología para el diseño de aplicaciones paralelas, propuesta por Ian Foster [38]. Esta metodología se divide en 4 etapas, las primeras dos etapas se enfocan en la concurrencia y la escalabilidad para descubrir algoritmos con estas características. En la tercera y cuarta etapa, se consideran aspectos específicos de las plataformas utilizadas, las cuales se pretende fortalezcan el desempeño final. Basado en esta metodología, a continuación, se presenta un diseño orientado al desarrollo de implementaciones paralelas de modelos de AC.

4.1.1 Particionamiento.

Debido al inherente paralelismo de grano fino de los AC, el espacio celular puede fácilmente particionado en entidades más pequeñas que lo conforman, a través de la descomposición de dominio. De tal forma que, cada celda del espacio de dominio puede evolucionar de manera simultánea y ser procesada por una tarea independiente.

4.1.2 Identificación de las comunicaciones.

Los elementos generados en la etapa de particionamiento pueden ser ejecutados concurrentemente, sin embargo, no son independientes. La Figura 4.1 muestra como para poder evolucionar, las celdas de los AC requieren obtener la información de sus vecinos.

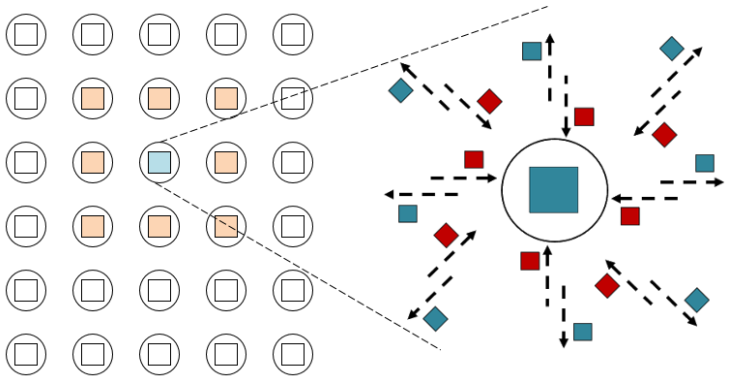


Figura 4.1. Vecindad de Moore de radio 1 para un espacio bidimensional mostrando los canales de comunicación necesarios para recopilar los datos de las celdas vecinas.

El patrón de comunicación de los AC depende directamente de la vecindad utilizada en el modelo, ya que la actualización de cada celda en el paso de tiempo actual (t), requiere conocer los estados de sus celdas vecinas y su mismo estado para utilizarlos como argumentos de la función de transición definida en el modelo. De esta forma, obtener el estado de la celda $c[i, j]$ en el siguiente paso de tiempo $(t+1)$. También es necesario que cada celda envíe su estado actual a las celdas en su vecindad, para que estas puedan realizar sus cálculos correspondientes.

Cabe recordar que, el espacio de dominio en memoria es finito; por lo que para actualizar las celdas al límite del espacio celular, se requiere definir una condición de frontera que se ajuste con las necesidades del modelo bajo estudio. De esta manera, todas las celdas del espacio celular tendrán la misma cantidad de vecinos y de canales de comunicación. Por ejemplo, para un espacio de dominio bidimensional con una vecindad de Moore de radio 1, cada celda tendrá 8 vecinos, por lo cual requiere de 8 canales de entrada, así como 8 canales de salida.

4.1.3 Aglomeración de acuerdo a los recursos disponibles

Por otra parte, la granularidad, $g(p_i)$, es también un factor importante. Ésta es una medida cualitativa que resulta de dividir el tiempo de procesamiento $t_{proc}(p_i)$ y el tiempo de comunicación $t_{com}(p_i)$ de un proceso p_i [57]:

$$g(p_i) = t_{proc}(p_i) / t_{com}(p_i)$$

El principal objetivo de esta fase es el de controlar la granularidad, ya sea para incrementar el procesamiento, o para decrementar los costos de comunicación; debido a que una descomposición de grano fino, como la que se realizó en la etapa de particionamiento, no necesariamente produce un algoritmo paralelo eficiente. La sobrecarga generada por los costos de comunicación, así como los costos por la creación de procesos pueden ser reducidos al incrementar la granularidad de la tarea, lo que a su vez permite mejorar la escalabilidad, incrementando el procesamiento.

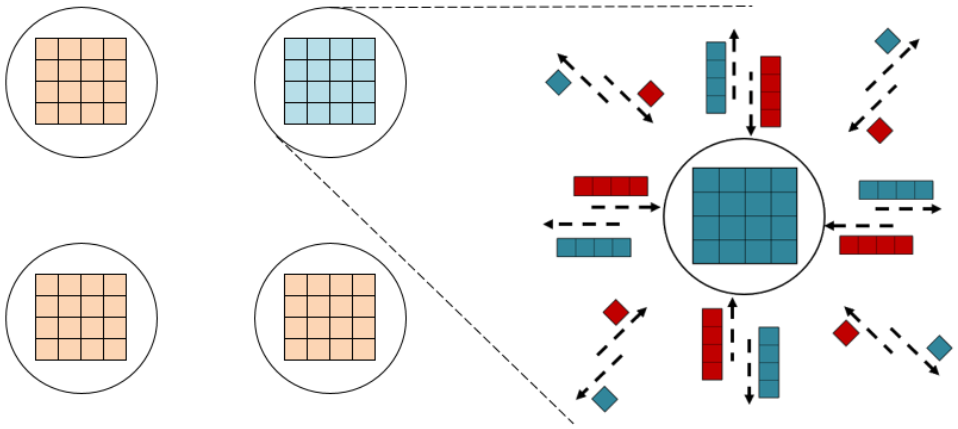


Figura 4.2. Aglomeración bidimensional de celdas contiguas.

Como se muestra en la Figura 4.2, la estrategia de aglomeración consiste en agrupar celdas contiguas del espacio de dominio del AC en un mismo bloque, de manera de que las cargas de trabajo se balanceen y se minimice la comunicación, preservando los canales de comunicación entre tareas dentro de la vecindad definida y eliminando comunicaciones innecesarias para aquellas celdas dentro de una misma tarea.

4.1.4 Mapeo sobre la arquitectura del sistema.

Con la finalidad de aprovechar y explotar adecuadamente las ventajas de las plataformas con diseño NUMA, es necesario incrementar la localidad, reduciendo los accesos remotos y la contención por el ancho de banda de los canales. Dos conceptos clave en el manejo del rendimiento en plataformas de memoria compartida con diseño NUMA son: la afinidad de procesos y la afinidad de memoria.

Afinidad de procesos.

Se refiere a asociar persistentemente un proceso con una unidad de procesamiento en particular, a pesar de la disponibilidad de otras instancias. El planificador del sistema operativo observará esta afinidad en sus decisiones de programación durante la vida útil del proceso.

La afinidad de procesos evita que los datos que estaba utilizando el proceso al momento de migrar, tengan que ser nuevamente cargados en las cache de la nueva unidad de procesamiento. Además, permite agrupar procesos que se comunican frecuentemente incrementando la localidad, reduciendo la contención por el ancho de banda entre los canales de comunicación de los sockets.

Afinidad de memoria.

La afinidad de memoria se refiere a la asociación de una reserva de memoria en un nodo NUMA específico, sin importar la ubicación del proceso que lo solicita.

Debido a que el mecanismo de acceso a memoria de las implementaciones de AC es estático, al utilizar en combinación la afinidad de procesos, la afinidad de memoria, y teniendo en cuenta la estructura topológica de la arquitectura de hardware en donde son ejecutados, es posible mejorar el rendimiento de las implementaciones paralelas de AC.

4.2. Caso de estudio.

En esta sección, se presenta la descripción de la implementación secuencial para el modelo bajo estudio basado en autómatas celulares: El Juego de la Vida (GoL).

4.2.1 El Juego de la Vida.

Descrito en el capítulo 2, el juego de la vida, GoL [3], es uno de los AC más conocido en la literatura, formulado por el matemático británico John H. Conway en 1970. El interés particular por el GoL es que logra un resultado parecido al comportamiento real de la vida, con el conjunto más simple de las reglas posible; por lo que, hasta la fecha, ha recibido un amplio interés por parte de investigadores. Además, una de sus características más importantes del GoL es su capacidad de realizar cómputo universal, es decir, que, con una configuración inicial apropiada, el GoL se puede convertir en una computadora de propósito general (máquina de Turing) [5].

El Pseudocódigo 4.1 (se presenta en inglés para un mejor entendimiento y simplicidad) muestra la versión secuencial standard del GoL utilizando 2 arreglos de tamaño N x N, cada uno para representar las celdas del paso de tiempo actual y siguiente (*t* y *t+1* respectivamente), y así, evitar errores causados por la sobreescritura de los estados de las celdas en el enmallado entre cada paso de tiempo.

```
1    for every generation in Life do
```

```
2       for every Cell in DomainSpace do
3           Neighbors ← 0
4           for every Cell in Neighborhood do
5               Neighbors ← Neighbors + Cell [i, j]
6           end for
7           if (Cell [i, j] is alive and neighbors == 3)
8               Cell[t+1] ← ALIVE //viva
9           else if (Cell is alive and (neighbors == 2 or neighbors == 3))
10              Cell[t+1] ← ALIVE
11          else
12              Cell[t+1] ← DEAD //muerta
13          end if
14      end for
15  end for
```

Pseudocódigo 4.1. "El juego de la vida".

A pesar de su simplicidad, el GoL es un modelo que genera patrones interesantes. Este modelo es importante, ya que provee los fundamentos básicos para crear simulaciones que muestran las características y comportamientos reproductivos de sistemas biológicos. Además, la implementación secuencial y paralela de este modelo brindan las bases para el desarrollo de modelos de AC más complejos.

4.3 Implementación paralela.

La paralelización de la implementación secuencial del modelo de AC descrito en la sección previa, se realiza utilizando una técnica tradicional de descomposición de dominio basada en el modelo Single Program Multiple Data (SPMD), en donde cada proceso ejecuta el mismo programa en su propio subconjunto de datos. A continuación, se describen varias consideraciones para las implementaciones realizadas.

4.3.1 Consideraciones generales para las implementaciones.

En lo siguiente, se describen las consideraciones que se tomaron en cuenta para el desarrollo de las implementaciones paralelas que se presentan en este trabajo.

Implementación estándar utilizada.

Existen en la literatura múltiples maneras de implementar los modelos de AC. En este trabajo se utiliza la versión convencional o estándar, que consiste de 2 arreglos para contener el estado actual y siguiente de las celdas, e intercambiándolas en cada generación para evaluar la evolución del modelo para múltiples generaciones.

Modelos de memoria utilizados.

Existen dos modelos de memoria, el modelo de memoria distribuida y el de memoria compartida. En el caso del primero, al incrementar el número de nodos de procesamiento, es posible aumentar la escalabilidad. El segundo modelo, pretende explotar la eficiencia intranodo, ahorrando memoria, incrementando la localidad de los datos. La programación paralela híbrida permite aprovechar las ventajas de cada modelo de memoria, es por ello que en este trabajo se desarrollaron las implementaciones utilizando ambos modelos. Esto permite incluso reducir la parte no paralelizable

Descomposición de dominio.

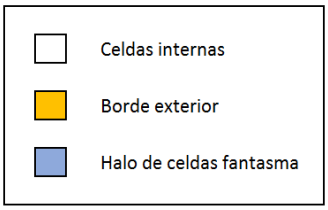
Para llevar a cabo la implementación paralela de los modelos de AC mencionados anteriormente, se utiliza la técnica de descomposición de dominio en el espacio celular del AC, basada en la metodología de diseño previamente descrita en la sección 4.1. Así, el espacio de dominio se representa por un arreglo bidimensional, el cual se divide de manera balanceada entre el número de entidades, de acuerdo a la cartografía especificada. Cada división representa un subdominio, que será procesado por una entidad independiente. La descomposición de dominio puede ser realizada de manera global o de manera local. De tal manera que, al utilizar un modelo de programación híbrido es posible una descomposición de dominio jerárquica, en donde el primer nivel realiza la descomposición del espacio de dominio global utilizando el modelo de memoria distribuida; mientras que los niveles siguientes utilizan el modelo de memoria compartida.

Inicialización del sistema.

Para realizar la descomposición de dominio en el primer nivel, se identifica cada proceso cartográficamente, así como a los procesos en su vecindad. Esto es posible a través de las funciones de la librería MPI, al obtener los identificadores y coordenadas de los procesos involucrados.

Halo de celdas fantasma. / Estructura del espacio de dominio.

En lo referente a la estructura del espacio de dominio, para cada nodo de procesamiento se utiliza un halo de celdas fantasma adicional al espacio de su subdominio, que consiste en un conjunto de celdas que lo rodean, en donde es posible almacenar los estados de las celdas en los bordes exteriores de los subdominios de los nodos en la vecindad; como se muestra en la Figura 4.3.



Espacio de dominio global de 24 x 24 dividido en 9 nodos de procesamiento organizados en una cartografía de 3 x 3, generando subdominios de 8 x 8. Cada subdominio utiliza un halo de celdas fantasma para almacenar las celdas en el borde exterior de los nodos en la vecindad, y así poder procesar las celdas en el borde exterior de cada subdominio.

Figura 4.3. Representación de un espacio de dominio global dividido entre 9 nodos de procesamiento organizados en una cartografía de 3 x 3.

Es importante mencionar, que es necesario que este halo de celdas fantasmas sea actualizado en cada iteración del modelo con los nuevos valores para que las celdas en el borde exterior del espacio de dominio local puedan ser procesadas.

Comunicación.

En lo que refiere al proceso de actualización del halo de celdas fantasma del subdominio (i, j), éste consiste en enviar las celdas en una dirección del borde exterior a los subdominios de la vecindad en la misma dirección. A su vez, el subdominio (i, j) envía las celdas en su borde exterior hacia los subdominios en su vecindad para que realicen el mismo procedimiento. De tal manera que se reduzca la sobrecarga de comunicación

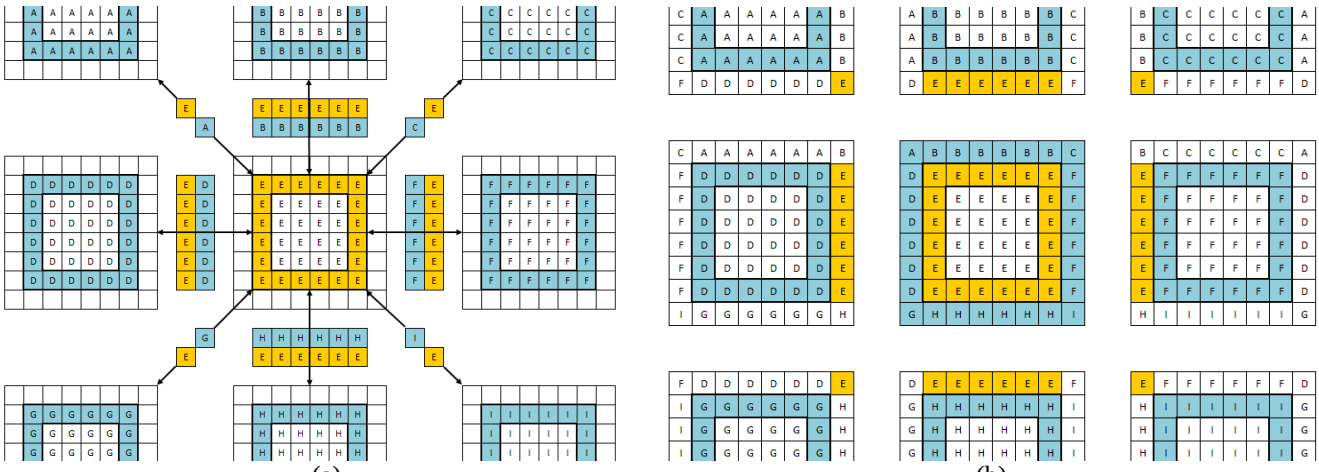


Figura 4.4. Transferencia de borde exterior de subdominio [i, j], hacia halo de celdas fantasma de subdominios vecinos, y viceversa.

La Figura 4.4 muestra el proceso de comunicación que se utiliza, como puede observarse, cada grupo de celdas enviadas hacia una dirección comparten el mismo canal de comunicación, a fin de reducir sobrecarga de comunicación. Adicionalmente, para evitar tiempos de espera causados por la sincronización de la transferencia, el procesamiento y la comunicación se realizan de manera simultánea, a través del uso de las funciones asíncronas MPI_Isend y MPI_Irecv, las cuales permiten ocultar la latencia al procesar las celdas internas de cada subdominio mientras se transfieren en segundo plano las celdas fantasmas. Idealmente, cada proceso tendrá suficiente trabajo para ocultar por completo el proceso de comunicación. En el pseudocódigo 4.2, se muestra un ejemplo de cómo se realiza el traslape de la comunicación y el procesamiento utilizando un halo de celdas fantasma.

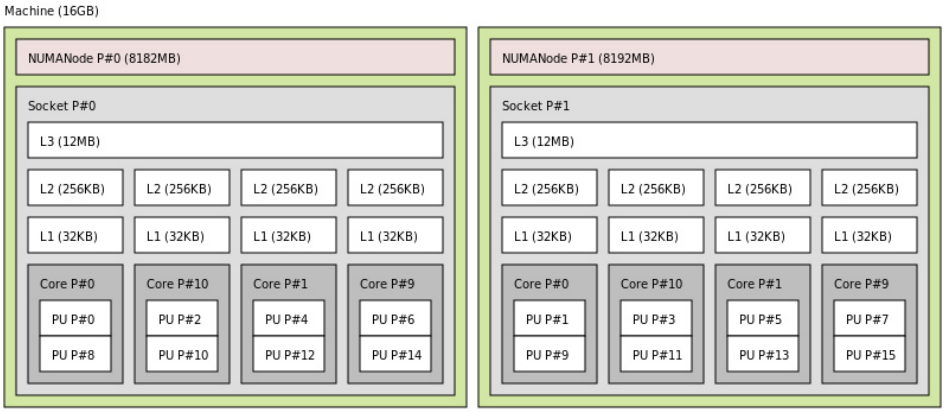
```
1  exchange_borders ()
2  for each iteration do
3      compute_internal cells ()
4      wait_for_borders ()
5      compute_external_cells ()
6      swap_lattice ()
7      exchange_borders ()
8  end for
```

Pseudocódigo 4.2. Traslapando la comunicación y el procesamiento utilizando un halo de celdas fantasma.

Idealmente, siempre habrá suficiente procesamiento de las celdas en el interior para ocultar la latencia causada por la comunicación. En caso contrario, la función *wait_for_borders* () bloqueará el proceso hasta que se terminen de recibir las celdas y pueda procesarse el borde exterior de manera consistente.

Obtención de la topología del sistema.

Para obtener la topología del sistema, se utiliza la librería libnuma, a través de la cual es posible no sólo obtener información sobre la estructura topológica del sistema e identificar la cantidad de nodos NUMA que conforman el sistema; tal como la cantidad y los identificadores de las unidades de procesamiento y su ubicación en el sistema, así como la distancia entre los nodos NUMA (ver Figura 4.5(b)), etc.



(a)

```
node distances:
node   0   1
  0:  10  20
  1:  20  10
```

(b)

Figura 4.5. Estructura topológica de un sistema NUMA con 2 nodos NUM, 8 cores con SMT (a) y su tabla de distancias entre nodos (b)

Este proceso es muy importante, ya que como se presenta en la Figura 4.5, las unidades de procesamiento utilizan identificadores con números no necesariamente asignados de manera continua, además de que cada plataforma puede manejar una numeración diferente.

Identificación de los procesos e hilos.

Otro punto importante de la implementación es la identificación de los procesos e hilos a usar. Para las implementaciones de este trabajo de tesis, se considera que cada proceso o hilo cuenta con un identificador único, con los cuales es posible ubicar su posición cartográfica en la topología que se encuentran. Para el caso de los procesos MPI se utiliza la funcion MPI_Comm_rank. Para los hilos en OpenMP, la función omp_get_thread_num (), con el que es posible administrarlo.

Afinidad de procesos.

Cuando se aplica la afinidad de procesos, los hilos son colocados en las unidades de procesamiento de acuerdo a la topología del sistema, a través del uso de la función set_schedaffinity.

Asignación de memoria.

Para reservar la memoria se utilizan diferentes métodos dependiendo de la implementación. Uno es a través de la función malloc (), la cual reserva memoria de acuerdo a las políticas del sistema operativo. Otro es a través de la función numa_alloc_onnode (), presente en la librería libnuma, con la que es posible reservar memoria en un nodo NUMA de manera explícita. Normalmente, esta última función continúa reservando memoria en otro nodo NUMA

cuando no existe memoria suficiente en el nodo NUMA especificado; por lo que se hace uso de la función numa_set_strict () para no continuar, en tal caso.

Establecimiento de la configuración inicial.

Para el caso del modelo de AC del GoL, la configuración inicial se puede establecer de manera estática, aplicando una configuración predefinida, con el objetivo de verificar el correcto comportamiento del modelo; o establecerse de manera aleatoria, a través del uso de funciones generadoras de números pseudoaleatorios "thread-safe". De acuerdo al estudio realizado por Gibson et al. [43], la mayor actividad de celdas para el modelo del GoL se da al utilizar una probabilidad de existencia de celdas vivas entre 20% y 60%, por lo que en este trabajo se utiliza una probabilidad de 50% de celdas vivas, para alcanzar un alto nivel de actividad después de al menos 1000 generaciones.

Puntos de sincronización.

En el caso particular del modelo de memoria distribuida, algunas funciones de comunicación sirven implícitamente como puntos de sincronización debido a su naturaleza bloqueante. Sin embargo, en este trabajo se han reemplazado por funciones asíncronas no bloqueantes, con las que se incrementa el rendimiento al reducir los tiempos de espera, evitando este tipo de puntos de sincronización. El único punto de sincronización implícito es a través de la función MPI_Waitall (), la cual únicamente se cumple sí el procesamiento se realiza en un tiempo menor que el proceso de comunicación. Para establecer un punto de sincronización de manera explícita en el modelo de memoria distribuida, se hace uso de barreras a través de la función MPI_Barrier ().

En el caso del modelo de memoria compartida, estos puntos de sincronización se llevan a cabo a a través del uso de barreras.

Intercambio de mallas.

Una vez que se procesan todas las celdas de un subdominio, se habrá obtenido su nuevo estado para el tiempo $t + 1$ y finalizado una generación del modelo de AC. En ese momento, entonces se intercambian las referencias entre la malla actual y siguiente para continuar la evolución del modelo de AC, como se muestra en la Figura 4.6.

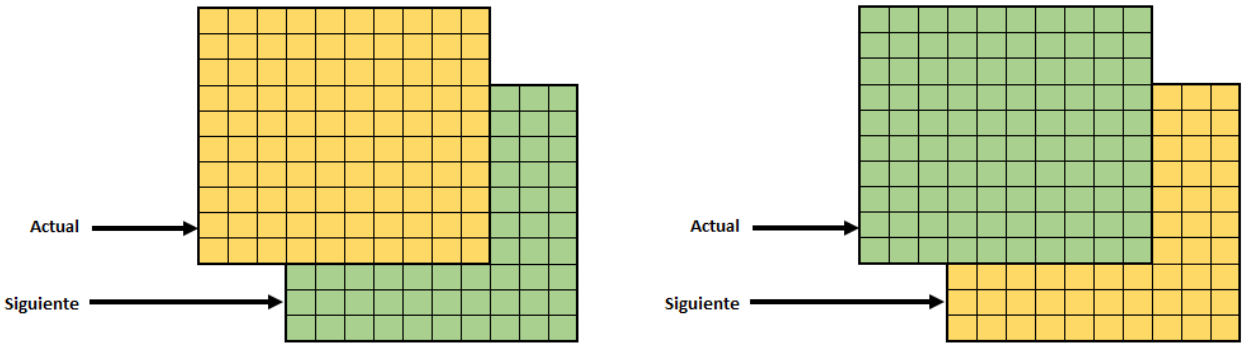


Figura 4.6 Intercambio de referencias entre mallas que representan el estado actual (t) y siguiente (t+1) del espacio de dominio.

4.3.2 Consideraciones específicas para la implementación explícita.

Descomposición de dominio.

Este tipo de implementación llamada *explícita*, se deferencia con respecto a las otras implementaciones en la descomposición del dominio. Para ello, se realiza una descomposición de dominio entre los nodos NUMA especificados para cada nodo de procesamiento. En la Figura 4.7, se muestra el ejemplo de un subdominio de tamaño 14 x 24 dividido en 8 nodos NUMA, organizados en una cartografía de 2 x 4. Como puede notarse de esta figura, cada bloque cuenta con una fracción del borde exterior y del halo de celdas fantasma del subdominio. En sistemas más grandes pueden existir cartografías que incluyan bloques de memoria con solo celdas internas.

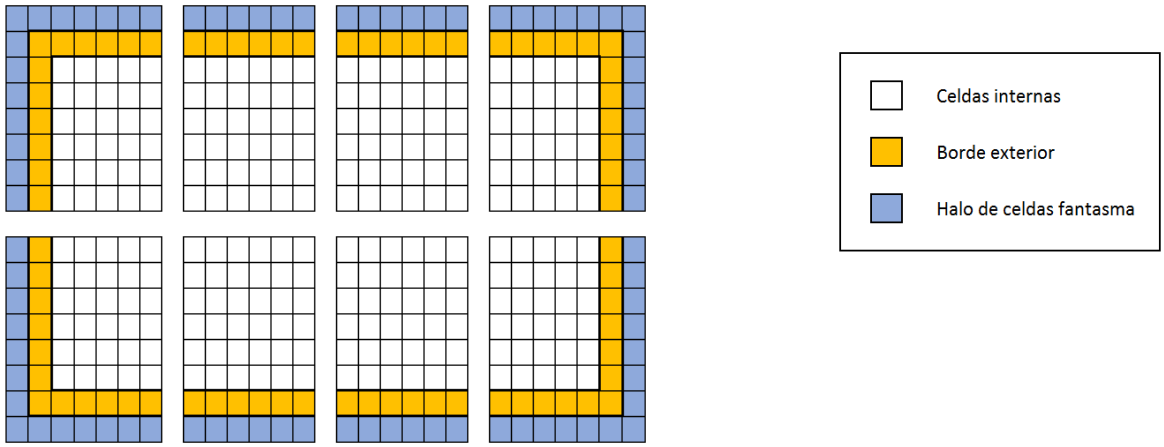


Figura 4.7. Descomposición del subdominio al tomar en cuenta la topología de un sistema NUMA.

Identificación de procesos e hilos.

Para la implementación explícita, los hilos son agrupados en los nodos NUMA con base en la topología del sistema. Así, para administrar y sincronizar los hilos tanto del mismo como de diferente grupo, es necesario generar identificadores adicionales. Por simplicidad, a cada nodo NUMA utilizado se le asigna la misma cantidad de hilos. Además de su identificador global, cada hilo cuenta con un identificador dentro de su grupo, el cual simplemente corresponde al número de hilos por nodo NUMA (id % numa_threads). En cada grupo, el hilo con el identificador más pequeño es el hilo coordinador del grupo, el cual se encarga de la sincronización de variables que son compartidas en el grupo, pero "privadas" entre los otros grupos.

Estructura de la malla.

Por otra parte, se considera que cada nodo NUMA debe conocer su estructura local, y su posición global dentro de la cartografía a la que pertenece. Para ello, se cuenta con una estructura de datos en donde se guarda dicha información, la cual es administrada por el hilo coordinador de cada grupo y es legible para todos los hilos del grupo. Además, en esta estructura se encuentra el bloque de memoria reservado por el nodo NUMA para representar la porción del subdominio al que representa. Esta información es inicializada en un arreglo de estructuras compartido y posteriormente, se pasa dicha información a una estructura local para cada hilo, con el objetivo de evitar accesos simultáneos y remotos a los datos.

Identificación de bordes.

Para la implementación explícita, se identifica cuales celdas son del halo de celdas fantasma, cuales pertenecen al borde exterior y cuales son celdas al interior (que no en todos los casos son las mismas), a través de las coordenadas cartográficas de cada nodo NUMA. Este proceso se requiere también al momento de reservar y liberar la memoria, ya que con el uso de la función numa_free (), es necesario especificar la cantidad de memoria a liberar.

Procesamiento de bordes

Debido a que cada nodo NUMA cuenta únicamente con una pequeña porción del borde exterior del subdominio, la responsabilidad de procesamiento le es asignada al hilo coordinador del nodo NUMA.

Balanceo de carga en los hilos del grupo.

Con la finalidad de maximizar la utilización de los recursos, la cantidad de trabajo realizada por cada hilo debe ser aproximadamente igual, de manera que no haya un hilo sobrecargado. Así, la cantidad de trabajo se asigna de acuerdo al número de celdas procesadas por cada hilo.

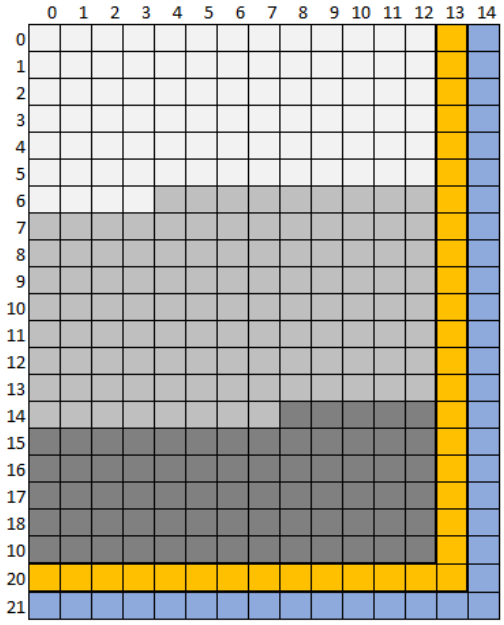


Figura 4.8. Balanceo de carga entre los hilos asignados a un nodo NUMA.

En la Figura 4.8, se muestra un ejemplo de cómo se realiza el balanceo de carga para las celdas al interior del bloque de memoria asignado a un nodo NUMA, entre sus 3 hilos, con base en el número de celdas a procesar.

Estableciendo los límites de procesamiento.

Para establecer la configuración inicial del modelo bajo estudio, tanto las celdas que pertenecen al borde exterior de la malla, como las celdas internas se procesan. Particularmente, para el procesamiento en sí del espacio celular del AC, primero se procesan las celdas internas y después de haber recibido las celdas de los bordes exteriores de los nodos vecinos se procesa el borde exterior. Por esta razón, antes de establecer la configuración inicial, se realiza un balanceo de carga entre los hilos del nodo NUMA para procesar el borde exterior y las celdas internas. Además, antes del procesamiento del modelo de AC se realiza nuevamente el balanceo de carga entre los hilos considerando únicamente las celdas internas.

Descripción de la implementación explicita.

En el Pseudocódigo 4.3, se muestra la manera en la que la implementación explícita se lleva a cabo, tomando en cuenta las consideraciones presentadas en este capítulo.

```
1  get_numa_topology ()
2  init_environment ()
3  local_domain_decomposition ()
4  allocate_shared_numa_array ()
```

```
5   parallel
6       get_thread_ids ()
7       thread_binding ()
8       if (is_numa_manager_thread ())
9           get_numa_info ()
10          detect_borders ()
11          allocate_curr_lattice ()
12          allocate_next_lattice ()
13          allocate_thread_boundaries ()
14      end if
15      allocate_numa_array ()
16      allocate_numa_map ()
17      copy_shared_numa_array_to_local ()
18      set_processing_boundaries ()
19      thread_load_balancing ()
20      set_initial_configuration ()
21      set_processing_boundaries ()
22      thread_load_balancing ()
23      if (is_numa_manager_thread ())
24          free_curr_lattice ()
25          free_next_lattice ()
26          free_thread_boundaries ()
27      end if
28  end parallel
29  free_shared_numa_array ()
30  clean_environment ()
```

Pseudocódigo 4.3. Descripción general de implementación explicita

En el siguiente capítulo, se presentan los resultados obtenidos a partir de los experimentos realizados en arquitecturas específicas, a fin de analizar la hipótesis planteada en este trabajo de tesis.

Capítulo 5.

Resultados obtenidos.

En este capítulo, se presenta una evaluación de la propuesta para el diseño de implementaciones paralelas del modelo de AC presentada en el capítulo previo. Pare este propósito, se generan y analizan los resultados que se obtienen de varios experimentos de las implementaciones sobre el modelo de AC bajo estudio, en varias plataformas. Con base en los resultados que se obtienen, se corrobora el cumplimiento de la hipótesis planteada en la introducción de este trabajo de tesis.

5.1. Características de la plataforma de ejecución.

Todas las simulaciones que se presentan en este capítulo, se realizaron utilizando el cluster Tonatiuh del Instituto de Ingeniería de la UNAM (IINGEN). Tonatiuh se compone de 5 nodos (1 nodo de administración y 4 nodos de procesamiento), interconectados a través de un enlace Gigabit Ethernet a 1 Gbps. De tal manera que, los nodos de procesamiento suman en conjunto un total de 200 núcleos, 448 GB de RAM, y 6784 núcleos GPU. Tonatiuh utiliza el sistema operativo GNU/Linux, con la versión 2.6.32 del kernel, a través de la distribución Rocks 6.2 basado en CentOS 6.9. En la Tabla 5.1 se muestran las características específicas de cada nodo de Tonatiuh.

Nodo admin: tonatiuh	2 Intel Xeon CPU E5-2670 v3 @ 2.30GHz, 12 cores per socket, 1 thread per core; 128 GB RAM; 2 NUMA domains.
Nodo 0: compute-0-0	4 AMD Opteron Processor 6380 @ 2.50Ghz, 8 cores per socket, 2 threads per core; 128 GB RAM; 8 NUMA domains.
Nodo 1: compute-0-1	4 AMD Opteron Processor 6276 @ 2.30Ghz, 8 cores per socket, 2 threads per core; 128 GB RAM; 8 NUMA domains.
Nodo 2: compute-0-3	2 Intel Xeon CPU E5-2680 v2 @ 2.80GHz, 10 cores per socket, 2 threads per core; 64 GB RAM; 2 NUMA domains; 2 NVIDIA Tesla K40m, 2880 GPU cores, 12 GB RAM.
Nodo 3: compute-0-4	2 AMD Opteron Processor 6272 @ 1.4 Ghz, 8 cores per socket, 2 threads per core; 128 GB RAM; 4 NUMA domains; 2 NVIDIA Tesla M2090, 512 GPU cores, 6 GB RAM.

Tabla 5.1. Resumen de características de los nodos que conforman el cluster tonatiuh del IINGEN.

Para las implementaciones de los modelos bajo estudio, se utilizaron las herramientas siguientes: el compilador de gcc 4.4.7, la API OpenMP 3.0, la interfaz de paso de mensajes MPI a través de la implementación OpenMPI 1.6.2, y la librería libnuma 2.0.9.

Además, sólo se usan los nodos “compute-0-0” y “compute-0-1” del clúster. Esta decisión se tomó debido a que la heterogeneidad de clúster complicaba realizar una evaluación del desempeño de las implementaciones de manera adecuada. Cabe mencionar que los dos nodos de procesamiento seleccionados cuentan con la mayor cantidad de nodos NUMA. En la Figura 5.1 se muestra la organización y distribución de las unidades de procesamiento y de los nodos NUMA para estos equipos.

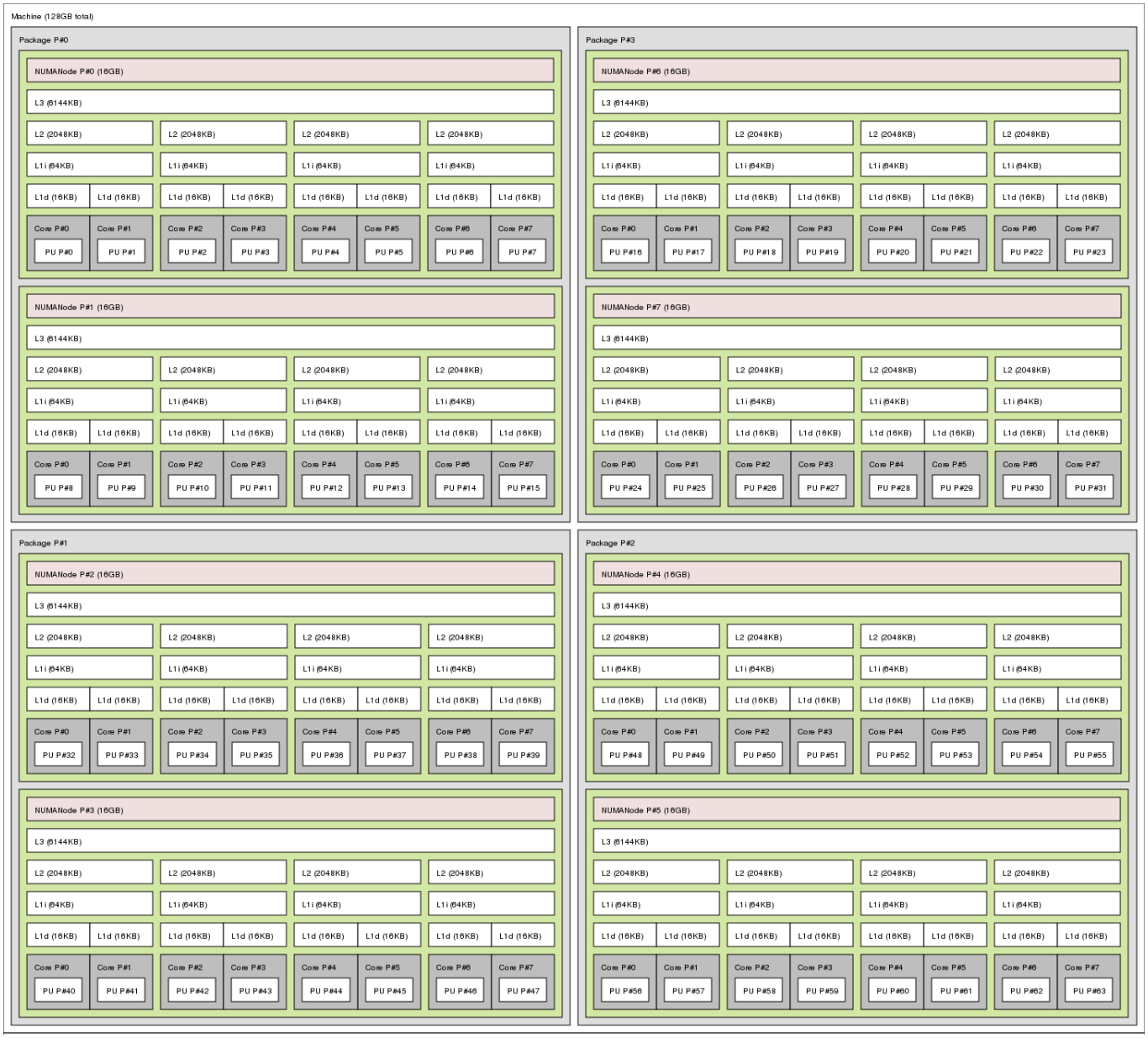


Figura 5.1. Topología de los nodos de procesamiento compute-0-0 y compute-0-1 del cluster Tonatiuh

5.2. Metodología de la evaluación.

Para llevar a cabo la evaluación de las optimizaciones propuestas, se realizaron múltiples implementaciones de los modelos utilizados como caso de estudio. En la Tabla 5.2, se describen cada una de las implementaciones realizadas para cada caso bajo estudio y se detallan sus características principales.

5.2.1 Implementaciones de prueba.

Nombre de la implementación	Descripción
Secuencial	Realiza el procesamiento de todo el espacio de dominio utilizando únicamente el proceso inicial. Esta implementación sirve como base para comprobar que los resultados de las implementaciones paralelas sean correctos.
Base	Implementación paralela híbrida de la versión "secuencial". Hace uso tanto de memoria distribuida como de memoria compartida. El espacio de dominio global se divide entre los nodos de procesamiento especificados de acuerdo a la cartografía indicada. De manera convencional, cada nodo de procesamiento reserva memoria para los arreglos necesarios del tamaño del subdominio que le corresponde. El sistema operativo, a través de su planificador, calendariza los hilos asignados en las unidades de procesamiento según sus políticas de planificación.
Implícita	La diferencia con la implementación "base" es que posterior a la reserva de memoria, se enlazan los procesos a unidades de procesamiento específicas, de acuerdo a la topología del sistema. De esta manera, se evita la migración de los procesos. Posteriormente, se inicializan los arreglos por primera vez, lo que obliga al sistema operativo a hacer uso de la política de asignación de memoria "first-touch" de manera implícita; como consecuencia, las páginas de memoria serán reservadas físicamente en el banco de memoria del nodo NUMA en el que se encuentran los procesos que accedan al primer elemento de cada página previamente reservada.
Explicita	En este caso, además de enlazar los procesos como en la implementación implícita, el subdominio de cada nodo de procesamiento es dividido físicamente, y haciendo uso de la librería "libnuma" cada uno de estos bloques de memoria es reservado de manera explícita, colocándolo en el nodo NUMA en donde se encuentran los hilos que procesarán esos datos, a fin de incrementar la localidad. El tamaño especificado para reservar memoria a través de la función "numa_alloc_onnode ()", será redondeado al múltiplo superior del tamaño de la página de memoria en cada llamada.

Tabla 5.2. Descripción de las implementaciones utilizadas para la evaluación.

Las implementaciones mencionadas en la Tabla 5.2 se desarrollaron utilizando el lenguaje de programación C y sus códigos fueron compilados con la bandera de optimización -O2. En el caso de las implementaciones paralelas, todas fueron desarrolladas utilizando la API OpenMP para el caso de memoria compartida, así como la librería de paso de mensajes MPI para memoria distribuida. Además, las implementaciones "implícita" y "explicita" hacen uso de la librería libnuma para recopilar información sobre la estructura topológica del sistema, o para reservar memoria específicamente en los nodos NUMA necesarios, según sea el caso**.**

5.2.2 Descripción de los parámetros de entrada.

Para llevar a cabo la evaluación, se utilizaron los tamaños de malla mostrados en la Tabla 5.3 para el espacio de dominio global de la simulación. Estos tamaños se definieron considerando el trabajo se enfoca a sistemas de escala grande.

Dimensiones	Total de elementos	Espacio (por arreglo)
2048 x 2048	4,194,304	16 MB
4096 x 4096	16,777,216	64 MB
8192 x 8192	67,108,864	256 MB
16384 x 16384	268,435,456	1 GB

Tabla 5.3. Tamaños de espacios de dominio utilizados para cada experimento.

Para cada una de las implementaciones, se realizaron múltiples experimentos, con el mismo número de generaciones en todos los casos, 1,000.

El número de hilos evaluado para cada implementación varía en función de las características de las mismas. Particularmente, para el caso de la implementación "explícita", los hilos fueron agrupados de acuerdo al número de nodos NUMA especificados, permitiendo evaluar para NP = 1, 2, 4, 6, 8, 12, 16, 24, 32, 36, 48 y 64. Mientras que en el caso de las implementaciones base e implícita, se evaluó con NP = 1, 2, 4, 6, 8, …, de 2 en 2 hasta 64.

Por otra parte, la cartografía global depende del número de nodos de procesamiento, mientras que la cartografía local depende de la cantidad de nodos NUMA. Ambas cartografías se especifican al momento de la ejecución, y se organizan de manera que ambos ejes de la cartografía tengan la menor diferencia posible de tamaño entre ellos.

Cabe mencionar que para el caso de la implementación "explícita", a pesar de que cada nodo NUMA tiene una capacidad de 16 GB de memoria RAM, no fue posible realizar pruebas con arreglos de dimensiones más grandes; ya que cada llamada para reservar memoria en un nodo NUMA especifico, a través de la función "numa_alloc_onnode ()" crea un nuevo mapa de memoria. Sin embargo, cada proceso tiene permitido un número máximo de mapeos de memoria definido en /proc/sys/vm/max_map_count, el cual solamente puede ser modificado por el administrador del sistema.

5.2.3 Prueba de verificación.

Para asegurar el correcto funcionamiento de las implementaciones paralelas, antes del proceso de recopilación de datos, se comprobó que la salida generada después de un gran número de generaciones fuera la misma que en las versiones secuenciales. Para ello y por simplicidad, se capturaron en archivos de texto las salidas finales de cada una de las implementaciones, y se comprobaron los resultados a través del comando "diff" como se muestra a continuación:

```
diff <salida_secuencial.dat> <salida_paralela.dat>
Files salida_secuencial.dat and salida_paralela.dat are identical
```

Figura 5.2. Comando utilizado para verificar los resultados entre las salidas de las implementaciones secuenciales y paralelas de los modelos de AC.

5.3 Evaluación experimental.

El interés de este trabajo es analizar el rendimiento de los accesos a memoria durante el procesamiento del modelo del AC, por lo cual los tiempos reportados únicamente corresponden a dicho evento. Otros tiempos como el de la inicialización del arreglo, tiempos administrativos, etc., no son tomados en consideración.

Cada resultado de las simulaciones mostrado en las gráficas o tablas se obtuvo del promedio de 10 ejecuciones. La desviación estándar es muy pequeña (generalmente menor al 1 %), por lo que las barras de error no fueron incluidas en las figuras.

La evaluación experimental se realizó a través de un análisis de escalabilidad (fuerte y débil) para observar el impacto de las políticas mencionadas anteriormente, en el rendimiento de las implementaciones paralelas de AC cuando el número de hilos se incrementa.

5.3.1 Escalamiento fuerte.

En el escalamiento fuerte, el tamaño del problema es fijo mientras el número de procesos o hilos se incrementa, por lo que analizar la proporción de la reducción del tiempo con base en el número de procesos o hilos agregados permite determinar el rendimiento obtenido.

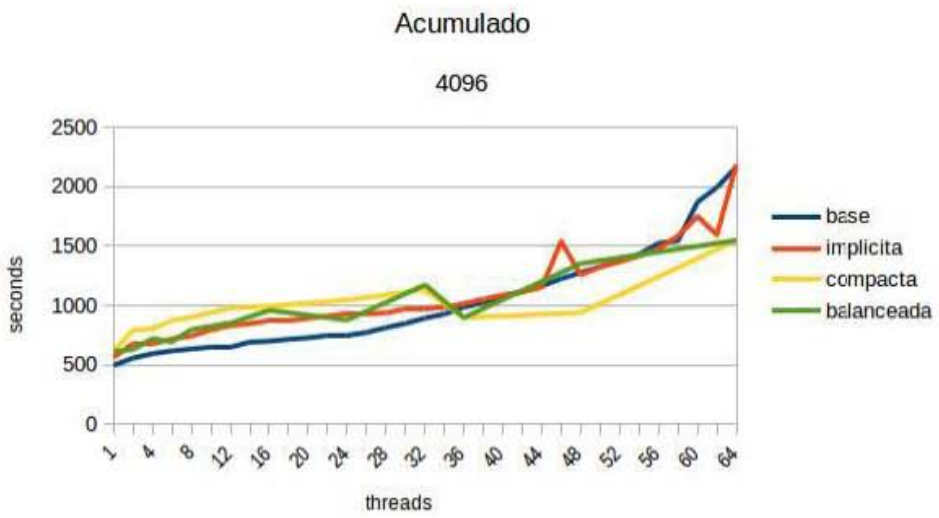
En este caso, se realizó un escalamiento fuerte para N = 2048, 4096, 8192 y 16384, con las 3 implementaciones paralelas (base, implícita y explícita), en donde la implementación "explícita" fue organizada de 2 maneras diferentes: "compacta", en la que los hilos son colocados lo más cercanos posible y "balanceada", en la que los hilos se distribuyen entre los nodos NUMA. En lo siguiente se presentan los resultados obtenidos.

Tiempo acumulado.

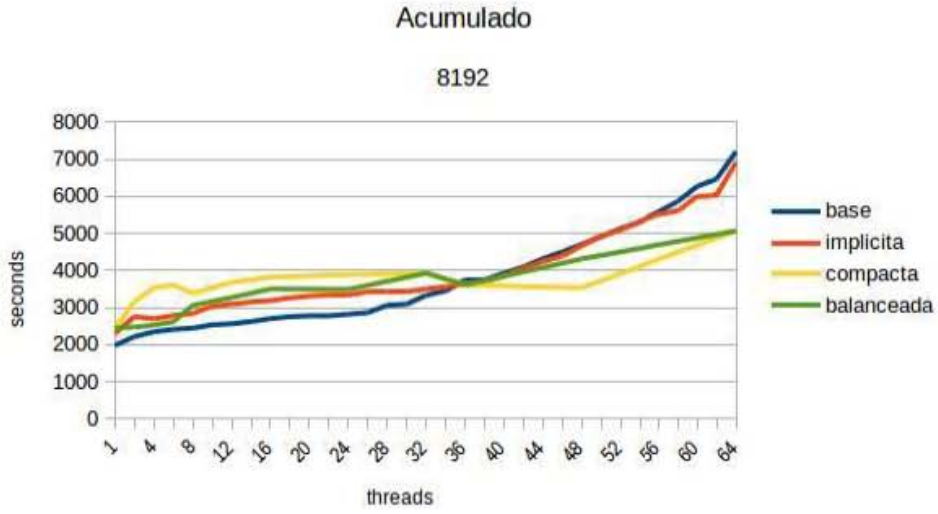
Primeramente, se realizó una evaluación del tiempo acumulado con respecto al número de hilos utilizados para diferentes espacios de dominio, cuyos resultados se muestran en la Figura 5.3. Como puede observarse de esta figura, para todos los espacios de dominio considerados, al inicio los tiempos de ejecución de las implementaciones "implícita" y "explicita" son superiores al tiempo de la implementación "base". Sin embargo, al incrementar el número de hilos, este comportamiento cambia aproximadamente a partir de 36 hilos, especialmente para el caso de la implementación "explicita-compacta". Esto es más notorio al incrementar el tamaño del espacio de dominio. El incremento del tiempo con pocos hilos, se debe a la sobrecarga de operaciones paralela derivada de la comunicación y sincronización intrínseca que requieren los AC. Estos resultados indican que se logra un uso más adecuado de los recursos de cómputo al incrementar la localidad de los datos con respecto a los procesos, como se esperaba.

(a)

(b)



(c)



(d)

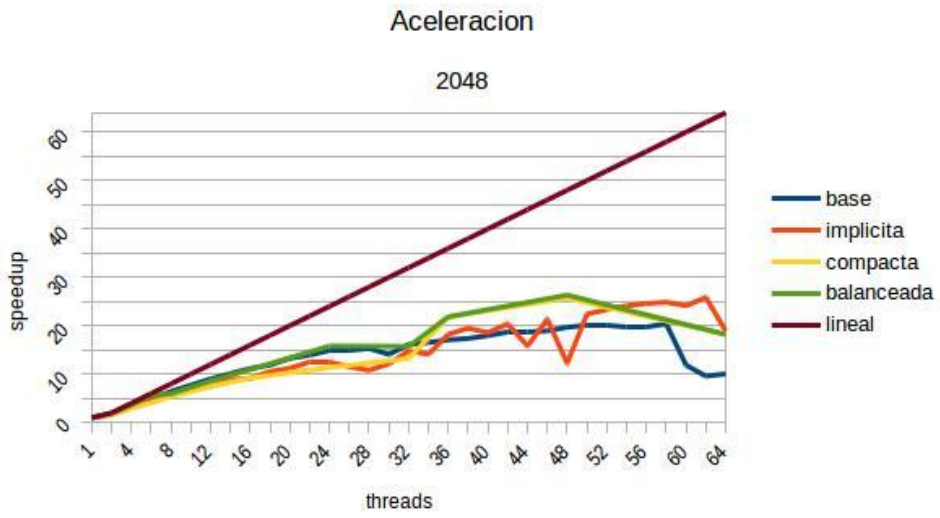
Figura 5.3 Tiempos acumulados utilizando un espacio de dominio de 2048 (a), 4096 (b), 8192 (c) y

16384 (d).

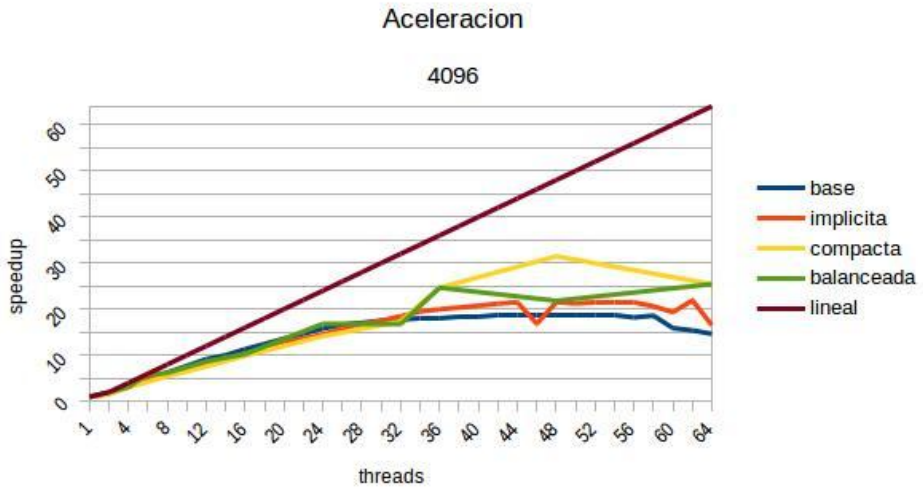
Aceleración.

Segundo se estudió la aceleración de la implementación. En la Tabla 5.4 se muestran las aceleraciones de las implementaciones “base” y “explicita-compacta” para NP = 1, 2, 4, 8, 16, 32, 48 y 64, y T = 1000, y sus correspondientes gráficas con respecto al número de hilos se muestran en la Figura 5.4. Como se observa de estas figuras, la implementación explicita alcanza una aceleración de ~34x para NP = 48 y N = 16384, contra un ~19.5x de la implementación base considerando los mismos parámetros. Nuevamente, se logra un uso más adecuado de los recursos de cómputo porque al incrementar la localidad de los datos con respecto a los procesos, se reduce o evitan, en la medida de lo posible, los accesos remotos a memoria.

(a)



(b)



(c)

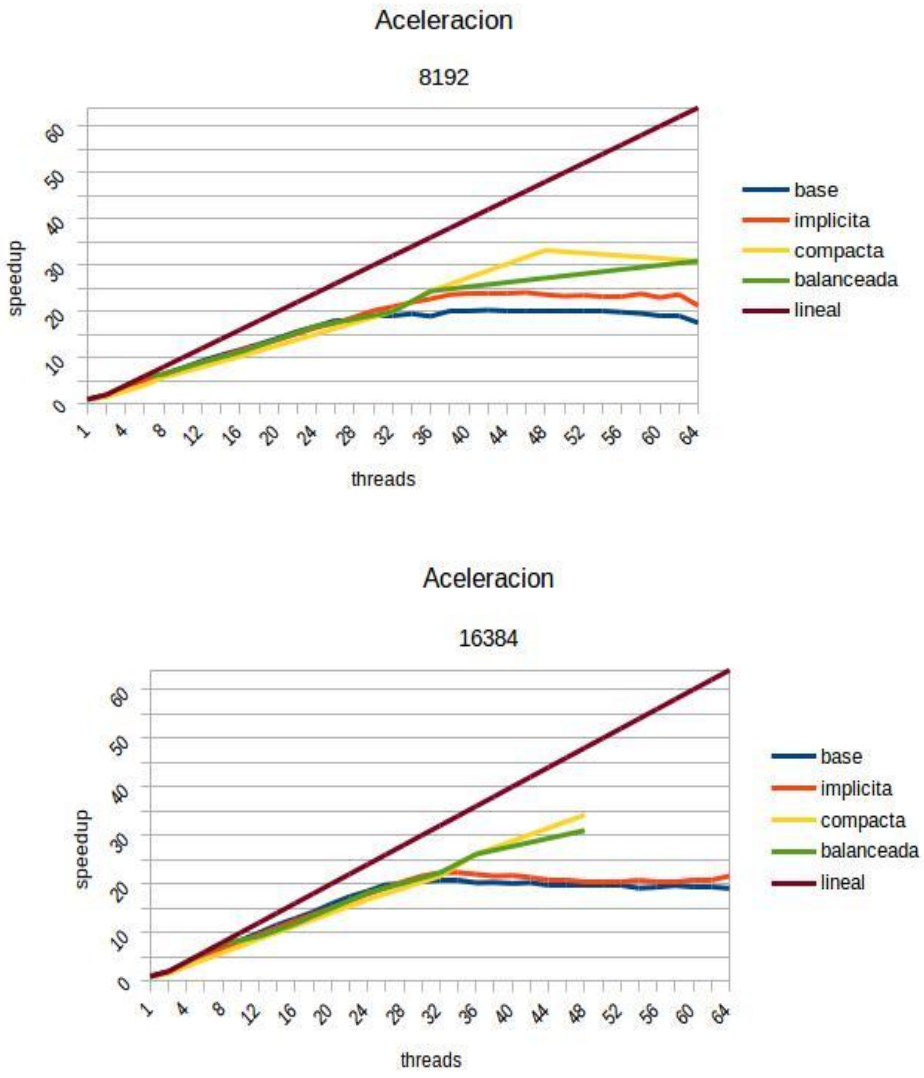


Figura 5.4. Tabla de aceleración relativa utilizando un espacio de dominio de 2048 (a), 4096 (b), 8192 (c) y 16384 (d).

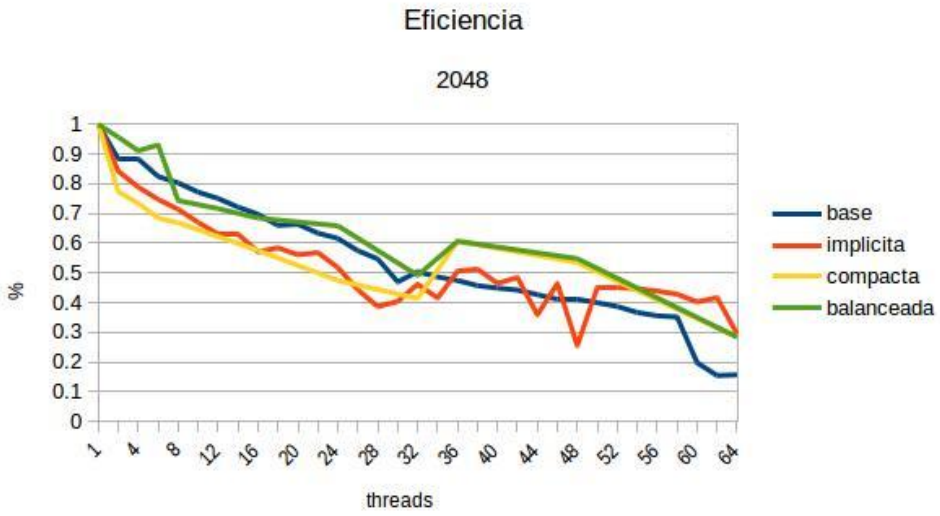
	2048		4096		8192		16384	
threads	base	Compacta	base	compacta	base	compacta	base	compacta
1	1.0000 s	1.0000 s	1.0000 s	1.0000 s	1.0000 s	1.0000 s	1.0000 s	1.0000 s
2	1.7722 s	1.5461 s	1.7749 s	1.5542 s	1.7844 s	1.5484 s	1.86746 s	1.56159 s
4	3.5421 s	2.9365 s	3.3310 s	3.0568 s	3.3699 s	2.7644 s	3.58945 s	3.07216 s
8	6.4237 s	5.3508 s	6.2473 s	5.4676 s	6.4691 s	5.7752 s	6.77029 s	5.83335 s
16	11.1610 s	9.2116 s	11.3267 s	9.9007 s	11.7064 s	10.2473 s	12.8713 s	11.3219 s
32	16.1231 s	13.2455 s	17.7343 s	17.3588 s	18.9763 s	19.9563 s	20.6579 s	21.6086 s
48	19.6893 s	25.6774 s	18.6209 s	31.4641 s	20.1514 s	33.2004 s	19.5762 s	34.2084 s
64	10.0587 s	18.2001 s	14.6424 s	25.4342 s	17.5474 s	30.8926 s	19.1080 s	-

Tabla 5.4. Aceleración de las implementaciones base y explicita (compacta) para N = 2048, 4096, 8192, 16384, y T = 1000.

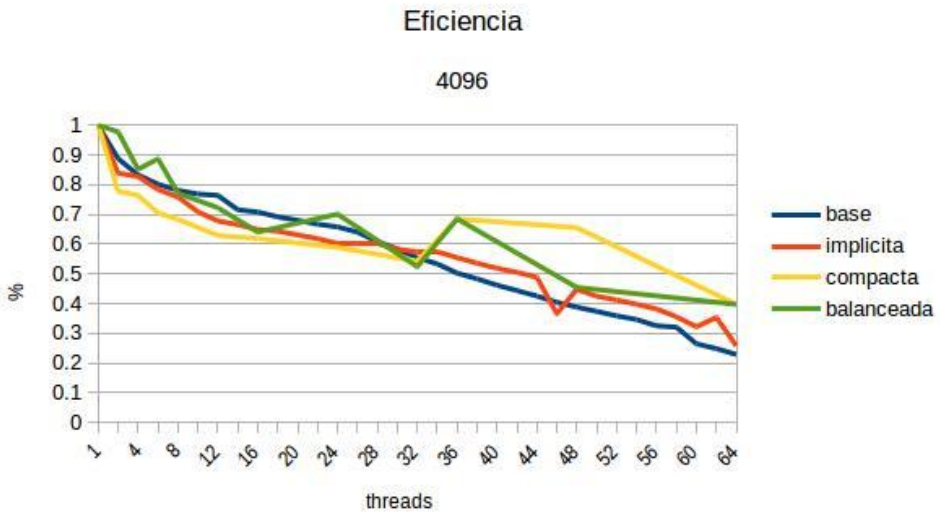
Eficiencia.

Finalmente, en la Figura 5.5 se muestra la eficiencia con respecto al número de hilos utilizados para cada implementación y tamaño de espacio de dominio considerados. Nótese la diferencia en rendimiento entre las diferentes implementaciones, al incrementar la cantidad de procesos. Esta ganancia en rendimiento se debe al incremento de la localidad, resultado de la afinidad de procesos y memoria, evitando accesos remotos y contención por el ancho de banda entre los enlaces entre sockets.

(a)



(b)



(c)

(d)

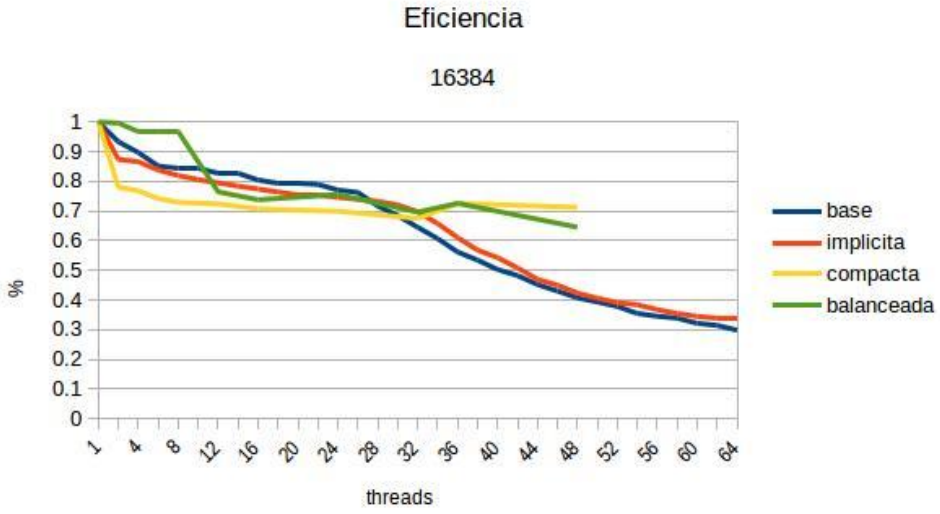


Figura 5.5. Eficiencia de las implementaciones utilizando un espacio de dominio de 2048 (a), 4096 (b), 8192 (c) y 16384 (d).

De la Figura 5.5, puede también observarse que la implementación “explícita-compacta” para un número de hilos aproximadamente mayor a 32, se mantiene una eficiencia superior a ~70x, mientras que las implementaciones “base” e “implícita” tienen una eficiencia por debajo del 50x. Esto muestra la importancia de tomar en cuenta la topología del sistema con base en el patrón de acceso a memoria requerido por el modelo de AC bajo estudio.

5.3.2 Escalamiento débil.

Por otra parte, también se realizó un análisis con base en el escalamiento débil, para evaluar las implementaciones realizadas. Para ello, en el escalamiento débil, el tamaño del problema se incrementa proporcionalmente con la cantidad de hilos, esperando que el tiempo de simulación tienda a ser aproximado para cada escenario.

Para este trabajo, el escalamiento débil se realizó utilizando únicamente un espacio de dominio con N = 2048, es decir, 4,194,304 elementos ~16MB por malla, por hilo. Este tamaño se seleccionó, debido a que ocupa un espacio mayor que la caché L3 + L2 (6 MB + 2 MB) de los equipos de prueba.

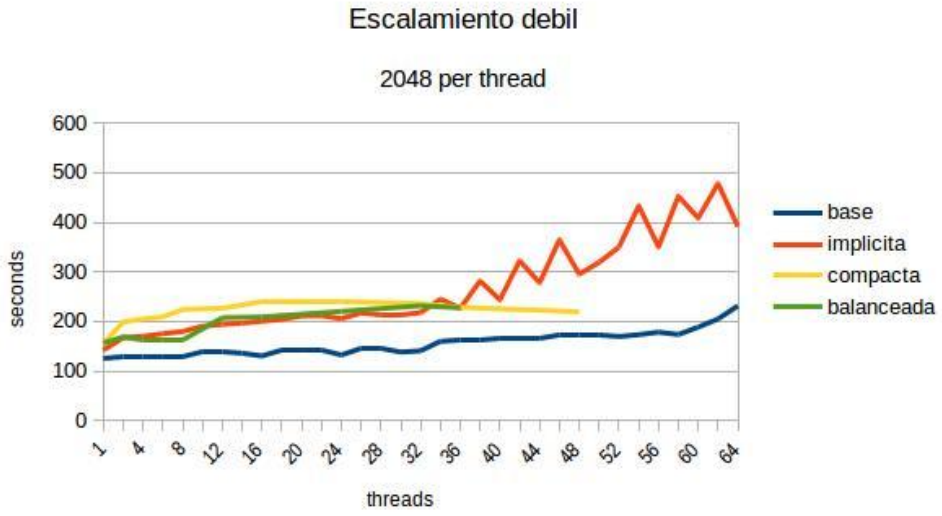


Figura 5.6. Escalamiento débil utilizando un espacio de dominio de N = 2048 por hilo y T = 1000.

En la Tabla 5.6 se muestra el escalamiento débil de las implementaciones "base" y "explicita-compacta" para NP = 1, 2, 4, 8, 16, 32, 48 y 64, y T = 1000 iteraciones y la gráfica correspondiente se muestra en la Figura 5.6. Cómo puede observarse, aun cuando el tiempo de ejecución ideal (1 hilo) de la implementación base es inferior que el de la implementación explícita compacta, esta última tiene un mejor escalamiento al mantener una tendencia a decrementar el tiempo total de ejecución conforme el número de hilos se incrementa.

threads	1	2	4	8	16	32	48	64
base	125.408 s	127.581 s	129.301 s	129.178 s	130.206 s	140.722 s	173.315 s	231.398 s
compacta	155.635 s	198.712 s	205.230 s	224.089 s	239.536 s	235.846 s	219.567 s	-

Tabla 5.6. Escalamiento débil de las implementaciones base y explicita (compacta) para N = 256, 512, 1024 y 2048 por hilo, y T = 1000 iteraciones.

5.4. Comentarios del capítulo.

En este capítulo, se presentó la evaluación de las implementaciones desarrolladas en esta tesis tomando como caso de estudio el modelo del juego de la vida. Para ello, se realizó un análisis de escalamiento fuerte y débil. Los resultados indican que, el uso de afinidad de procesos y memoria puede mejorar el rendimiento de las implementaciones paralelas de modelos de AC a gran escala, a partir de un cierto número de hilos. Aún más, los resultados indican que a medida que se considera un sistema más grande, con un mayor número de células, los beneficios son más notorios. Además, las implementaciones propuestas toman en cuenta aspectos específicos de las arquitecturas de hardware en donde se ejecutan, a diferencia de la mayoría de los estudios existentes en la literatura, que simplemente realizan una paralelización plana, en donde únicamente se limitan a dividir el trabajo entre las unidades de procesamiento de la plataforma utilizada sin considerar los accesos no uniformes a memoria.

Finalmente, en lo siguiente se presentan las conclusiones de este trabajo de tesis.

Conclusiones y trabajo futuro.

En este trabajo, se analizaron 3 implementaciones paralelas de modelos de AC a gran escala, cada una con una manera diferente de organizar los procesos y datos, respetando el patrón de accesos a memoria del modelo de AC. Para ello se abordaron los conceptos de afinidad de memoria y de procesos, mismos que fueron aplicados en las implementaciones para arquitecturas paralelas con acceso no uniforme a memoria (NUMA).

Mediante un análisis computacional de rendimiento basado en escalamiento fuerte y débil, se mostró que el rendimiento de las implementaciones propuestas, se puede incrementar significativamente al tomar en cuenta aspectos de la arquitectura de hardware en donde son ejecutadas; debido a que la localidad de los accesos a memoria tiene un alto impacto en el rendimiento de las aplicaciones. Particularmente, los resultados de la evaluación del rendimiento muestran que tanto la colocación de procesos como la asignación de memoria conscientes de la topología NUMA del sistema, mejoraron el rendimiento de la implementación paralela de los modelos de AC al utilizar un alto número de procesos e incrementar el tamaño del espacio de dominio; principalmente para el caso de la agrupación compacta. Como consecuencia, los tiempos de ejecución de la simulación se reducen debido a que al asignar la memoria en el mismo nodo en el que se ejecuta el proceso que la accede, se reduce la latencia causada por los accesos remotos y la contención por el ancho de banda de la memoria; logrando así, un incremento en la aceleración y la eficiencia.

Por lo tanto, de los resultados obtenidos, se concluye que es posible mejorar el rendimiento de las implementaciones paralelas de modelos a gran escala basados en AC, si se considera la estructura topológica de la arquitectura. De tal forma que, al aplicar tanto la afinidad de procesos como la afinidad de memoria de manera adecuada se incrementa la eficiencia y a la vez se logran algoritmos altamente escalables.

Es importante mencionar que, las implementaciones propuestas en este trabajo de tesis toman en cuenta aspectos específicos de las arquitecturas de hardware en donde se ejecutan, contrario a la mayoría de los estudios existentes en la literatura; que se limitan únicamente a dividir el trabajo entre las unidades de procesamiento de la plataforma utilizada, sin considerar los accesos no uniformes a la memoria. Este aspecto resulta así, ser un componente crítico para la optimización real de las aplicaciones basadas en HPC.

A pesar de que se logró cumplir los objetivos propuestos en este trabajo de tesis y comprobar la hipótesis planteada, también se tuvieron algunas limitaciones del trabajo realizado. Una de estas limitaciones fue que, aunque cada nodo NUMA del equipo utilizado tiene una capacidad de 16 GB de memoria RAM, con lo que podrían haberse realizado pruebas de espacios de dominio mucho más grandes, esto no fue posible. Ello se debe a que como se mencionó previamente, cada llamada para reservar memoria en un nodo NUMA especifico, a través de la función “numa_alloc_onnode ()”, crea un nuevo mapa de memoria. Sin embargo, cada proceso tiene permitido un número máximo de mapeos de memoria definido en /proc/sys/vm/max_map_count, el cual solamente puede ser modificado por el administrador del sistema. Además, realizar esta modificación, puede afectar el comportamiento de otras aplicaciones que se ejecutan en el cluster, por lo que quedó fuera del alcance de este trabajo. Por

otra parte, el esquema planteado en este trabajo, únicamente se aplica a modelos de AC con un patrón de acceso a memoria similar a los mencionados anteriormente. Al modificar el comportamiento del modelo de AC, es necesario analizar el patrón de acceso a memoria que se utilice para de acuerdo a esto realizar una propuesta, en donde este trabajo podría o no servir de base para el desarrollo de implementaciones paralelas de modelos de AC más complejos. Un análisis más detallado al respecto sería conveniente.

Trabajo futuro.

A continuación, se plantean diversas propuestas orientadas a ampliar, mejorar y/o aplicar este trabajo de tesis, como trabajo futuro, y que pueden representar posibles temas de estudio:

- Sería importante realizar estudios donde además de sólo considerar el acceso no uniforme a memoria, también se tomen en cuenta aspectos en niveles superiores de la jerarquía de memoria; como es el caso de memoria caché.
- Una evaluación de las implementaciones presentadas para diferentes modelos específicos de AC permitiría evaluar aún más la funcionalidad de las mismas y su eficiencia.
- Sería conveniente realizar un análisis más detallado basado en herramientas de perfilado de memoria, con la finalidad de identificar de mejor manera los cuellos de botella que suelen ocurrir en las arquitecturas multicore.
- Debido a que la gran mayoría de los clusters de computadoras están formados por nodos heterogéneos, valdría la pena realizar una evaluación más detallada con base en simulaciones de modelos de AC que utilicen la técnica de descomposición de dominio en arquitecturas heterogéneas.

- Por otra parte, existen modelo de AC, con comportamientos diferentes a los mencionados en este trabajo, en donde la carga de trabajo varía conforme la simulación evoluciona, por lo que sería conveniente llevar a cabo un estudio de implementaciones paralelas de modelos de AC, que hagan uso de balanceo de carga dinámico, migración de páginas de memoria, etc.; con la finalidad de permitir el desarrollo paralelo de modelos de AC más complejos y aplicables a múltiples áreas de la ciencia.

- Finalmente, la propuesta planteada en este trabajo debe ser explotada en una aplicación real de los modelos de AC a gran escala, donde actualmente hay un campo de estudio amplio. Un modelo de AC actualmente muy utilizado, en donde esta mejora podría tener un gran impacto es en la simulación del crecimiento de tumores, como se mencionó en el capítulo 2. Al aprovechar las ventajas que este trabajo presenta, se podría realizar un análisis sobre la reacción del comportamiento de los tumores a medicamentos o tratamientos en un menor tiempo.

Referencias.

[1] von Neumann J. (1966) *The Theory of Self-Reproducing Automata*. A. W. Burks (ed), Univ. of Illinois Press, Urbana and London.

[2] von Neumann, J. (1951) *The general and logical theory of automata*. In L.A. Jeffress (Ed.), Cerebral mechanisms in behavior; the Hixon Symposium Wiley. 1 - 41

[3] Gardner M. (1970) *The Fantastic Combinations of John Conway's New Solitaire Game 'Life'*. Scientific American, 223: 120–123.

[4] Berlekamp, E., Conway, J. & Guy, R. (2001) *Winning ways for your mathematical plays*. Natick, Mass: A.K. Peters.

[5] Wolfram, S. (1986) *Theory and Applications of Cellular Automata*. World Scientific, Singapore.

[6] Wolfram, S. (1994) *Cellular Automata and Complexity*. Addison-Wesley, Reading MA.

[7] Wolfram, S. (2001) *A New Kind of Science*. Wolfram Media, Champaign.

[8] Lárraga M.E., Alvarez-Icaza L. (2010) *Cellular automaton model for traffic flow based on safe driving policies and human reactions*, Physica A: Statistical Mechanics and its Applications. 389 (23): 5425-5438

[9] Bouadi, M., Jetto, K., Benyoussef, A., El Kenz, A. (2017) *The investigation of the lateral interaction effect's on traffic flow behavior under open boundaries*. Physics Letters, Section A: General, Atomic and Solid State Physics, 381 (42)

[10] Heeroo K., Gukhool O., Hoorpah D., (2016) *A Ludo Cellular Automata model for microscopic traffic flow*, Journal of Computational Science, 16: 114-127.

[11] Sfa F., Nemiche M., and Lopez R., (2015). *A Theoretical Learning Model Combining Stochastic Cellular Automata and Economic Indicators to Simulate Land Use Change*. International Journal of Applied Evolutive Computing. 6 (3): 1-8.

[12] Fuglsang, M., Münier, B., Hansen, H. S. (2013). *Modelling land-use effects of future urbanization using cellular automata: An Eastern Danish case*. Environmental Modelling & Software, 50: 1-11.

[13] Chen, X., Yu, S. X. and Zhang, Y. P. (2013). *Evaluation of Spatiotemporal Dynamics of Simulated Land Use/Cover in China Using a Probabilistic Cellular Automata-Markov Model*, In Pedosphere, 23 (2): 243-255

[14] Zahedi Sohi H., Khoshandam B., (2012). *Cellular automata modeling of non-catalytic gas–solid reactions*, In Chemical Engineering Journal, 200–202: Pages 710-719

[15] Kar S., Nag K., Dutta A., Constales D., Pal T., (2014) *An improved cellular automata model of enzyme kinetics based on genetic algorithm*, In Chemical Engineering Science, 110: 105-118

[16] Hoekstra, A. (2010). *Simulating Complex Systems by Cellular Automata* Springer-Verlag, Berlin, Heidelberg.

[17] Hanahan, D., Weinberg, R.A. (2000) *The hallmarks of cancer*. Cell, 100 (1): 57–70.

[18] Abbott R., Forrest, S., Pienta, K. (2006). *Simulating the hallmarks of cancer*. Artificial Life, 12 (4): 617–634.

[19] Monteagudo Á., Santos J. (2012) *A Cellular Automaton Model for Tumor Growth Simulation*. In: Rocha M., Luscombe N., Fdez-Riverola F., Rodríguez J. (eds) 6th International Conference on Practical Applications of Computational Biology & Bioinformatics. Advances in Intelligent and Soft Computing, 154: 147-155

[20] Santos J., Monteagudo Á. (2012). *Study of Cancer Hallmarks Relevance Using a Cellular Automaton Tumor Growth Model*. In: Coello C.A.C., Cutello V., Deb K., Forrest S., Nicosia G., Pavone M. (eds) Parallel Problem Solving from Nature - PPSN XII. PPSN 2012. Lecture Notes in Computer Science, 7491:489-499

[21] Monteagudo Á., Santos J. (2013). *Cancer Stem Cell Modeling Using a Cellular Automaton*. In: Ferrández Vicente J.M., Álvarez Sánchez J.R., de la Paz López F., Toledo Moreo F.J. (eds) Natural and Artificial Computation in Engineering and Medical Applications. IWINAC 2013. Lecture Notes in Computer Science, 7931:21-31

[22] Monteagudo Á., Santos J. (2014) *Studying the capability of different cancer hallmarks to initiate tumor growth using a cellular automaton simulation*. Application in a cancer stem cell context, In Biosystems, 115: 46-58

[23] Santos J., Monteagudo Á. (2015) *Analysis of behaviour transitions in tumor growth using a cellular automaton simulation*, IET Systems Biology, 9 (3): 75-87.

[24] Poleszczuk, J., & Enderling, H. (2014). *A High-Performance Cellular Automaton Model of Tumor Growth with Dynamically Growing Domains*. Applied Mathematics, 5(1), 144–152.

[25] Boondirek A, Triampo W, Nuttavut N (2010). *A review of cellular automata models of tumor growth*. International Mathematical Forum, 5: 3023–3029.

[26] Butler J., Mackay F., Denniston C., Daley M. (2014). *Simulating Cancer Growth Using Cellular Automata to Detect Combination Drug Targets*. In: Ibarra O., Kari L., Kopecki S. (eds) Unconventional Computation and Natural Computation. UCNC 2014. Lecture Notes in Computer Science, 8553: 67-79

[27] Butler, J., Mackay, F., Colin D., Daley M (2016). *Halting the hallmarks: A cellular automaton model of early cancer growth inhibition*. Natural Computing 15 (1): 15-30.

[28] Flynn, M. (1972). *Some Computer Organizations and Their Effectiveness*, IEEE Trans. Comput., C-21: 948

[29] Hager, G. & Wellein, G. (2011). *Introduction to high performance computing for scientists and engineers.* Boca Raton, FL: CRC Press.

[30] Rauber, T. & Rünger, G. (2013). *Parallel programming: for multicore and cluster systems*. Heidelberg: Springer-Verlag.

[31] Hennessy, J., Patterson, D., Asanović, K. (2012). *Computer architecture: a quantitative approach 5th ed.* Waltham, MA: Morgan Kaufmann.

[32] Stallings, W. (2016). *Computer organization and architecture: designing for performance.* Boston: Pearson-Prentice Hall.

[33] Patterson, D. & Hennessy, J. (2012). *Computer organization and design: the hardware/software interface*. Waltham, MA: Morgan Kaufmann.

[34] G. M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities.* In: AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (ACM, New York, NY, USA), 483–485.

[35] Gustafson, J.L. (1988) *Reevaluating Amdahl's law*. Commun. *ACM* 31 (5): 532-533.

[36] McCool, M., Reinders, J. & Robison, A. (2012). *Structured parallel programming: patterns for efficient computation*. Amsterdam Boston: Elsevier/Morgan Kaufmann.

[37] Breshears, C. (2009). *The art of concurrency 1st ed*. Farnham: O'Reilly.

[38] Foster, I. (1995). *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc.,

[39] Regueira D., Iturriaga S., Nesmachnow S. (2017) *Communication-Aware Affinity Scheduling Heuristics in Multicore Systems*. In: Barrios Hernández C., Gitler I., Klapp J. (eds) High Performance Computing. CARLA 2016. Communications in Computer and Information Science, 697: 33-48.

[40] Alves, M.A., Cruz, E.H., Diener, M., Koren, I., Navaux, P.O. (2016). *Affinity-Based Thread and Data Mapping in Shared Memory Systems*. ACM Comput. Surv., 49 (4):1-64.

[41] Ntinas V.G., Moutafis B.E., Trunfio G.A., Sirakoulis G.C. (2016) *GPU and FPGA Parallelization of Fuzzy Cellular Automata for the Simulation of Wildfire Spreading*. In: Wyrzykowski R., Deelman E., Dongarra J., Karczewski K., Kitowski J., Wiatr K. (eds) Parallel Processing and Applied Mathematics. Lecture Notes in Computer Science, 9574: 560-569

[42] Millán, E.N., Bederián, C., Piccoli, F., Garcia, C., Bringa, E. (2015). *Performance analysis of Cellular Automata HPC implementations.* Computers & Electrical Engineering. 48: 12-24

[43] Gibson, M.J., Keedwell, E.C., Savić, D.A. (2015) *An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware,* In Journal of Parallel and Distributed Computing, 77: 11-25

[44] Dogaru, I., Dogaru, R. (2014) *A comparative study of several 2D cellular automata implementations in FPGA,* 2014 International Symposium on Fundamentals of Electrical Engineering (ISFEE), 1-4.

[45] Topa, P. (2014) *Cellular Automata Model Tuned for Efficient Computation on GPU with Global Memory Cache,* 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Torino, 380-383.

[46] Millán, E.N., Martínez, P.C., Costa, G.V.G., Piccoli, M.F., Printista, A.M., Bederian, C. (2013). *Parallel implementation of a cellular automata in a hybrid CPU/GPU environment.* In: De Giusti A, editor. XVIII Congreso Argentino de Ciencias de la Computación, Red de Universidades con Carreras en Informática RedUNCI; 184–93

[47] Bandman, O. (2013) *Implementation of large-scale cellular automata models on multi-core computers and clusters,* 2013 International Conference on High Performance Computing & Simulation (HPCS), Helsinki. 304-310.

[48] Kalgin, K.V. (2012) *Parallel implementation of asynchronous cellular automata on a 32-core computer,* Numerical Analysis and Applications. 5(1): 45–53

[49] Drieseberg, J., Siemers C. (2012) *C to Cellular Automata and execution on CPU, GPU and FPGA,* 2012 International Conference on High Performance Computing & Simulation (HPCS), 216-222.

[50] Vourkas, I., Sirakoulis, G.C., (2012) *FPGA based cellular automata for environmental modeling*, 19th IEEE International Conference on Electronics, Circuits, and Systems, 93-96.

[51] Bezbradica, M., Crane, M., Ruskin, H.J. (2012) *Parallelisation strategies for large scale cellular automata frameworks in pharmaceutical modelling,* 2012 International Conference on High Performance Computing & Simulation (HPCS), 223-230

[52] Oliverio, M., Spataro, W., D'Ambrosio, D., Rongo, R., Spingola, G., Trunfio, G.A. (2011) *OpenMP parallelization of the SCIARA Cellular Automata lava flow model: Performance analysis on shared-memory computers*, In Procedia Computer Science, 4: 271-280

[53] Chorley, M.J., Walker, D.W. (2010) *Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters,* In Journal of Computational Science, 1 (3): 168-174

[54] Kalgin K. (2010) *Comparative Study of Parallel Algorithms for Asynchronous Cellular Automata Simulation on Different Computer Architectures.* In: Bandini S., Manzoni S., Umeo H., Vizzari G. (eds) Cellular Automata. ACRI 2010. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg 6350: 399-408

[55] Rybacki, S., Himmelspach, J., Uhrmacher, A.M. (2009) *Experiments with Single Core, Multi-core, and GPU Based Computation of Cellular Automata*, 2009 First International Conference on Advances in System Simulation, Porto, 62-67.

[56] Guisado, J.L., Jiménez-Morales, F., Vega, F. (2007). *Cellular Automata and Cluster Computing: an Application to the Simulation of Laser Dynamics. Advances in Complex Systems.* 10: 167-190

[57] Culler, D., Singh, P., Gupta, A. (1999). *Parallel Computer Architecture: A Hardware-Software Approach*. In: Parallel Programs., Chap. 2