UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Algoritmos paralelos sobre cluster de *multicore*. Aplicación a problemas de simulación.
Autor: Silvana Lis Gallo
Director: Ricardo Marcelo Naiouf
Codirector: Laura Cristina De Giusti
Asesor profesional: --
Carrera: Licenciatura en Sistemas - Plan 2003

Resumen

El objetivo de la presente Tesina es realizar la investigación y desarrollo de algoritmos paralelos sobre arquitecturas de soporte actuales de múltiples núcleos (como multicores y cluster de multicores), con énfasis en la aplicación a problemas con alta demanda computacional.

Realizar el estudio de los temas de investigación derivados en lo que refiere tanto a las técnicas de diseño y desarrollo de algoritmos como a su evaluación en plataformas de memoria compartida, mensajes e híbridas con el objetivo de lograr eficiencia en el uso de los múltiples núcleos.

Analizar la posibilidad de aplicación de los resultados obtenidos en la solución de otros problemas de cómputo científico similares, desde el punto de vista de los requerimientos computacionales que plantea el caso de aplicación.

Para ello se buscó como caso de aplicación la solución al problema Sharks and Fishes, que implica realizar una simulación de la vida de varias especies (en este caso tiburones y peces) en el océano.

Palabras Claves

Algoritmos paralelos y distribuidos.
Multicore. Cluster de multicore.
Evaluación de performance.
Simulación.

Trabajos Realizados

Se estudiaron los fundamentos del procesamiento paralelo para poder desarrollar, analizar y evaluar diferentes soluciones al problema Sharks and Fishes, las cuales emplean distintos modelos de comunicación: memoria compartida (con OpenMP y con Pthreads), pasaje de mensajes (MPI) y diferentes soluciones híbridas (MPI+OpenMP y MPI+Pthreads).

Conclusiones

Finalizado el trabajo experimental, se analizaron y compararon los comportamientos de las soluciones implementadas. Puede observarse que al comparar los algoritmos puros el que mejor se comporta es el de OpenMP. Por otro lado, al trabajar con más de una hoja, se compara el algoritmo de pasaje de mensajes con los híbridos, y como resultado se obtiene que al crecer el tamaño de la máquina con la que se trabaja el algoritmo hibrido con MPI+OpenMP se comporta mejor que el resto, incrementando cada vez la diferencia.

Trabajos Futuros

Extender el estudio de escalabilidad para tamaños de problema más grandes
Estudiar alternativas de solución mediante estrategias optimistas.
Extender la aplicación de las estrategias y los resultados obtenidos a otros problemas de cómputo científico que presenten características similares.
Incorporar heterogeneidad en la arquitectura, estudiando el impacto sobre el mapeo de procesos y datos a procesadores.

Fecha de la presentación: **Marzo 2012**

Agradecimientos

Más allá de todo lo que escrito en el desarrollo de la tesina no quería olvidarme de dedicar algunas palabras de agradecimiento a todas aquellas personas que durante estos meses de trabajo me brindaron su apoyo y acompañamiento, no sólo ante las dudas o problemas que se me presentaban desde la parte "técnica" sino también el apoyo brindado desde lo afectivo, como amigos y compañeros de trabajo.

Quiero agradecer a mi familia, que de pequeña me enseñaron que con esfuerzo y perseverancia todo se logra. Me dieron la posibilidad de elegir lo que realmente me gusta y me incentivaron a seguir esta carrera, decisión de la que hasta el día de hoy no me arrepiento.

A mi Director y Codirectora de tesina, Marcelo y Laura, por su predisposición y apoyo incondicional, por su tiempo y dedicación.

Otro agradecimiento especial al equipo de trabajo del III-LIDI que me dieron la posibilidad de ser parte del mismo. Gracias por la comprensión, tiempo, trabajo y dedicación, por ayudarme a resolver mis dudas y problemas, simplemente gracias por estar siempre.

A mis compañeros de "Box", Emma, Fabi, Enzo, Seba, Fer, Mariano y Erica, que más que compañeros son amigos. Siempre cerca para darme una mano o aconsejarme no sólo en lo laboral sino también en la vida.

También les agradezco mucho a Adrián, Franco y Fernando quienes supieron orientarme correctamente ante las consultas que les planteaba.

Gracias a todos los compañeros y amigos, con los que he comenzado el paso por esta Facultad. Por las tardes de mates y estudio en el anexo de 49, en la biblioteca y las juntadas en la plaza a "leer". A Tammy, Cristian, Pato, Nati, Augus y uff!! la lista es larga..

Por último, no me quiero olvidar de todos los profesores y ayudantes que he tenido a lo largo de mi carrera, que supieron trasmitirme todos sus conocimientos, sabidurías y experiencias para poder afrontar los desafíos que se presenten en un futuro como profesional.

Índice de contenidos

Introducción………… 1

Objetivo de la Tesina………… 3

Organización del documento………… 3

Capítulo 1: Cómputo y Sistemas Paralelos………… 4

Procesamiento Paralelo………… 4

Resolver problemas más grandes………… 4

Resolver problemas con límite de tiempo………… 4

Resolver problemas con mayor precisión………… 5

Límites del procesamiento serial………… 5

Arquitecturas paralelas………… 5

Máquinas de memoria compartida y de memoria distribuida………… 6

Clusters………… 6

Multicores………… 7

Cluster de multicores………… 7

Modelos de programación paralela………… 8

Memoria compartida………… 8

Pasaje de mensajes………… 8

Pthreads………… 9

OpenMP………… 10

MPI………… 10

Evaluación de performance en aplicaciones paralelas………… 11

Tiempo de ejecución………… 11

Tamaño del problema………… 11

Speedup………… 12

Overhead paralelo………… 12

Eficiencia………… 13

Costo………… 13

Grado de concurrencia………… 13

La Ley de Amdahl………… 14

Evolución de la Ley de Amdahl: Ley de Gustafson-Barsis………………….. 14
Escalabilidad……………………………………………………………….. 15
Optimizaciones…………………………………………………………….. 15
Métodos de mapeo………………………………………………………… 17
Memoria compartida (alternativa 1)…………………………………………… 17
Memoria distribuida (Pasaje de mensajes)(alternativa 2)…………………… 18
Mapeo en cluster de multicore…………………………………………………. 19
Capítulo 2: Conceptos de Simulación………………………………………………… 20
Etapas del proceso de simulación……………………………………………….. 22
Elementos de un modelo de simulación………………………………………. 24
Cambios de estado en el sistema………………………………………………… 25
Ejecución time-stepped……………………………………………………... 27
Ejecución event-driven……………………………………………………… 27
Tipos de soluciones………………………………………………………………… 28
Programas de simulación discreta…………………………………………… 28
Comienzo y fin de la simulación……………………………………………….. 29
Capítulo 3: Simulaciones paralelas y distribuidas…………………………………… 31
Enfoques de simulaciones………………………………………………………… 32
Autómatas celulares………………………………………………………… 33
Algoritmos conservativos…………………………………………………… 33
Algoritmos optimistas………………………………………………………… 36
Sincronización conservativa vs. Sincronización optimista………………….. 36
Autómatas celulares y el mecanismo de sincronización……………………. 37
Capítulo 4: Descripción del problema y su solución secuencial…………………… 39
Reglas para la población de peces………………………………………………. 39
Reglas para la población de tiburones…………………………………………… 39
Características del océano o escenario…………………………………………. 40
Solución Secuencial……………………………………………………………….. 40
Observaciones………………………………………………………………… 41
Pruebas realizadas…………………………………………………………………. 41
Capítulo 5: Soluciones implementadas………………………………………………. 43
Soluciones de Memoria Compartida……………………………………………… 43
Solución Pthreads…………………………………………………………….. 43
Observaciones………………………………………………………………. 44

Pruebas realizadas………………………………………………………………… 45
Solución OpenMP…………………………………………………………………… 45
Observaciones……………………………………………………………………… 45
Pruebas realizadas………………………………………………………………… 46
Solución de Pasaje de Mensajes……………………………………………………… 46
Observaciones……………………………………………………………………… 49
Pruebas realizadas………………………………………………………………… 49
Soluciones Híbridas………………………………………………………………… 49
Solución MPI+Pthreads…………………………………………………………… 49
Observaciones……………………………………………………………………… 50
Pruebas realizadas………………………………………………………………… 51
Solución MPI+OpenMP…………………………………………………………… 51
Observaciones……………………………………………………………………… 52
Pruebas realizadas………………………………………………………………… 52
Capítulo 6: Resultados obtenidos……………………………………………………….. 53
Descripción de la Plataforma y Pruebas Realizadas…………………………… 53
Resultados Obtenidos……………………………………………………………….. 54
Solución Secuencial………………………………………………………………. 54
Soluciones paralelas con Memoria Compartida……………………………… 54
Solución paralela con Pasaje de Mensajes…………………………………... 58
Comparación de las tres soluciones puras (OpenMP, Pthreads y MPI)…... 60
Soluciones paralelas Híbridas MPI + (OpenMP o Pthreads)……………….. 62
Comparación de las soluciones híbridas y la de Pasaje de Mensajes…….. 66
Capítulo 7: Conclusiones y Trabajos Futuros……………………………………….. 69
Bibliografía………………………………………………………………………………… 71
Anexo I: Resultados completos………………………………………………………. 73
Resultados Obtenidos de la solución con OpenMP……………………………. 73
Resultados Obtenidos de la solución con Pthreads……………………………. 74
Resultados Obtenidos de la solución con Pasaje de Mensajes………………. 76
Resultados Obtenidos de la solución híbrida con Pthreads…………………… 77
Resultados Obtenidos de la solución híbrida con OpenMP…………………… 79

Introducción

Más allá de la evolución constante en las arquitecturas físicas de cómputo, uno de los mayores desafíos se centra en cómo aprovechar al máximo la potencia que brindan. En la década pasada, las CPU han mejorado su rendimiento y, junto a las mejoras introducidas en los compiladores, incrementa el número de instrucciones ejecutables por unidad de tiempo.

Sin embargo, el aumento de temperatura de los procesadores ha impuesto un límite, y la tendencia ha sido crear procesadores de múltiples núcleos (*cores*) para mantener el crecimiento del número de instrucciones ejecutables por unidad de tiempo. En consecuencia, los diseños de hardware, software, lenguajes y algoritmos se han encontrado impactados fuertemente por este cambio tecnológico.

El software puede beneficiarse por el uso de multicores cuando puede ejecutarse en paralelo. En la mayoría de los sistemas operativos comunes esto requiere que el código ejecute en *threads* o procesos separados. De esta manera, múltiples aplicaciones pueden beneficiarse con la ejecución en multicores. Modificar un programa secuencial para distribuirlo y que ejecute en paralelo mediante múltiples procesos o hilos tiene dos principales razones: acelerar la velocidad de ejecución y/o mejorar la cantidad de memoria disponible. Para ello, existen alternativas: (a) utilizar clusters, y (b) utilizar multicores.

Una tercera alternativa consiste en un sistema híbrido que combina las características de ambos (memoria compartida y comunicación por mensajes), lo que introduce una modificación en la jerarquía de memorias, e incrementar aún más la capacidad y el poder de los sistemas computacionales (clusters de multicores). Pero, el diseño de aplicaciones para este tipo de sistema trae aparejado una serie de complejidades y es necesario estudiar la utilización de diferentes lenguajes de programación ya que si bien puede mencionarse el uso de MPI, OpenMP y Pthreads, aún no se cuenta con un standard establecido.

Las aplicaciones donde resulta imprescindible el uso del paralelismo pertenecen a numerosas áreas, siendo característico en ellas la gran cantidad de computación a realizar (simulaciones, modelización, biología, física, química, etc.). La ganancia en performance por el uso de multicore depende del problema a resolver, los algoritmos, y su implementación en software. En este punto cobra importancia el desarrollo de técnicas que exploten las ventajas de los multicore para acelerar las aplicaciones paralelas con gran demanda de cómputo, y es esencial poder desarrollar estrategias y algoritmos que optimicen la comunicación y sincronización de procesos de diferentes núcleos (del mismo procesador o de diferentes en un cluster).

La performance de la solución paralela está dada por una compleja relación con numerosos factores y existen diferentes métricas para evaluarla, siendo las tradicionales el tiempo de ejecución, el speedup y la eficiencia. Además de estas se encuentran: costo, overhead, grado de concurrencia, escalabilidad, etcétera. Esto se torna más complejo si cada nodo puede ser un multicore con varios niveles de memoria. Las mejoras que se obtienen al utilizar procesadores con múltiples núcleos se reflejan a través del paralelismo desarrollado para las aplicaciones y su correcto mapeo en la arquitectura.

La simulación por eventos discretos es una técnica informática de modelado dinámico de sistemas. Frente a su homóloga, la simulación de tiempo continuo, ésta se caracteriza por un control en la variable del tiempo que permite avanzar a éste a intervalos variables, en función de la planificación de ocurrencia de tales eventos a un tiempo futuro. Un requisito para aplicar esta técnica es que las variables que definen el sistema no cambien su comportamiento durante el intervalo simulado. Estos sistemas se caracterizan por mantener un estado interno global del sistema, que puede no obstante estar física o lógicamente distribuido, y que cambia parcialmente debido a la ocurrencia de un evento.

La simulación de sistemas es un área en constante evolución, y existen varias razones para pensar en el desarrollo de modelos de simulación mediante herramientas paralelas. Entre ellas se encuentran:

- El alto tiempo de duración de la simulación: debido a la gran extensión del mismo, mediante la división del cómputo de la simulación y la ejecución concurrente de las diferentes componentes, se puede lograr una reducción significativa del tiempo de ejecución.

- La necesidad de cómputo: si es necesario simular un escenario muy complejo, la carga de cómputo será mayor, por lo cual habrá que tener en cuenta ciertos parámetros a la hora de la implementación de la misma, y conforme a la arquitectura utilizada para la ejecución.

Actualmente, la simulación de sistemas ocupa una amplia etapa en los proyectos de varias disciplinas, ya sea porque se necesita tener una predicción de una situación futura y los altos costos de realizar el proyecto sin tener una orientación en ese aspecto lo requieren, o bien se necesita tener una representación de un sistema con ciertas condiciones preestablecidas.

En este tipo de sistemas en general, las necesidades de cómputo son altas, y en este sentido se trabaja en el desarrollo de algoritmos que permiten optimizar la ejecución de aplicaciones paralelas de simulación sobre multicore y cluster de multicore. Es interesante analizar diferentes modelos de solución a problemas de simulación de eventos discretos, utilizando programación por memoria compartida, por mensajes e híbrida. Se realiza el análisis de performance de las alternativas en relación a métricas de speedup, eficiencia y escalabilidad. Asimismo, a futuro se espera analizar la posibilidad de aplicación de los

resultados obtenidos a otros problemas de cómputo científico similares, desde el punto de vista de los requerimientos computacionales que plantea el caso de aplicación.

Objetivo de la Tesina

Se plantea realizar la investigación y desarrollo de algoritmos paralelos sobre arquitecturas de soporte actuales de múltiples núcleos (como multicores y cluster de multicores), con énfasis en la aplicación a problemas con alta demanda computacional (como es el caso de las simulaciones de eventos discretos).

Se propone realizar el estudio de los temas de investigación derivados en lo que refiere tanto a las técnicas de diseño y desarrollo de algoritmos como a su evaluación en plataformas de memoria compartida, mensajes e híbridas con el objetivo de lograr eficiencia en el uso de los múltiples núcleos.

Analizar la posibilidad de aplicación de los resultados obtenidos en la solución de otros problemas de cómputo científico similares, desde el punto de vista de los requerimientos computacionales que plantea el caso de aplicación.

Organización del documento

En el Capítulo 1 se realiza una introducción a los conceptos de programación paralela, métricas y lenguajes. En el Capítulo 2 se presentan los conceptos generales de modelos, sistemas y simulación. En el Capítulo 3 se introduce la problemática de las simulaciones paralelas y distribuidas. En el Capítulo 4 se plantea el problema de aplicación elegido y su solución secuencial. En el Capítulo 5 se describen las soluciones utilizando memoria compartida, mensajes e híbrida. En el Capítulo 6 se muestran y analizan los resultados obtenidos en la experimentación. Finalmente, en el Capítulo 7 se plantean las conclusiones y posibles trabajos futuros.

Capítulo 1: Cómputo y Sistemas Paralelos

Procesamiento Paralelo

Reducir el tiempo de ejecución de aplicaciones con grandes requerimientos de procesamiento es el objetivo principal del procesamiento paralelo. Es decir, disminuir los tiempos de ejecución con respecto a los tiempos secuenciales. Esto se debe al constante incremento del volumen de procesamiento y las limitaciones que impone el cómputo secuencial en cuanto a tiempos de respuesta, acceso a datos distribuidos y manejo de la concurrencia implícita en los problemas del mundo real.

El procesamiento paralelo implica la existencia de múltiples procesadores y se utiliza en una gama muy amplia de aplicaciones entre las cuales podemos citar el cómputo científico, las simulaciones, el procesamiento de imágenes, etc. Asimismo, el tipo y organización de los múltiples procesadores de las arquitecturas de soporte, influye en el diseño de las aplicaciones para cumplir con el objetivo propuesto.

Existen diversas razones que permiten explicar la importancia del paralelismo. A continuación se analizan las principales de ellas.

Resolver problemas más grandes

Algunos problemas son tan grandes y/o complejos que resulta poco práctico (y en algunos casos, imposible) resolverlos utilizando una computadora individual, en especial aquellas que poseen una memoria limitada. Por ejemplo, el modelado de enormes estructuras de ADN. Éste problema forma parte de una familia de problemas denominados problemas *Grand Challenge*. Un problema Grand Challenge es aquel que no puede ser resuelto en una cantidad de tiempo razonable con las computadoras estándares. Este tipo de problema suelen requerir enormes cantidades de cálculos repetitivos sobre grandes cantidades de datos para obtener resultados [BAR10][WIL05].

Resolver problemas con límite de tiempo

Algunos problemas tienen un límite temporal para realizar sus cálculos, por ejemplo el pronóstico del tiempo. Si se necesita de dos días para poder pronosticar el tiempo del día siguiente, entonces la predicción es inútil. En la industria, los cálculos de los ingenieros y las simulaciones deben lograrse en segundos o minutos, de ser posible. Una simulación que toma dos semanas para alcanzar una solución es usualmente inaceptable en un ambiente de diseño, ya que el tiempo de la simulación debe ser lo suficientemente corto como para que el diseñador pueda trabajar de forma efectiva [WIL05].

Resolver problemas con mayor precisión

Además de obtener el potencial para aumentar la velocidad de un problema determinado, el uso de múltiples computadoras/unidades de procesamiento permite a veces una solución más precisa de un problema a ser resuelto en una cantidad razonable de tiempo. Por ejemplo, pronosticar el clima requiere dividir el aire en una cuadrícula tridimensional donde cada celda representa una solución. Utilizar múltiples computadoras o unidades de procesamiento permite a menudo calcular más soluciones en un tiempo dado, y por lo tanto una solución más precisa [WIL05].

Límites del procesamiento serial

Existen razones tanto físicas como prácticas que ponen límites significativos a la construcción de computadoras secuenciales cada vez más rápidas. Entre ellas se encuentran:

- Velocidades de transmisión: la velocidad de una computadora serial depende directamente de cuan rápido puede mover los datos a través del hardware. Como límites absolutos se encuentran la velocidad de la luz (30 cm/ns) y el límite de transmisión del cable de cobre (9cm/ns). Incrementar la velocidad implica aproximar las unidades de procesamiento.

- Límites a la construcción a pequeña escala: la tecnología actual de las unidades de procesamiento está permitiendo colocar una gran cantidad de transistores por chip. Sin embargo, aún con componentes de nivel molecular o atómico, se alcanzará un límite con respecto al tamaño de los mismos.

- Limitaciones económicas: resulta cada vez más caro hacer que las unidades de procesamiento individuales sean más rápidas. En lugar de eso, se puede obtener el mismo rendimiento (o mejor) al utilizar un mayor número de unidades de procesamiento moderadamente veloces, lo cual resulta más barato.

Durante las últimas décadas, la tendencia ha sido hacia redes cada vez más veloces, sistemas distribuidos y arquitecturas con múltiples unidades de procesamiento (aún a nivel de escritorio), lo cual claramente demuestra que el futuro del procesamiento es el paralelismo [BAR10].

Arquitecturas paralelas

Una plataforma paralela consiste de dos o más unidades de procesamiento vinculadas a partir de algún tipo de red de interconexión. Podemos clasificar a las plataformas paralelas según su organización lógica (la manera en que la ve el programador), así como también según su organización física (el hardware real de la plataforma).

Máquinas de memoria compartida y de memoria distribuida

Un multiprocesador de memoria compartida consiste de múltiples unidades de procesamiento conectadas a múltiples módulos de memoria. La conexión entre unidades de procesamiento y memorias se da a través de algún tipo de red de interconexión, como puede ser un bus o un switch crossbar. En un sistema de memoria compartida, sólo existe un único espacio de direcciones, por lo que cualquier dirección de memoria es accesible por todas las unidades de procesamiento. Si el tiempo que le toma a una unidad de procesamiento acceder a una dirección de memoria es idéntico para todas las unidades de procesamiento, entonces se dice que el multiprocesador tiene acceso uniforme a memoria (UMA). Si el tiempo que toma acceder a determinadas direcciones de memoria es mayor que a otras, el multiprocesador es llamado de acceso no uniforme a memoria (NUMA).

En cualquier caso, una memoria caché de alta velocidad está presente para mantener los contenidos de las direcciones de memoria principal recientemente referenciados. La presencia de cachés en las unidades de procesamiento también acarrea la problemática de tener múltiples copias de una única palabra de memoria siendo manipulada por más de una unidad de procesamiento al mismo tiempo. El mecanismo que permite que múltiples operaciones concurrentes sobre una misma palabra de memoria tengan una semántica bien definida se llama coherencia de caché.

Lógicamente, una plataforma de memoria distribuida consta de p nodos de procesamiento unidos por una red de interconexión. Cada uno nodo puede ser una computadora individual o un multiprocesador de memoria compartida. De cualquier forma, cada nodo posee su propio espacio de direcciones, el cual no es accesible por el resto. La red de interconexión permite a los nodos enviarse mensajes entre ellos. Éste intercambio de mensajes es utilizado para transferir datos, trabajo y sincronizar acciones entre nodos. Resulta más fácil escalar físicamente a los sistemas de memoria distribuida que a los sistemas de memoria compartida.

Clusters

Un cluster es una colección de computadoras individuales interconectadas que trabajan en conjunto como un único recurso integrado de computación. Lo que caracteriza a los clusters es que cada nodo de procesamiento es un sistema de cómputo en sí mismo, dotado de características de hardware y sistema operativo propios. Aún más, un nodo podría ser incluso un multiprocesador de memoria compartida. Los componentes de un cluster usualmente se encuentran interconectados mediante algún tipo de red de alta velocidad, como una LAN, y en muchos aspectos pueden aparecer como un sistema individual para usuarios y aplicaciones.

Este tipo de sistemas ofrece una manera rentable de mejorar el rendimiento (velocidad, confiabiliadad, disponibilidad, rendimiento, etc.) comparado con supercomputadoras de similares características.

Cuando todas las máquinas que componen un cluster tienen las mismas características, el cluster es homogéneo. De otro modo, se dice que es heterogéneo.Los clusters permiten obtener un alto rendimiento a un bajo costo. Si a esto se suma que pueden expandirse fácilmente, se explica por qué la computación con clusters es hoy una de las formas de computación paralela/distribuida más populares.

Multicores

Históricamente se ha buscado incrementar el poder computacional de los sistemas de computación. En 1965, Gordon Moore, uno de los fundadores de Intel, afirmó que el número de transistores en un circuito integrado se duplicaría cada 18 meses. Aunque Esta afirmación se mantiene en vigencia y es lo que conocemos como Ley de Moore. Sin embargo, hoy en día resulta cada vez más difícil acelerar la velocidad de los procesadores incrementando la frecuencia de los mismos.

Son dos los problemas que los arquitectos de hardware deben enfrentar. El primero es la generación de calor y el segundo es el consumo de energía. La solución que presentaron los arquitectos de hardware a estos problemas ha sido integrar dos o más núcleos computaciones dentro de un mismo chip, lo cual se conoce como procesador multicore o multinúcleo. Los procesadores multicore mejoran el rendimiento de una aplicación al distribuir el trabajo entre los núcleos disponibles [McC07].

Existen diferentes alternativas con respecto a la organización de la jerarquía de caché. Normalmente, cada uno de los núcleos de un multicore posee su propio nivel L1 de caché y comparte de a pares el nivel L2 (Figura 1.1).

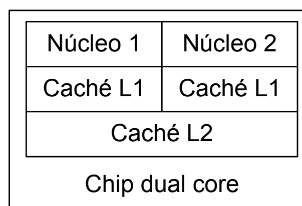


Figura 1.1. Arquitectura de procesador dual core o doble núcleo.

Cluster de multicores

La aparición de la tecnología multicore ha impactado sobre los clusters, introduciéndolos en una nueva etapa. Un cluster de multicores es similar a un cluster tradicional, sólo que en lugar de monoprocesadores tenemos procesadores multicore [CGP07] [CHA07]. Los clusters de multicores agregan un nivel más a la jerarquía de memoria de los clusters tradicionales. Además de la caché compartida entre pares de núcleos y la memoria compartida entre todos los núcleos de un mismo procesador, contamos con la memoria distribuida accesible vía red (Figura 1.2).

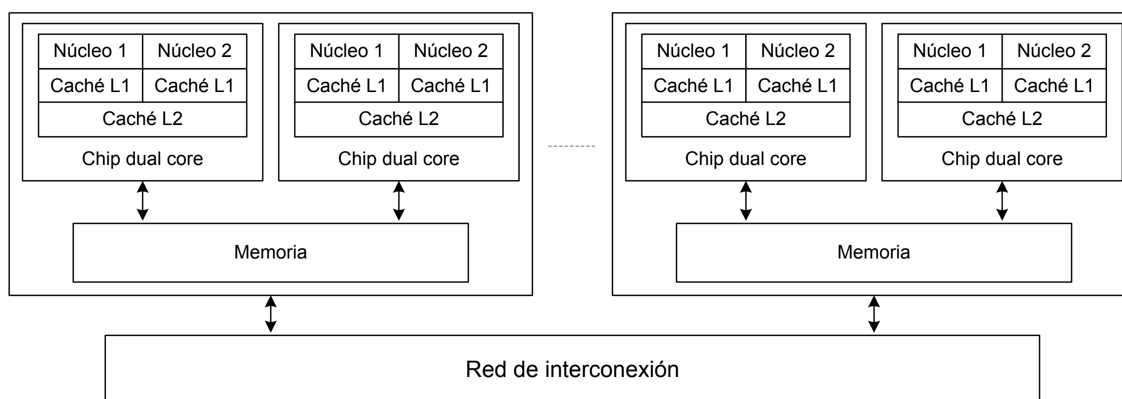


Figura 1.2. Una posible arquitectura cluster de multicores.

A la hora de escribir un algoritmo paralelo, no puede obviarse la jerarquía de memoria presente en los clusters de multicores, ya que esta incide sobre el rendimiento alcanzable por los mismos. Por el contrario, se deben estudiar nuevas técnicas de programación de algoritmos paralelos que permitan aprovechar de manera eficiente la potencia de estas arquitecturas [RAU10].

Modelos de programación paralela

Diversos lenguajes de programación y librerías han sido desarrollados para la programación paralela explícita. Estas difieren principalmente en la manera en que el usuario ve al *espacio de direcciones*. Los modelos se dividen básicamente en los que proveen un espacio de direcciones compartido o uno distribuido, aunque también existen modelos híbridos que combinan las características de ambos. El espacio de direcciones influye significativamente sobre la manera en que los hilos o procesos intercambian la información. A continuación se detallan los diferentes modelos [GRA05].

Memoria compartida: todos los datos accedidos por la aplicación se encuentran en una memoria global accesible por todos los procesadores paralelos. Esto significa que cada procesador puede buscar y almacenar datos de cualquier posición de memoria independientemente. Este modelo se caracteriza por la necesidad de la sincronización para preservar la integridad de las estructuras de datos compartidas [DON03].

Pasaje de mensajes: los datos son vistos como si estuvieran asociados a un procesador particular. De esta manera, se necesita de la comunicación entre ellos para acceder a un dato remoto. Generalmente, para acceder a un dato que se encuentra en una memoria remota, el procesador dueño de ese dato debe enviar el dato y el procesador que lo requiere debe recibirlo, y el dato viaja a través de un *canal*. En este modelo, las primitivas de envío y recepción son las encargadas de manejar la sincronización [DON03]. Las variantes en el envío y recepción (sincrónica o asincrónica), y a las características de los canales de comunicación (uni- o bidireccionales), da lugar a los diferentes mecanismos de comunicación).

Debido al avance de las arquitecturas paralelas y especialmente a la aparición de la arquitectura de múltiples núcleos (multicore), surge el modelo de programación híbrido en el cual se combinan las estrategias recientemente expuestas. Por ejemplo, en un cluster de multicores, podría pensarse en utilizar el modelo de memoria compartida para los procesadores lógicos dentro de cada procesador y el modelo de pasaje de mensajes para la comunicación entre los procesadores físicos. Este es un ejemplo de modelo de programación híbrida.

El objetivo de utilizar el modelo híbrido es aprovechar y aplicar las potencialidades de cada una de las estrategias que el modelo brinda, de acuerdo a la necesidad de la aplicación. La elección del modelo de programación que se utiliza afecta la decisión del lenguaje de programación y de la librería a utilizar. En las siguientes secciones se describen las librerías de programación paralela más comúnmente utilizadas.

Pthreads

A lo largo del tiempo, los fabricantes de hardware han implementado sus propias versiones para el manejo y administración de hilos. Estas versiones difieren mucho unas de otras, dificultando a los programadores desarrollar aplicaciones multihilos que sean portables. Por este motivo, en el año 1995 la IEEE estableció el standard POSIX (Portable Operating System Interface). La última versión que existe es la IEEE Std 1003.1, 2004 Edition. Pthreads es una librería que implementa el standard POSIX y está compuesto por un conjunto de tipos y llamadas a procedimientos en el lenguaje de programación C que incluye un header file y una librería de hilos que forma parte por ejemplo de la librería libc, entre otras. Se utiliza para la programación de aplicaciones paralelas que utilizan memoria compartida [PTH].

Las subrutinas que conforman la API de Pthreads pueden ser clasificadas en cuatro grandes grupos:

- Manejo de hilos: rutinas que trabajan directamente con los threads -crear, separar, unir, etc. También incluyen funciones para establecer/ consultar atributos de hilo (acoplables, scheduling, etc.).
- Mutex: rutinas que se ocupan de la sincronización y exclusión mutua.
- Mutex (semáforos) proporciona funciones para crear, destruir, bloquear y desbloquear semáforos. Estos se complementan con las funciones de atributos de semáforos que establecen o modifican los atributos asociados con los mismos.
- Variables condición: rutinas para el manejo de la sincronización por condición entre hilos que comparten semáforos. Provee funciones para crear, destruir, wait y signal sobre las variables. También provee funciones para establecer/consultar atributos de las mismas.
- Sincronización: rutinas que manejan locks y barreras.

OpenMP

OpenMP es una interface de programación (API) definida por un conjunto de fabricantes de hardware y software entre los que se encuentran: Sun Microsystems, IBM, Intel, AMD, entre otros. Provee una interface de programación portable y escalable para desarrolladores de aplicaciones paralelas que utilizan memoria compartida.

Esta API soporta C/C++ y Fortran en múltiples arquitecturas, incluyendo LINUX y Windows NT. Provee varios constructores y directivas para especificar regiones paralelas, trabajo compartido, sincronización y variables de entorno [OPE]. Está constituida por los siguientes tres componentes:

- Directivas de compilación
- Biblioteca de rutinas en tiempo de ejecución
- Variables de entorno

Una de las ventajas que presenta OpenMP por sobre Pthreads es que los hilos se encuentran creados durante toda la ejecución del programa; al encontrar una zona paralela los hilos se activan y comienzan a ejecutarse hasta el fin de la zona paralela pero al estar ya creados, no hay consumo ni de recursos ni de tiempo de ejecución para crear y destruir los mismos.

En Pthreads los hilos se crean y se destruyen cada vez que se utilizan. Sin embargo, la ventaja que posee Pthreads por sobre OpenMP es que el programador tiene mayor control sobre la creación, destrucción y comportamiento del hilo ya que el manejo de los mismos es a más bajo nivel que en OpenMP.

MPI

La librería MPI (*Message Passing Interface*) define un estándar para el pasaje de mensajes que puede ser utilizado desde los lenguajes C o Fortran, y potencialmente desde otros también. MPI define tanto la sintaxis como la semántica de un conjunto básico de rutinas, las cuales resultan útiles a la hora de escribir programas con mensajes. En la actualidad, existen diversas implementaciones para diferentes proveedores de hardware [OHI96].

Para poder utilizar las rutinas provistas por MPI es necesario que todos los procesos invoquen la operación MPI_Init. La misma inicializa el ambiente MPI. La rutina MPI_Finalize debe ser llamada al finalizar la computación, ya que se encarga de realizar diferentes tareas de mantenimiento para poder cerrar el ambiente MPI.

MPI ofrece diferentes rutinas que implementan las operaciones básicas send y receive. Para comunicación bloqueante, ofrece una primitiva de recepción y cuatro primitivas de emisión que difieren en el modo de transmisión.

Al igual que para la comunicación bloqueante, MPI provee primitivas de recepción y emisión para comunicación no bloqueante. El término “no bloqueante” en MPI implica que la rutina regresa inmediatamente y que sólo ha iniciado la operación de transferencia del mensaje, no necesariamente la ha completado. La aplicación no podrá reutilizar el buffer de manera segura luego de que la rutina no bloqueante regrese. Las primitivas son MPI_Irecv y MPI_Isend. Además, para poder chequear la finalización de las operaciones send y receive no bloqueantes, MPI provee dos funciones: MPI_Test y MPI_Wait. La primera chequea si una operación ha finalizado o no, y la segunda permite bloquear al proceso hasta que una operación no bloqueante realmente finalice.

MPI provee un conjunto de rutinas que permiten las comunicaciones colectivas. La importancia de emplearlas, siempre que sea posible, radica en que las mismas reducen el overhead de los programas paralelos.

Evaluación de performance en aplicaciones paralelas

Al desarrollar una solución paralela, debe evaluarse su comportamiento. La diversidad de opciones en cuanto a arquitecturas torna complejo el análisis de performance de los sistemas paralelos, ya que los ejes sobre los cuales puede realizarse la evaluación son varios. Se mencionan a continuación algunas de las métricas más utilizadas.

Tiempo de ejecución

Reducir el tiempo de ejecución de aplicaciones con grandes requerimientos de procesamiento es el objetivo principal del procesamiento paralelo. Es decir, disminuir los tiempos de ejecución con respecto a los tiempos secuenciales. El tiempo de ejecución secuencial de un algoritmo es el tiempo transcurrido entre el inicio y el fin de su ejecución en una computadora con una única unidad de procesamiento.

Sin embargo, el tiempo de ejecución de un algoritmo paralelo (Tp) es el tiempo transcurrido desde el momento en que comienza a ejecutarse el algoritmo paralelo hasta que el último procesador termina su ejecución. Para un sistema paralelo dado, Tp normalmente es una función del tamaño del problema (W) y el número de procesadores (p) y suele escribirse como Tp(W,p) [GRA05].

Tamaño del problema

Se define el tamaño del problema (W) como una medida del número total de operaciones básicas necesarias para resolverlo. Dado que puede haber varios algoritmos distintos para resolver el mismo problema, para mantener único el tamaño se lo define como el número de operaciones básicas requeridas por el algoritmo secuencial conocido más rápido en un solo procesador [GRA03][LEO01][AND00].

Speedup

Una de las mediciones de performance más usadas en el dominio paralelo intenta describir cuánto más rápido ejecuta la aplicación sobre una máquina paralela. En otras palabras, cuál es el beneficio derivado del uso de paralelismo.

El speedup se define como el cociente entre el tiempo requerido por la solución secuencial utilizando una única unidad de procesamiento y el tiempo requerido por la solución paralela de interés utilizando más de una unidad de procesamiento.

$$S = Ts/Tp$$

Al hacer referencia a la solución secuencial se debe tener en cuenta que puede existir más de un algoritmo que resuelva el problema. Se debe elegir el algoritmo secuencial que soluciona el problema en la menor cantidad de tiempo. De otra manera no es justa la comparación con el algoritmo paralelo [GRA05].

Los resultados que se obtienen al calcular el speedup pueden ser de tres tipos:

Speedup lineal: también llamado speedup perfecto debido a que el algoritmo paralelo se ejecuta p veces más rápido que el algoritmo secuencial.

Speedup sublineal: es el caso más frecuente en el cual el speedup obtenido es menor que p debido a factores como comunicación, sincronización, trabajo extra impuesto por la paralelización, entre otros.

Speedup superlineal: es el caso menos frecuente, en el cual el speedup obtenido supera a p. Esto puede darse cuando por ejemplo el volumen de datos de un programa es tan grande que no entra en la cache de un único procesador, entonces cuando ese volumen de datos es dividido para ser procesado por más de un procesador, cada división entra en la cache del procesador correspondiente.

Otros casos en donde suele encontrarse la superlinealidad es en algoritmos de exploración en estructuras de búsqueda como árboles y grafos. De esta manera, los tiempos de ejecución pueden presentar una notable mejora que se traduce en una mejora superlineal.

Overhead paralelo

El overhead paralelo total To es la suma de los overheads en que incurren todos los procesadores debido al procesamiento paralelo. Incluye los costos de comunicación, trabajo no esencial y tiempo ocioso debido a la sincronización y componentes seriales del algoritmo:

$$To = pTp – Ts$$

Asumiendo que To es una cantidad no negativa, el speedup está acotado por p. Para un sistema paralelo dado To normalmente es una función de W y p, por lo que suele denotarse To(W,p) [DEG08].

Eficiencia

La eficiencia (E) es una medida de performance paralela estrechamente relacionada con el speedup, ya que está dada por el cociente entre este (S) y el número de procesadores (p):

$$E = S/p = Ts / pTp = 1/(1+To/Ts)$$

Puede pensarse en la eficiencia como el speedup promedio por procesador, y es una medida que refleja cuan bien utilizadas son las unidades de procesamiento para resolver el problema. En la teoría, se puede obtener una eficiencia igual a 1 si el speedup obtenido con p unidades de procesamiento es igual a p. En la práctica, debido a los costos que implican la sincronización y la comunicación, el speedup suele ser menor a p. Esto lleva a la eficiencia a ser menor a 1 [GRA05][DEG08].

Costo

El costo de un sistema paralelo se define como el producto del tiempo de ejecución paralelo (Tp) y el número de procesadores utilizados (p). Refleja la suma del tiempo que cada procesador utiliza resolviendo el problema.

Se dice que el sistema paralelo es de costo óptimo si y sólo si el costo es asintóticamente del mismo orden de magnitud que el tiempo de ejecución serial, es decir pTp = O(W). Por lo tanto, el costo de resolver un problema en una máquina paralela es proporcional al tiempo de ejecución del algoritmo secuencial conocido más rápido en un solo procesador. En el caso de que se cuente con una arquitectura heterogénea, se debe elegir el procesador de mayor performance para la ejecución del algoritmo secuencial.

Dado que la eficiencia es el cociente entre el costo secuencial y el costo paralelo, un sistema paralelo de costo óptimo tiene una eficiencia de O(1) [GRA05][LEO01][AND00].

Grado de concurrencia

El grado de concurrencia o grado de paralelismo C(W) es el número máximo de tareas que pueden ser ejecutadas simultáneamente en cualquier momento en el algoritmo paralelo. Para un W dado, el algoritmo paralelo no puede usar más de C(W) procesadores. C(W) depende sólo del algoritmo paralelo, y es independiente de la arquitectura. Es una función discreta de tiempo, y refleja cómo el paralelismo de software matchea con el de hardware [GRA05][LEO01][AND00].

Así definido, el grado de concurrencia supone un número ilimitado de procesadores y otros recursos necesarios disponibles, aunque esto no siempre puede ser alcanzable en una computadora real con recursos limitados.

La Ley de Amdahl

Gene Amdahl enunció una ley, que lleva su nombre, en la cual establece que hay una máxima mejora en el rendimiento de un algoritmo que depende de la fracción secuencial del mismo (aquella parte no paralelizable). Esto significa que cuando se alcanza el límite superior de mejora, aunque se agreguen más procesadores, el speedup no aumentará sino que por el contrario se mantendrá constante (y eventualmente decrecerá). Si la parte secuencial de un algoritmo es Sec, existe un límite máximo para el speedup alcanzable que está limitado por: T(1)/Sec. Es decir, la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente [AMD07].

Evolución de la Ley de Amdahl: Ley de Gustafson-Barsis

Durante un tiempo, la ley de Amdahl trazó una mirada pesimista sobre el procesamiento paralelo. La ley de Gustafson-Barsis, también conocida como corrección de Gustafson-Barsis, está estrechamente relacionada a la ley de Amdahl (a la cual corrige). Establece que si los problemas escalan (crecen) y el tiempo secuencial crece más que la parte secuencial (no paralelizable) entonces hay un margen mayor para el speedup.

Sea n una medida del tamaño del problema. El tiempo de ejecución de un programa en una computadora paralela puede ser descompuesto en:

$$\mathbf{a(n) + b(n) = 1}$$

donde:
- a es la fracción secuencial
- b es la fracción paralela.

En una computadora secuencial, el tiempo relativo será igual a a(n) + pb(n) donde *p* es el número de procesadores para el caso paralelo. El speedup es entonces:

$$\mathbf{(a(n) + pb(n))}$$

Si la función secuencial a(n) disminuye a medida que se incrementa el tamaño del problema n, entonces el speedup alcanzará p cuando se aproxima a infinito. Por lo tanto la ley de Gustafson rescata el procesamiento paralelo que no era favorecido por la ley de Amdahl [GUS88].

Por último, esta ley mantiene dos suposiciones: la primera es que el tiempo de ejecución paralela es constante, mientras que la segunda es que la parte que se ejecuta secuencialmente, fTs, es también constante y no es función de p.

Escalabilidad

Un algoritmo secuencial es evaluado usualmente en términos de su tiempo de ejecución, expresado como una función del tamaño de su entrada. El tiempo de ejecución de un algoritmo paralelo depende no solo del tamaño de su entrada sino también de la arquitectura paralela, el número de procesadores, y características de la máquina tales como: velocidad del procesador, velocidad de los canales de comunicación, topología de interconexión y técnicas de ruteo. Por este motivo no puede evaluarse un algoritmo paralelo aisladamente de la arquitectura paralela sobre la que es implementado. Se denomina sistema paralelo a la combinación de algoritmo y arquitectura paralela sobre la cual corre [GRA05].

Un algoritmo que posee una buena performance para un problema seleccionado sobre un número determinado de procesadores en una máquina dada puede funcionar pobremente si alguno de los parámetros cambia. La escalabilidad de un algoritmo paralelo sobre una arquitectura es una medida de su habilidad para obtener speedup creciente linealmente con respecto al número de procesadores, en consecuencia refleja la capacidad del sistema paralelo para usar efectivamente una cantidad creciente de recursos de procesamiento.

También hay casos en los cuales el concepto de escalabilidad puede ser utilizado para indicar aquellos algoritmos paralelos que ante un incremento del tamaño del problema, aumentan la cantidad de trabajo de forma razonable. Ahora, incrementar el tamaño del problema requiere mayor precisión. Inicialmente, se podría pensar en la cantidad de elementos a procesar como una medida de tamaño. Sin embargo, doblar el tamaño del problema no necesariamente dobla la cantidad de trabajo. Esto depende de la complejidad del algoritmo.

Optimizaciones

La optimización es una técnica que se utiliza para reducir los tiempos de ejecución de los algoritmos. Existen diferentes tipos de optimizaciones, las realizadas por el compilador, que se explicará a continuación y las algorítmicas, entre las que se pueden mencionar el solapamiento de cómputo y comunicación, que se explicarán más adelante [GCC][LIN].

El compilador más frecuentemente utilizado en cómputo paralelo es el gcc, compilador de GNU. El mismo cuenta con diferentes niveles de optimización que se detallan a continuación.

- -O No se realiza ningún tipo de optimización
- -O1 el compilador trata de reducir el tamaño del código que genera y el tiempo de ejecución sin realizar una optimización que aumente el tiempo de compilación notoriamente.

- -O2 busca un equilibro entre el tamaño del código y el tiempo de ejecución del mismo, manteniendo un compromiso entre los mismos, aumentando el tamaño pero también la velocidad.
- -Os es similar al nivel -O2 habilitando todas las optimizaciones que no incrementan el tamaño del código. Pone énfasis en el tamaño del código por sobre el tiempo
- -O3 realiza optimizaciones poniendo énfasis en el tiempo de ejecución del código por sobre el tamaño. Es el nivel de optimización más alto.

La Tabla 1.1 resume el tipo de operaciones que se realizan en cada nivel de optimización.

	Included in Level			
Optimization	-O1	-O2	-Os	-O3
defer-pop	●	●	●	●
thread-jumps	●	●	●	●
branch-probabilities	●	●	●	●
cprop-registers	●	●	●	●
guess-branch-probability	●	●	●	●
omit-frame-pointer	●	●	●	●
align-loops	○	●	○	●
align-jumps	○	●	○	●
align-labels	○	●	○	●
align-functions	○	●	○	●
optimize-sibling-calls	○	●	●	●
cse-follow-jumps	○	●	●	●
cse-skip-blocks	○	●	●	●
gcse	○	●	●	●
expensive-optimizations	○	●	●	●
strength-reduce	○	●	●	●
rerun-cse-after-loop	○	●	●	●
rerun-loop-opt	○	●	●	●
caller-saves	○	●	●	●
force-mem	○	●	●	●
peephole2	○	●	●	●
regmove	○	●	●	●
strict-aliasing	○	●	●	●
delete-null-pointer-checks	○	●	●	●
reorder-blocks	○	●	●	●
schedule-insns	○	●	●	●
schedule-insns2	○	●	●	●
inline-functions	○	○	○	●
rename-registers	○	○	○	●

Tabla 1.1.: Operaciones de optimización

Métodos de mapeo

La asignación manual de tareas a núcleos de procesamiento es una fase muy importante en el diseño de algoritmos paralelos. Su objetivo es minimizar el tiempo de ejecución de la aplicación. Existen dos formas de llevarlo a cabo: scheduling y mapping. En el primer caso, se especifica para cada tarea en qué núcleo y cuándo debe ejecutarse; mientras que en el caso del mapping solo se especifica dónde debe ejecutarse. El análisis que se lleva a cabo en este caso se centra en las técnicas de mapping.

Según el tipo de sistema que se trate, es decir, memoria compartida o distribuida, se utilizan diferentes técnicas para determinar el mapeo explícito de procesos o hilos a procesadores y/o núcleos. A continuación se detallan las técnicas para cada caso.

Memoria compartida (alternativa 1)

En los sistemas de memoria compartida no existe una función específica para asignar una tarea a un determinado núcleo de procesamiento. En lugar de ello, existe una función que permite definir la planificación de cada tarea. La planificación, permite especificar a cuáles de todos los núcleos existentes en un sistema se puede asignar un determinado proceso o hilo. Para poder asignar los mismos a un núcleo particular es necesario establecer que el único procesador/core planificable es al que queremos asignarlo.

Los sistemas operativos basados en Unix proveen una llamada al sistema que permite definir la afinidad explícitamente. Ésta debe ser utilizada en ambientes con memoria compartida ya que puede administrar los núcleos y procesadores de la máquina sobre la cual está corriendo el sistema operativo.

El encabezado de esta función es el siguiente:

sched_setaffinity(pid_t pid, unsigned long cpusetsize, cpu_set_t *mask)

donde:

- pid: es el identificador del proceso al que se le define la afinidad. Si el valor es 0, se asume que es el proceso en ejecución.
- cpusetsize: es la longitud (en bytes) de los datos apuntados por el parámetro mask.
- mask: representa la máscara de afinidad. Consiste en una máscara de bits en la cual cada uno de ellos representa un procesador (lógico) en el sistema. El orden se establece desde el bit menos significativo que corresponde al primer procesador lógico hasta el bit más significativo que corresponde al último procesador lógico del sistema.

Si bit = 0 ese procesador no será planificable

Si bit = 1 ese procesador será planificable

Por defecto cuando un hilo/proceso se crea es planificable a todos los núcleos existentes en el sistema. Para poder realizar el mapping de un proceso o hilo es necesario establecer que el único procesador planificable es al que queremos asignarlo.

Una de las ventajas que presenta esta técnica de planificación es la posibilidad de modificar dinámicamente (durante la ejecución) la planificación de un determinado proceso. Esto es posible debido a que la llamada del sistema se puede incluir en el código de la aplicación.

Una de las desventajas que presenta esta alternativa es que para poder llevar a cabo la planificación, el código fuente de la aplicación debe ser modificado. Por otro lado, al ser una planificación dinámica e incluida dentro del código fuente, consumirá tiempo de cpu, lo que incidirá en el tiempo de ejecución final.

Memoria distribuida (Pasaje de mensajes)(alternativa 2)

A la hora de utilizar librerías de comunicación para sistemas distribuidos existen diferentes alternativas, tales como: PVM y MPI, entre otras. En la actualidad, la librería más utilizada es MPI.

Open MPI es un proyecto de código abierto que implementa el standard de MPI y que como funcionalidad extra al mismo provee directivas para la asignación explícita de procesos a núcleos de procesamiento. Para ello requiere de dos archivos llamados rankfile y hostfile.

En el archivo hostfile se define el número de núcleos disponibles en el sistema y el nombre de la máquina dentro de la red. El formato del archivo es el siguiente:

- hostNameX slots = nro de núcleos
- hostNameY slots = nro de núcleos

En el archivo rankfile se define una entrada por cada proceso de la siguiente manera:

- rank N = hostNameX slot = nro de cpu
- rank M = hostNameY slot = nroSocket:nroNúcleo

Estos dos archivos deben ser pasados como parámetro en el momento de la ejecución de la siguiente manera:

mpirun -np nro_de_procesos_a_crear -hostfile archivo_hostfile –mca rmaps_rank_file_path archivo_rankfile ejecutable parámetros

La ventaja de esta alternativa es que el código fuente queda intacto, no es necesario modificarlo para agregar esta funcionalidad y como consecuencia el mapping no consume tiempo de ejecución. La desventaja es que la asignación es estática, ya que como es pasada como parámetro en el momento de la ejecución, no se puede modificar dinámicamente.

En función de la arquitectura que se analiza en esta Tesina, puede pensarse en utilizar ambas alternativas de mapeo combinándolas entre sí para obtener una mejor performance global del sistema. Esto es aplicable siempre y cuando el algoritmo que se analice utilice un modelo hibrido, combinando memoria compartida con pasaje de mensajes.

Mapeo en cluster de multicore

El mapeo en la arquitectura de cluster de multicore debe tener en cuenta las dos alternativas recién planteadas dada la jerarquía de memoria que posee.

Si se piensa en un algoritmo que esté dividido en dos niveles jerárquicos, el primer nivel será de división a nivel de procesos para que sean mapeados a los procesadores dentro del cluster. El segundo nivel será el de hilos dentro de cada uno de los procesos creados. De esta manera se obtiene un algoritmo híbrido que puede aprovechar las potencialidades de ambas técnicas (memoria compartida y pasaje de mensajes).

El mapeo de los procesos se llevará a cabo utilizando la alternativa 2 que provee la librería de pasajes de mensajes Open MPI, mientras que para la planificación de los hilos se utilizará la alternativa 1.

Capítulo 2: Conceptos de Simulación

La simulación tiene numerosas aplicaciones ocupando una amplia etapa en los proyectos de diversas disciplinas, ya sea porque se necesita llevar a cabo algún tipo de **experimentación** y realizarla directamente sobre el sistema real es muy costoso o imposible; o bien se busca **diseñar** un nuevo sistema, y un modelo del mismo puede ir modificándose fácilmente hasta obtener el comportamiento deseado. En otros casos, se puede utilizar para **predecir** el comportamiento del objeto real bajo ciertos estímulos y hacer una **evaluación** de diferentes estrategias de acción, entre otros.

Además de un amplio abanico de posibilidades de aplicación, la simulación posee múltiples ventajas: puede colaborar en la reducción del tiempo de desarrollo de sistemas, y permitir que las decisiones puedan chequearse artificialmente. Es de aplicación más simple que ciertas técnicas analíticas y puede ser utilizada para analizar un sistema cuyas entradas sean difusas ó en los casos en que no se encuentra otro medio para solucionar el problema. Otra de las ventajas de su aplicación es la posibilidad de que un mismo modelo puede ser reutilizado en diferentes contextos.

Este capítulo está dedicado a dar una introducción a los temas de simulación, razón por la cual, son necesarias algunas definiciones y conceptos.

Inicialmente debemos dar una definición acerca de qué es una simulación. Thomas H. Naylor dijo: "Simulación es una técnica numérica para conducir experimentos en una computadora digital. Estos experimentos comprenden ciertos tipos de relaciones matemáticas y lógicas, las cuales son necesarias para describir el comportamiento y la estructura de sistemas complejos del mundo real a través de largos periodos de tiempo".

La definición anterior es bastante amplia, por lo que a continuación se presenta una definición más formal enunciada por R.E. Shannon: "La simulación es el proceso de diseñar un modelo de un sistema real y llevar a término experiencias con él, con la finalidad de comprender el comportamiento del sistema o evaluar nuevas estrategias (dentro de los límites impuestos por un cierto criterio o un conjunto de ellos) para el funcionamiento del sistema".

En una simulación mediante computadora el sistema que realiza la simulación es un programa [FUJ99]. El mismo representa o emula el comportamiento de otro sistema a lo largo del tiempo.

Si se simula un sistema es necesario saber qué es. Se llama sistema a una parte de una realidad, restringida por un entorno. Está compuesto por entidades que experimentan efectos espacio-tiempo y relaciones mutuas. Estas entidades u objetos se encuentran lógicamente relacionados y atraviesan ciertas actividades, interactuando para cumplir ciertos objetivos.

Los sistemas a simular se suelen dividir en dos categorías: **discretos** y **continuos**. En un sistema discreto la ocurrencia de cada evento es instantánea y fijada en un punto particular de tiempo, por lo tanto, el estado del mismo cambia únicamente en un conjunto discreto de instantes en el tiempo. En contraposición, un sistema continuo es aquel en el que el estado del sistema varía continuamente en el tiempo.

El sistema que es simulado se denomina **sistema físico**. El mismo puede ser una entidad real que se encuentra en funcionamiento, o un sistema hipotético o artificial. Un sistema físico posee alguna noción del estado con el que evoluciona a lo largo del tiempo, por ejemplo, el número de individuos dentro de una población.

Los sistemas poseen ciertas propiedades que los identifican:

1. **Sinergia:** La interrelación de las partes es mayor o menor que la simple suma de las partes.

2. **Entropía:** Indica el grado de desorden del sistema. Se puede reducir la entropía mediante el ingreso de información al sistema.

3. **Equilibrio homeostático:** Equilibrio dinámico.

Los sistemas físicos pueden ser abiertos, cerrados o aislados, según que realicen o no intercambios con su entorno:

1. Un **sistema abierto** es un sistema que recibe flujos (energía y materia) de su entorno. Los sistemas abiertos, por el hecho de recibir energía, pueden realizar el trabajo de mantener sus propias estructuras e incluso incrementar su contenido de información. El hecho de que los seres vivos sean sistemas estables capaces de mantener su estructura a pesar de los cambios del entorno requiere que sean sistemas abiertos.

2. Un **sistema cerrado** sólo intercambia energía con su entorno, en un sistema cerrado el valor de la entropía es máximo compatible con la cantidad de energía que tiene.

3. Un **sistema aislado** no tiene ningún intercambio con el entorno.

Etapas del proceso de simulación

Si se observa la bibliografía de simulación de sistemas, varios autores coinciden en que los pasos necesarios para realizar un estudio de simulación son [COS05]:

Definición del sistema

Para tener una definición exacta del sistema que se desea simular, es necesario hacer primeramente un análisis preliminar de éste, con el fin de determinar la interacción con otros sistemas, las restricciones del sistema, las variables que interactúan dentro del mismo y sus interrelaciones, las medidas de efectividad que se van a utilizar para definir y estudiar el sistema y los resultados que se esperan obtener del estudio.

Formulación del modelo

Una vez definidos con exactitud los resultados que se esperan obtener del estudio, se define y construye el modelo con el cual se obtendrán los resultados deseados. En la formulación del modelo es necesario definir todas las variables que forman parte de él, sus relaciones lógicas y los diagramas de flujo que describan en forma completa al modelo.

Colección de datos

Es posible que la facilidad de obtención de algunos datos o la dificultad de conseguir otros, pueda influenciar el desarrollo y formación del modelo. Por consiguiente, es importante que se definan con claridad y exactitud los datos que el modelo va a requerir para producir los resultados deseados.

Implementación del modelo en el ordenador

Con el modelo definido, el siguiente paso es decidir si se utiliza algún lenguaje de programación general (como C, Java, Fortran, Algol, Lisp, etc.), o se utiliza algún paquete o lenguaje de simulación (como GPSS, Simula, Simscript, etc.), para procesarlo en la computadora y obtener los resultados deseados.

Validación

Es una de las etapas principales de un estudio de simulación. A través de esta etapa es posible detallar deficiencias en la formulación del modelo o en los datos alimentados al modelo. Las formas más comunes de validar un modelo son:

1. La opinión de expertos sobre los resultados de la simulación.
2. La exactitud con la que se predicen datos históricos.
3. La exactitud en la predicción del futuro.
4. La comprobación de falla del modelo de simulación al utilizar datos que hacen fallar al sistema real
5. La aceptación y confianza en el modelo de la persona que hará uso de los resultados que arroje el experimento de simulación.

Experimentación

La experimentación con el modelo se realiza después que éste haya sido validado. La experimentación consiste en generar los datos deseados y en realizar un análisis de sensibilidad de los índices requeridos.

Interpretación

En esta etapa del estudio, se interpretan los resultados que arroja la simulación y con base a esto se toma una decisión. Es obvio que los resultados que se obtienen de un estudio de simulación ayudan a soportar decisiones del tipo semi-estructurado, es decir, la computadora en si no toma la decisión, sino que la información que proporciona ayuda a tomar mejores decisiones y por consiguiente a sistemáticamente obtener mejores resultados.

Verificación

Se verifica que el sistema simulado tenga el grado de precisión y exactitud conforme al sistema físico simulado. No se debe olvidar que los sistemas son dinámicos y con el transcurso del tiempo es necesario modificar el modelo de simulación, ante los nuevos cambios del sistema real, con el fin de llevar a cabo actualizaciones periódicas que permitan que el modelo siga siendo una representación del sistema.

Documentación

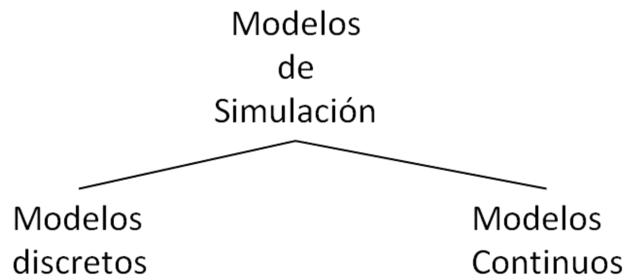
Se elabora la documentación técnica y el manual de usuario del modelo.

Debido a la dinámica de los sistemas, el proceso descripto anteriormente es iterativo, lo cual permite realizar cambios y mejoras ante las variaciones del sistema simulado.

Como se describió anteriormente, para poder llevar a cabo la simulación es necesario un **modelo** que represente el sistema a evaluar. Todos los modelos de simulación son una especificación de un sistema físico (o al menos de algunas de sus componentes) en términos de un conjunto de **estados** y **eventos** [FER95].

Los modelos de simulación suelen clasificarse en estáticos o dinámicos, deterministas o estocásticos, y discretos o continuos. Un modelo es estático cuando trata de representar el estado de un sistema en un punto particular del tiempo, y es dinámico cuando trata de representar la evolución del sistema a lo largo de un determinado intervalo de tiempo. Un modelo es determinista cuando tiene un conjunto conocido de valores de entrada y produce un único conjunto de valores de salida, mientras que es estocástico cuando se usan como entrada una o más variables aleatorias. Entradas aleatorias producen salidas aleatorias, luego estas salidas sólo pueden ser consideradas como una estimación del comportamiento real del modelo.

La clasificación de los modelos en discretos o continuos respeta la definición dada a los sistemas de dichos tipos y utiliza un criterio basado en el mecanismo de *flujo de tiempo* (Figura 2.1.). En los modelos discretos, el cambio del estado del sistema físico se realiza en pasos discretos de tiempo. Mientras que en los modelos continuos, el estado cambia constantemente [FUJ99].



2.1. Clasificación de los paradigmas de simulación

El comportamiento del sistema en las simulaciones continuas se describe mediante un conjunto de ecuaciones diferenciales que realizan los cambios en el estado del sistema en función del tiempo de simulación. En el caso de los modelos discretos, el sistema va cambiando su estado en saltos o etapas, entre un lapso de tiempo y el siguiente.

Es importante destacar que no tiene necesariamente que modelarse un sistema de una clase mediante un modelo de la misma clase: es posible, por ejemplo, usar un modelo discreto para simular un sistema continuo. El tipo de modelo elegido depende tanto de la naturaleza del sistema como de los resultados que se deseen obtener.

El presente trabajo, se enfoca en las simulaciones discretas. En las siguientes secciones se continúa con la descripción de los mismos.

Elementos de un modelo de simulación

La simulación de un sistema físico se logra mediante un conjunto de elementos que su modelo debe proveer, ellos son [FUJ99]:

(1) La representación del estado del sistema físico: para lo cual se definen las variables de estado del sistema, entidades del mismo, etcétera, de acuerdo al nivel de detalle deseado para el modelo;

(2) Algún mecanismo para modelar la evolución del sistema físico: que representa la dinámica del sistema y refleja sus resultados en la representación del estado del mismo.

(3) Alguna representación del tiempo: lograda mediante una abstracción denominada tiempo de simulación (simulation time), que representa el avance del tiempo en el sistema físico.

Tanto (1) como (2), se encuentran resueltos con el mantenimiento y la modificación de las variables de estado. En cambio, para el tiempo de la simulación, existen varias acepciones:

1. Tiempo físico o real (Physical time): hace referencia al tiempo en el sistema real;
2. Tiempo de la simulación (Simulation time): que es la abstracción utilizada por la simulación para modelar el tiempo físico.
3. Tiempo de reloj de pared (Wallclock time): que es el tiempo de ejecución de la simulación. A menudo, dicho valor se obtiene mediante varias lecturas de un reloj de hardware mantenido por el sistema operativo.

Es indispensable reconocer y diferenciar estos conceptos, ya que son uno de los principales puntos de confusión en el área de la simulación paralela y distribuida. Por ejemplo, si se quiere simular la evolución de una población de bacterias con ciertas características a lo largo de una semana, el tiempo físico se extenderá desde el primer día al séptimo, pero dentro de la simulación el transcurso de ese mismo tiempo podría encontrarse representado mediante un contador de días (que contendrá los valores de 0 a 7 durante la ejecución de la simulación). Mientras que, si el programa de simulación se ejecuta en 3 horas y desde las 14 hs. de un día X, el tiempo de ejecución de la simulación o wallclock time se extenderá desde las 14:00 hasta las 17:00 hs.

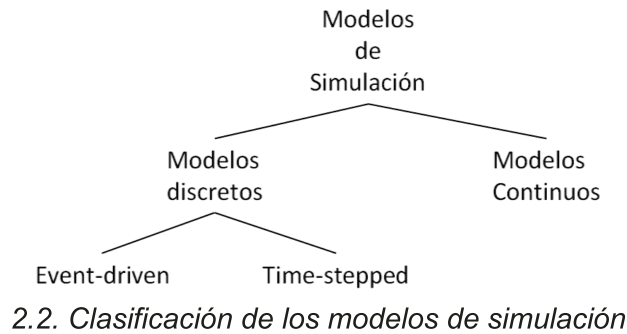
El tiempo físico y el tiempo de ejecución de la simulación (Wallclock), son esencialmente el mismo tiempo que el utilizado en el sentido convencional. En cambio, el tiempo de simulación es un concepto existente únicamente en los mundos simulados.

> El tiempo de simulación se define como un conjunto ordenado de valores donde cada uno de ellos representa un instante de tiempo en el sistema físico. Además, para un par de valores de tiempo de simulación T_1 que representa el tiempo físico F_1, y T_2 representando F_2, si $T_1 < T_2$, entonces F_1 ocurre antes que F_2, y $(T_2 - T_1)$ es igual a $(F_2 - F_1) * K$ siendo K una constante. Si $T_1 < T_2$, luego T_1 ocurre antes que T_2, y si $T_1 > T_2$, T_1 ocurre después de T_2.

La relación lineal que existe entre los intervalos de tiempo real y del tiempo de la simulación asegura que la duración del tiempo de simulación tenga una correspondencia apropiada a la duración en el tiempo físico. Para una simulación dada, se utiliza una misma escala de tiempo de simulación que es respetada por todos los componentes de la simulación. Esto asegura que todas las partes de la simulación tienen un punto de referencia y nociones o conceptos comunes ante las relaciones de anterioridad y posterioridad entre las acciones que ocurren en instantes específicos del tiempo simulado.

Cambios de estado en el sistema

Los programas de simulación discreta definen una colección de variables de estado y reglas para su modificación, a lo largo del tiempo de simulación. Lo cual extiende la clasificación los modelos de simulación con dos categorías de simulaciones discretas: (1) Dirigida por sucesos o *Event-driven*, donde el cambio de estado del sistema se produce ante la ocurrencia de un evento y (2) Dirigido por el tiempo o *Time-driven (también conocida como Time-stepped)*, para la cual el tiempo de simulación avanza en pasos de tiempo constantes (véase Figura 2.2.) [FUJ99].



2.2. Clasificación de los modelos de simulación

El cambio de las variables de estado a lo largo del tiempo puede ser representado mediante un diagrama espacio-tiempo donde el eje x representa el tiempo de la simulación y diferentes strips o rectángulos representan la evolución de las variables en el tiempo. Los cambios de estado están denotados con líneas verticales que cortan los strips en el punto de tiempo donde se producen los mismos (véase Figura 2.3.). La tarea de los programas de simulación es completar el diagrama mediante el cálculo de los valores de cada variable de estado a lo largo de la simulación.

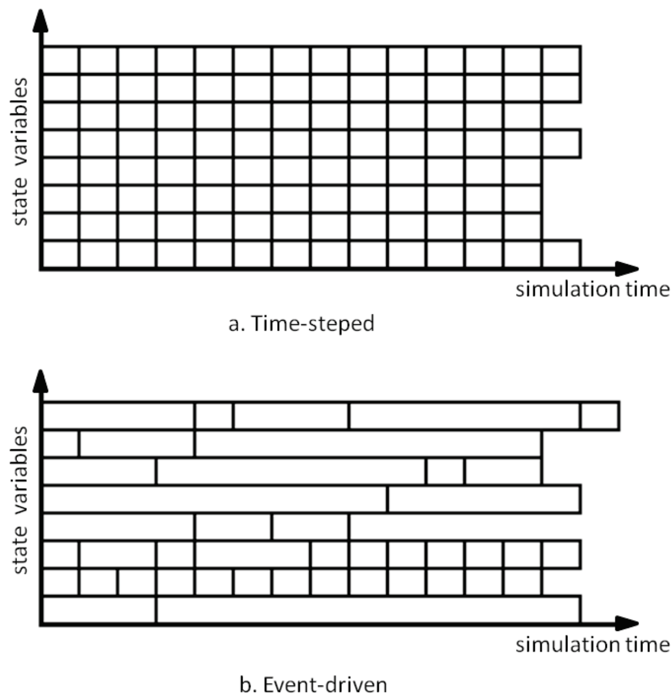


Figura 2.3. Diagramas espacio-tiempo para simulaciones discretas

Ejecución time-stepped

En las simulaciones de este tipo, el tiempo de simulación se subdivide en una secuencia de pasos de tiempo de igual tamaño, y la simulación avanza de un paso al siguiente [FUJ99]. Para llevar a cabo la simulación se completa el diagrama tiempo-espacio mostrado en la Figura 2.3.a. mediante el computo de un nuevo estado, en cada paso de tiempo. En realidad no todas las variables de estado necesitan ser modificadas en cada paso de tiempo, pero el mecanismo de ejecución sólo puede avanzar de un paso de tiempo al siguiente.

Las acciones que se producen en el mismo paso de tiempo se consideran simultáneas, y con frecuencia se supone que no tienen un efecto sobre las demás. Esto es importante porque permite que las acciones que se producen dentro de cada intervalo de tiempo puedan ser ejecutadas simultáneamente en diferentes equipos. En este paradigma, si dos acciones tienen una relación de causalidad que debe ser modelada con precisión en la simulación, dichas acciones deben ser simuladas en intervalos de tiempo diferentes. Por lo cual, el tamaño del paso del tiempo es importante ya que determina la precisión de la simulación con respecto al tiempo.

Ejecución event-driven

En lugar de realizar el cálculo de un nuevo valor para cada variable de estado, la ejecución por eventos actualiza solo los valores de las variables cuando “algo interesante” ocurre [FUJ99]. Este “algo interesante” que ocurre se denomina **evento**. Un evento es una abstracción utilizada dentro de la simulación para modelar una acción instantánea en el sistema físico. Cada evento tiene un time-stamp[1] asociado con el cual indica el punto de tiempo en el cual se dio la ocurrencia. El evento mencionado causa la modificación de una o más variables de estado definidas por la simulación. Si se observa la Figura 2.3.b., las actualizaciones de las diferentes variables se dan de manera irregular.

Otra característica a destacar es que el tiempo en este tipo de simulación avanza actualizándose con los time-stamps de los diferentes eventos, haciendo que los intervalos de tiempo no sean regulares. Si se quisiera emular una simulación time-stepped con una simulación event-driven, sería necesario que los eventos ocurran en intervalos de tiempo regulares y el efecto de cada uno de ellos sobre el sistema sea la actualización de los valores de todas las variables de estado del mismo.

---

[1] Time-stamp: marca de tiempo que permite saber el instante en el que fue generado el evento en cuestión.

Tipos de soluciones

Una vez que se conocen los diferentes elementos de una simulación, se pueden describir las alternativas posibles para llevarla a cabo.

Programas de simulación discreta

El primer caso a considerar dentro de esta clase de programas es el secuencial. En él se utilizan comúnmente tres estructuras de datos:

1. Las variables estado, que describen el estado del sistema;
2. Una lista de eventos que contiene los eventos que ocurrirán en algún momento futuro respecto al punto donde se encuentra la simulación actualmente;
3. Una variable de reloj global, que denota el instante de tiempo en el cual la simulación se encuentra.

Si la variable reloj contiene un valor T, todas las actividades del sistema físico con time-stamp anterior a T deben haber sido ejecutadas, todas las de igual time-stamp se deben encontrar en ejecución y las que tienen un time-stamp superior se encuentran en espera por la ejecución. Todos los eventos que se encuentran en la lista de eventos deben tener time-stamp mayor o igual a T.

Operacionalmente, un evento es representado mediante una estructura de datos que contiene el time-stamp del evento, alguna marca del tipo de evento y detalles del evento necesarios para la simulación.

Otro aspecto relevante es el mecanismo generador de eventos, o **planificador** que crea los eventos que sucederán dentro de la simulación y para los cuales creará las estructuras necesarias, completará los campos de time-stamp, tipo de evento e información antes mencionados; y lo agregará a la lista de eventos.

El programa secuencial puede ser dividido en dos componentes. La parte inferior es el simulation executive, que mantiene el reloj y la lista de eventos (es la parte del programa que no depende del sistema físico). La parte superior del programa que incluye las variables de estado y el software que modela el sistema físico se denomina simulation application y se encuentra íntimamente ligada al sistema físico. En el más simple de los casos, el simulation executive necesita proveer solo un proceso para la planificación de eventos a la simulation application.

El programa ejecutado por el executive sigue la siguiente estructura:

Mientras (la simulación está en proceso) {

Quitar el evento de menor time-stamp de la lista de eventos;

Actualizar el reloj de la simulación al time-stamp de dicho evento;

Ejecutar el manejador de eventos en la simulation application para procesar el evento;

}

El procedimiento que procesa el evento puede hacer, a su vez, las siguientes acciones:

1. Modificar las variables de estado para modelar los cambios producidos al sistema físico por la ocurrencia del evento;
2. Planificar nuevos eventos futuros dentro de la simulación.

Existen dos puntos importantes a destacar en la simulación de eventos discreta, y que se deben tener en cuenta para la ejecución en máquinas paralelas o distribuidas. Ambas están relacionadas con la característica de asegurar que la simulación representa fielmente las relaciones causales del sistema real.

1. La simulation application puede sólo planificar eventos dentro del futuro simulado, o sea dentro del límite de tiempo de la simulación;
2. El simulation executive siempre procesa el evento de menor time-stamp contenido en la lista de eventos.

Estas dos propiedades aseguran la ejecución en orden de los eventos y que para el reloj de simulación nunca se realice un decremento en su valor. Esto es importante porque asegura que cualquier operación que se realice sobre el estado del sistema no afectará el comportamiento de un evento anterior en el tiempo.

Comienzo y fin de la simulación

Quedan dos aspectos pendientes de la ejecución de simulaciones que necesitan ser descriptos: el comienzo y el fin de la simulación.

La simulación comienza con la inicialización de las variables de estado (realizada mediante técnicas de programación tradicionales) y la generación de los eventos iniciales (que pueden ser creados definiendo un evento de inicialización con time-stamp menor que el de cualquier evento generado y permitiendo que la simulation application provea un procedimiento que realice la planificación de todos los eventos iniciales requeridos por la simulación).

Existen varias técnicas para detectar el fin de la simulación. Puede crearse un evento "Detener simulación" definido para ser el último evento procesado por la simulación, incluso cuando existen eventos pendientes en la lista. Alternativamente, un evento "Fin de tiempo de la simulación" puede ser definido para indicar que se ha terminado cuando el valor del reloj excedió el tiempo de finalización de la simulación, sin dejar eventos pendientes en la lista.

Capítulo 3: Simulaciones paralelas y distribuidas

Aunque la simulación es ampliamente utilizada, presenta una serie de problemas. En particular, este trabajo se centra en el problema que poseen los modelos usados para estudiar sistemas de gran escala, ya que suelen ser muy complejos y además necesitan utilizar muchos recursos de cómputo.

Una simulación paralela y distribuida se encuentra compuesta, comúnmente, por un conjunto de simulaciones secuenciales, cada una de las cuales modelan una parte diferente del sistema físico y se ejecutan en diferentes procesadores. En terminología de simulación de eventos discretos paralela y distribuida, se denomina a cada simulación secuencial como un proceso lógico (logical process o LP), y la actividad que realizan es la ejecución, procesamiento y generación de nuevos eventos.

Puede ocurrir que un evento que es generado dentro de un LP es relevante para uno o más LPs. Cuando esto sucede, un mensaje es enviado a los demás LPs notificando la ocurrencia del evento. Visto de otra forma, las interacciones entre procesos físicos son simuladas en los ambientes distribuidos mediante el envío de mensajes entre los procesos lógicos correspondientes [FUJ99].

El rol que cumple cada LP dentro de la simulación está dado por el nivel de paralelismo definido en el diseño de la solución [MIS86]. Los niveles posibles son:

1. **Nivel Aplicación.** Una solución desarrollada en este nivel contiene réplicas independientes del mismo modelo de simulación y realiza la ejecución de los mismos con diferentes parámetros de entrada.
2. **Nivel Subrutinas.** En este caso, se realiza la distribución de copias de las rutinas de un programa.
3. **Nivel Componente.** Aquí, se realiza la utilización del paralelismo disponible en el sistema físico modelado, debido a que se modelan las componentes del sistema real.
4. **Nivel Event.** En este nivel, se manejan listas de eventos que son planificados y ejecutados a lo largo de la simulación. Existen dos posibilidades para el manejo de la lista de eventos.
    - **Lista de Eventos Centralizada**. Se posee una única lista de eventos que es distribuida a los procesadores esclavos.
    - **Lista de Eventos Descentralizada**. Permite la simulación concurrente con diferentes tiempos de cómputo en los diferentes procesadores.

Cada uno de estos niveles de diseño posee sus ventajas y desventajas. Por ejemplo, el nivel de aplicación otorga una alta eficiencia en la mayoría de los casos, permite que el código de la simulación sea reutilizable y la escalabilidad del problema ilimitada. Pero, por otro lado, impide la paralelización del programa. A su vez, el nivel de subrutinas acelera la generación de eventos o el procesamiento de datos, pero tiene como contraparte el hecho de que la generación de subrutinas es limitada y por consecuencia la obtención de Speedup también lo es [6]. Por estas razones, es necesario analizar cuál de ellos será utilizado en la resolución del problema, para evitar mayores inconvenientes.

Enfoques de simulaciones

Existen diferentes enfoques para el modelado de los problemas [FUJ99]. No se profundizará demasiado en ellos, debido a que el modelado no es el eje central de la presente tesina, pero es necesaria una introducción a los mismos para tener una noción básica.

En el enfoque **orientado a eventos**, el modelo se centra en los eventos y en cómo se ve afectado el estado de la simulación por la ocurrencia de los mismos. Los programas que utilizan este enfoque, se componen de procedimientos manejadores de eventos correspondientes con cada tipo de evento que puede llegar a ocurrir dentro de la simulación.

Otro enfoque es el **orientado a procesos** (process-oriented) que ataca los problemas mediante una abstracción denominada proceso de simulación (simulation-process). El proceso de simulación está destinado a modelar una entidad específica del modelo con un comportamiento bien definido. La descripción del comportamiento de la entidad es encapsulada por el proceso y describe las acciones a llevar a cabo por el proceso durante su tiempo de vida.

Otro elemento importante dentro de la simulación orientada a procesos es el concepto de **recurso**. Los recursos son la abstracción que representa una entidad compartida y por la cual los procesos compiten. Dichos recursos se adquieren y liberan mediante primitivas definidas dentro de una librería o propias del lenguaje de simulación. Conceptualmente, una simulación orientada a procesos puede ser vista como una colección de procesos autónomos, que interactúan con los demás procesos y compiten por los diferentes recursos compartidos.

Por otro lado, otras representaciones muy utilizadas son la **basada en objetos** (object-based) y la **orientada a objetos** (object-oriented), que utilizan colecciones de objetos para el modelado. Un objeto consiste en un conjunto de campos (variables de estado o atributos) y un conjunto de métodos para modelar el comportamiento del componente. Los objetos de la simulación son creados dinámicamente durante la ejecución del programa y pueden invocar métodos de otros objetos. La invocación de un método puede ser vista como el envío de un mensaje a un objeto requiriendo la ejecución de un método.

Cuando el sistema requiere que la simulación sea estructurada como colecciones de objetos en interacción, se está ante el caso de un sistema basado en objetos. Además, si el sistema provee mecanismos de herencia para caracterizar las relaciones entre colecciones de objetos similares, se tiene un sistema orientado a objetos.

Para completar esta clasificación, existe un último enfoque denominado **escaneo de objetos** (object scanning), el cual es una variación del mecanismo time-stepped. En él, el programa consta de una colección de procedimientos, con un predicado asociado a cada uno de ellos. En cada paso de la simulación, dicho predicado es evaluado y el procedimiento asociado se ejecuta si el resultado de la evaluación es TRUE (verdadero). Este proceso se repite hasta que no se encuentren evaluaciones de predicados en TRUE. Cuando esto último sucede, la simulación avanza al siguiente paso de tiempo.

Autómatas celulares

Otro hito fundamental, y que puede ser visto como un enfoque de simulaciones particular, son los autómatas celulares. Son modelos matemáticos que simulan sistemas dinámicos que evolucionan en pasos discretos de tiempo. Un autómata celular consiste en una rejilla o cuadriculado, donde cada celda de la cuadrícula se conoce como "célula". Cada una posee, en cada momento, un estado seleccionado de un número finito de estados posibles. Además, cada célula tiene una "vecindad", un conjunto finito de células en las cercanías de la misma. De esta forma, se aplica a todas las células (de forma homogénea, y en cada paso discreto de tiempo) una función de transición que, en base a los valores de la célula en cuestión y los valores de sus vecinos, devuelve el nuevo estado que tendrá dicha cuadrícula en la siguiente etapa de tiempo [WIL05[MIS86]].

Un Autómata Celular genera comportamientos complejos a partir de reglas muy sencillas. Es posible lograr sencillos modelos digitales que representen con suma fidelidad algunas leyes de la física, por ejemplo. Son útiles en la construcción de modelos donde los componentes (actores) son de similar naturaleza y comportamiento y se rigen por reglas parecidas. No existen herramientas basadas en Autómatas Celulares generales para cualquier problema, si se quiere un buen resultado hay que implementar desde cero. Sin embargo los algoritmos requeridos son muy simples.

Algoritmos conservativos

Como se mencionó anteriormente, el sistema físico es visto como un conjunto de procesos físicos que interactúan de alguna manera. Cada proceso físico es modelado por un proceso lógico (LP), y las interacciones entre los mismos se modelan mediante el intercambio de mensajes con marca de tiempo entre los procesos lógicos correspondientes. El procesamiento realizado por cada LP es el de una secuencia de eventos, donde cada uno de ellos puede modificar las variables de estado y/o programar nuevos eventos.

A primera vista este paradigma parece ser ideal para la ejecución paralela o distribuida, ya que simplemente se puede asignar diferentes procesos lógicos a diferentes procesadores y que cada LP ejecute en orden, evento por evento, e intercambie mensajes en los casos en que sea necesario.

Por desgracia, esto no es así. Cada proceso lógico debe procesar la totalidad de sus eventos, tanto los generados a nivel local como los generados por los demás LPs, en orden (los de menor time-stamp primero y los mayores luego).

Al procesar los eventos en orden podría suceder que el procesamiento de un evento afecte a otro evento en su pasado, provocando claramente una situación errónea. Si bien la marca de tiempo ordena el procesamiento de eventos fácilmente en una computadora secuencial, mediante el uso de una lista centralizada de eventos pendientes o una variable de tiempo global, esta situación no es tan fácil de realizar cuando la ejecución se distribuye en más de un procesador. Los errores resultantes del procesamiento fuera de orden de los eventos se denominan **errores de causalidad**, y el problema de asegurar que los eventos se procesan en un orden por marca de tiempo se conoce como el **problema de sincronización** [FUJ99].

Cuando la simulación se distribuye en múltiples procesadores, es necesario un mecanismo para la ejecución concurrente que permita reproducir exactamente los mismos resultados que la ejecución secuencial. El objetivo del algoritmo de sincronización es asegurar que esto ocurra. Es importante reconocer que el algoritmo de sincronización no es necesario para garantizar que los acontecimientos en diferentes procesadores se procesan en orden, sino sólo que el resultado final es el mismo que en el algoritmo secuencial.

Para evitar situaciones como ésta, se aplica la restricción de que no puede haber ninguna variable de estado que se comparta entre los procesos lógicos. El estado del simulador completo debe ser dividido en vectores de estado, con un vector de estado por LP. Cada proceso lógico contiene una porción del estado correspondiente al proceso físico que modela, así como un reloj local que denota hasta qué punto el proceso ha progresado en tiempo de simulación.

Aunque la exclusión del estado compartido en el paradigma de procesos lógicos evita muchos tipos de errores de causalidad, no impide que otros sucedan. Por ejemplo, tenga en cuenta dos eventos, en un proceso lógico E_{10} con indicación de la hora 10, y E_{20} en otro proceso lógico con indicación de la hora 20 (ver Figura 3.1.). Si el evento E_{10} genera un nuevo evento E_{15} en el proceso lógico 1 que contiene indicación de la hora 15, entonces E_{15} podría afectar E_{20}, lo que exige la ejecución secuencial de los tres eventos. Para evitar errores de este tipo se impone la siguiente restricción de causalidad local en cada proceso lógico:

Una simulación de eventos discretos, que consta de los procesos lógicos (LPs) que interactúan exclusivamente a través del intercambio de mensajes con marca de tiempo obedece a la restricción de causalidad local si y sólo si cada uno de los LPs procesa los eventos en orden según su time-stamp [FUJ99].

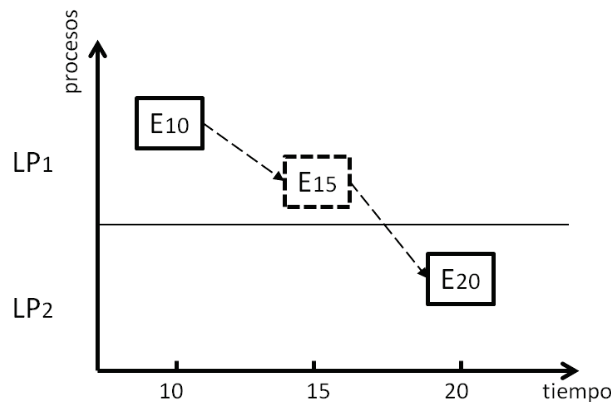


Figura 3.1. Relaciones de causalidad [FUJ99]

Operacionalmente, se debe decidir si E_{10} no se puede ejecutar al mismo tiempo que E_{20}, pero ¿cómo hace el simulador para determinar la relación entre los eventos sin haber ejecutado E_{10}? Este es el gran problema que debe abordarse. El escenario en el cual E_{10} afecta a E_{20} puede ser una secuencia de eventos compleja, que depende principalmente de los time-stamps de los eventos.

Si se asume que una simulación consiste de N procesos lógicos, LP_0,…, LP_{N-1}, y $Clock_i$ es el tiempo de simulación actual para un LP dado: cuando un evento es procesado, el reloj del proceso avanzará automáticamente al time-stamp de dicho evento. Si LP_i envía un mensaje a LP_j durante la simulación, se dice que existe un link o vinculo entre los LPs mencionados.

Existen varias alternativas para la solución del problema de sincronización. Cada una de ellas se ven reflejadas en los protocolos de sincronización conservativos, donde cada LP respeta estrictamente el orden de procesamiento de los eventos por su time-stamp. Entre ellos, se encuentran el envío de mensajes nulos para prevenir el bloqueo de procesos dentro de la simulación, o las diferentes alternativas para la detección de dichos bloqueos y la recuperación del sistema ante los mismos [MIS86].

Algoritmos optimistas

Como contraparte de los algoritmos conservativos, los algoritmos optimistas permiten la violación de la restricción de causalidad local, proveyendo un mecanismo de recuperación ante las mismas. El término **ejecución optimista** refiere a la forma en que los procesos lógicos procesan los eventos, asumiendo que no existen errores de causalidad [FUJ99].

El problema central que enfrenta un proceso lógico es que se puede tener uno o más mensajes que ha recibido de otros procesos, pero no se puede estar seguro de que es correcto procesar estos eventos, ya que hacerlo más tarde puede resultar en la violación de la restricción de causalidad local. Los protocolos optimistas de sincronización procesan los eventos a pesar de que su seguridad no puede ser garantizada, y proporcionan un mecanismo para "deshacer" estos cálculos en caso de detectarse un error de causalidad.

Time Warp es el protocolo de sincronización optimista más conocido. Aunque la ejecución optimista permite un avance más veloz dentro del tiempo de la simulación (ya que se procesan eventos y en caso de falla se realiza **rollback**[1] de las operaciones), también tiene sus desventajas. Un sistema Time Warp “puro” tiene ciertas deficiencias en cuanto a utilización de memoria que pueden disminuir la performance de algunas aplicaciones, por lo que se han creado otros mecanismos que resuelven dichos problemas.

Sincronización conservativa vs. Sincronización optimista

No existe un tipo de sincronización que sea “el mejor” para todos los tipos de aplicaciones. Las diferencias importantes entre estos enfoques se resumen en la Tabla 3.1. El simulation executive en protocolos conservadores generalmente es menos complejo que en la sincronización optimista. Si la simulación tiene un gran **lookahead**[2], el mecanismo de sincronización no tendrá que ser invocado con mucha frecuencia. Esto se traducirá en menores gastos de ejecución que los ejecutivos optimistas ya que estos últimos deben crear y recopilar información para el historial.

Guardar el estado es quizás el más costoso entre los overheads optimistas. Pero también las tareas comunes tales como la asignación dinámica de memoria, errores de ejecución, y E/S que se puede implementar utilizando métodos convencionales de programación en los sistemas conservadores exigen la inclusión de mecanismos especiales en los sistemas de optimistas. Todas las soluciones para estos problemas, suman complejidad al executive de la simulación.

[1] **rollback**: operación que deshace los cambios realizados en las variables de estado de la simulación, desde un punto especifico en el tiempo.
[2] **lookahead**: si un LP en un instante de tiempo dado T, puede planificar solo nuevos eventos con un time-stamp de al menos T+L, entonces L es el lookahead para ese proceso en ese momento.

Una de las áreas donde los sistemas conservadores incurren en gastos adicionales que no se producen en los sistemas de optimistas es cuando la topología de los procesos lógicos varía durante la ejecución. Si el protocolo de simulación utiliza la información de la topología, tales como la distancia entre procesos, se requiere un mecanismo adicional. Pero, en general, los sistemas optimistas son más complejos que los conservadores.

Los protocolos conservadores no pueden aprovechar plenamente el paralelismo en la aplicación debido a que deben protegerse contra el peor de los casos, que rara vez puede que ocurra. Un corolario de esta observación es que no existe un protocolo conservador que escale a menos que se hagan ciertas suposiciones sobre el lookahead, excepto en circunstancias relativamente especializadas, tales como un gran número de eventos que contienen exactamente el mismo time-stamp. Por "escalar" se entiende que si el número de procesadores aumenta en proporción, la simulación paralela/distribuida es capaz de mantener aproximadamente la misma tasa de avance en el tiempo de simulación por segundo de tiempo de wallclock. Por el contrario, el paralelismo en los protocolos de optimistas no está limitado por dependencias posibles entre los procesos lógicos, sino más bien por las dependencias reales representados por interacciones de los eventos. Así, los enfoques optimistas ofrecen un mayor potencial de escalabilidad en la ausencia de lookahead.

La explotación del lookahead puede llevar a código de simulación que es complejo, lo que conduce a un software que es difícil de desarrollar y mantener. Restricciones en la dinámica cambiando la topología de los procesos lógicos puede agravar aún más este problema. Si su meta es desarrollar un simulation executive de "propósito general" de rendimiento sólido a través de una amplia gama de modelos y no exigir que el promotor del modelo que esté familiarizado con los detalles del mecanismo de sincronización, sincronización optimista ofrece una mayor esperanza.

Por otro lado, si se tienen sistemas legados de simulación secuencial y se quiere adaptarlos para procesamiento paralelo o distribuido, la sincronización conservadora ofrece la ruta del mínimo esfuerzo. Esto se debe a que no hay necesidad de agregar los mecanismos necesarios de un sistema optimista. En el largo plazo, uno puede ser capaz de automatizar muchas de las tareas para el procesamiento optimista, pero las herramientas que automaticen este proceso aún no existen.

Autómatas celulares y el mecanismo de sincronización

En el caso de los autómatas celulares que fueron mencionados anteriormente y que serán el caso particular de estudio para los algoritmos descriptos en los próximos capítulos, si se utiliza un generador de números random que permita conseguir los mismos valores que en el algoritmo secuencial, se puede lograr que el resultado final sea exactamente el mismo que para el algoritmo secuencial, mediante un esquema de sincronización que se asemeja a un protocolo conservativo. Esto se debe a que el algoritmo secuencial realiza el barrido de las celdas en un orden predefinido y solo la variabilidad de las celdas y acciones

a realizar influyen en el cambio de estado. Se dice que el mecanismo utilizado se asemeja a un esquema conservativo porque como consecuencia de las dependencias propias de la aplicación (cada celda necesita de sus vecinas para ser procesada), el estado de la simulación no puede ser dividido en variables locales a los diferentes LPs. Primero se deben procesar todas las celdas para un instante de tiempo dado, luego se realiza otro barrido de la matriz para el paso siguiente, etc.

		Protocolo	
		Conservativo	Optimista
Característica	Overheads	- El simulation executive es más simple. - Puede necesitar un mecanismo especial para las topologías dinámicas de procesos lógicos. - Overheads bajos si se tiene un buen lookahead.	- El simulation executive es más complejo, ya que requiere guardar estados e historia. - Necesita mecanismos especiales para la alocación dinámica de memoria, E/S, errores en tiempo de ejecución.
	Paralelismo	- Limitado por el escenario del “peor caso”. Requiere un buen lookahead para la obtención de performance y escalabilidad.	- Limitado por las dependencias que puedan existir entre eventos.
	Desarrollo	- Potencialmente complejo	- Más robusto. - Menos dependiente del lookahead. - Mayor transparencia del mecanismo de sincronización.
	Sistemas heredados	- Inclusión directa	- Requiere mecanismos adicionales para soportar el rollback.

Tabla 3.1. Comparación Conservativo vs. Optimista [FUJ99]

Capítulo 4: Descripción del problema y su solución secuencial

En el presente capítulo se detalla la descripción del problema de aplicación analizado. El problema seleccionado es Sharks and Fishes. El mismo consiste en la simulación de la evolución de una población de tiburones y peces dentro del océano, siguiendo diferentes reglas. Este problema es representativo de una clase de sistemas biológicos que permite estudiar el comportamiento de diferentes poblaciones evolucionando en el tiempo.

En este problema, el océano se encuentra dividido en una grilla de tamaño NxN, donde el lado derecho de la misma se encuentra conectado con el izquierdo, y a su vez el lado superior con el inferior (es un espacio toroidal). Las diferentes celdas que constituyen la grilla se encuentran vacías u ocupadas por un pez, un tiburón o por plancton. La población evoluciona con pasos de tiempo discretos de acuerdo a ciertas reglas.

Reglas para la población de peces

En cada paso de tiempo, el pez intenta moverse a una celda vecina que se encuentre con plancton (para alimentarse), en caso de hallarlo, pasa a ocupar esa celda y su energía aumenta según un parámetro de la simulación. Si sus celdas vecinas se encuentran ocupadas (por otros peces o tiburones), se queda en la posición anterior.

Un pez no puede reproducirse si no puede moverse. Si un pez alcanza la edad de reproducción ante un nuevo paso de tiempo, se reproduce dejando una cría con edad 0 en la celda anterior del movimiento y con la mitad de la energía que tenía el pez inicial. Los peces mueren cuando ya no tienen energía suficiente (la energía se reduce con el paso del tiempo).

Reglas para la población de tiburones

En cada paso, si en alguna de las celdas vecinas se encuentra un pez, el tiburón se mueve para poder comerlo, pasando a ocupar la celda donde se encontraba el pez y ganado una cantidad determinada de energía.

Si no hay ningún pez a su alrededor, pero se encuentra alguna celda vacía (agua), el tiburón se mueve hacia ella. En caso contrario, no se mueve.

Los tiburones únicamente comen peces.

Cuando ya no tiene energía suficiente, el tiburón muere (la energía se reduce al pasar el tiempo).

Características del océano o escenario

Además de las reglas establecidas para los individuos, otro detalle que se deberá tener en cuenta es el tiempo de regeneración de plancton dentro de una celda, una vez que la misma es desocupada por un pez o tiburón.

Finalmente, deberán tenerse en cuenta como parámetros para la solución:

1. Las dimensiones de la grilla (NxN);
2. La edad de reproducción (tanto para los peces como para los tiburones);
3. Las cantidades de energía adquirida por cada especie, la proporción para la división de energía al reproducirse y la pérdida de energía por inanición;
4. El tiempo que lleva la regeneración de plancton.

Solución Secuencial

Para la resolución del problema en forma tradicional o secuencial, se utilizó un autómata celular. Se representó al océano mediante una matriz de estructuras o registros que mantiene el estado de cada posición de la misma.

Los pasos que sigue el algoritmo son bastante simples. Primero, se inicializa la matriz, con las cantidades de tiburones y peces necesarias y de acuerdo a las constantes definidas para la simulación. Luego, se realiza el recorrido de la matriz por cada paso de tiempo de la simulación, invocando a una función que analiza y elige aleatoriamente al comportamiento a seguir por el pez o tiburón, dependiendo del estado de las celdas adyacentes y de la acción que se desee realizar. Dicha función verifica las ocho celdas que rodean a la posición actual (ver Figura 4.1.), y de todos los movimientos posibles para la acción a realizar, elige uno aleatoriamente.

i-1,j-1	i-1,j	i-1,j+1
i,j-1	i,j	i,j+1
i+1,j-1	i+1,j	i+1,j+1

Figura 4.1.: Celdas para el procesamiento

Observaciones

Cuando se realiza la elección de manera aleatoria dentro de los movimientos posibles para el individuo, la función utilizada es la función rand() que proveen las librerías de C.

Una de las dificultades que se encontró en la implementación de este algoritmo fue cómo detectar que una celda ya fue modificada dentro del mismo paso de reloj de la simulación (para no realizar más de un paso de tiempo dentro del mismo recorrido). La solución a este problema establece un campo adicional dentro de las estructuras posicionales que lleva cuenta del último intervalo de tiempo en el que fue modificada la celda.

Si bien esta solución es simple, en ciertas situaciones es necesario tener en cuenta este aspecto en el comportamiento del algoritmo para no interpretarlos como falla. Por ejemplo, si un pez se mueve desde una celda de posición $p_{i,j}$ de la matriz hacia la posición $p_{i+1,j}$ (con $0 <= i < (N-1)$, siendo N la altura de la matriz), un tiburón que se encuentra en la posición $p_{i+1,j-1}$ no podrá considerar como opción comer el pez que anteriormente se ha movido.

Luego de una primera aproximación de la implementación, se buscó optimizar el funcionamiento, para lo cual se realizaron las siguientes modificaciones:

1. Se modificó el modo de almacenamiento de la matriz, posibilitando el acceso por filas.

2. Se agregaron condiciones de corte para el recorrido, en caso de que el océano quede deshabitado (sin peces o tiburones).

En esta implementación se puede apreciar que las componentes de un programa de simulación se encuentran definidas. En primer lugar, las variables de estado del sistema se ven reflejadas en las diferentes celdas de la matriz que representa al océano. Luego, la variable de reloj, representada por t, avanza un paso por cada barrido de la matriz. Y por último, la lista de eventos, al tratarse de un recorrido secuencial, está dada por las iteraciones propias del recorrido y la modificación de las celdas adyacentes por el comportamiento del ocupante de la celda actual.

Pruebas realizadas

La arquitectura sobre la cual se trabajó en este caso es un servidor Blade de 16 hojas, de las cuales se utilizó una para la ejecución de las pruebas del algoritmo secuencial, ya que solo es necesario un core. Cada una de las hojas posee 2 procesadores Quad Core Intel Xeón e5405 de 2.0 GHz, con 2 Gb de memoria RAM (compartido entre ambos procesadores) y cache L2 de 2 x 6Mb entre cada par de cores.

Al igual que para todos los algoritmos presentados en esta Tesina, se utilizó la opción de compilación de gcc -O3, con el objetivo de mejorar el rendimiento de todas las soluciones.

Para establecer un criterio de prueba, con diferentes tamaños de problema y así hacer visible los resultados, se ejecutaron simulaciones alterando los valores de pasos de tiempo de las mismas y del tamaño de matriz. Por otro lado, la cantidad de individuos con las cuales se ejecutaron las pruebas también se mantuvo de forma proporcional al tamaño de la matriz. En lo que refiere a las variables relacionadas con las modificaciones poblacionales (energía ganada o perdida, edad de reproducción, etc.), se mantuvieron los mismos valores para peces y tiburones, lo cual permite establecer un equilibrio de la población existente. En el Capítulo 6 se observan los resultados obtenidos.

Soluciones de Memoria Compartida

En esta sección, se plantean las soluciones para la implementación del modelo de memoria compartida. Se detallan dos soluciones en diferentes herramientas (Pthreads y OpenMP) y se establece una comparación en cuanto a la performance obtenida.

Solución Pthreads

Para el algoritmo en la herramienta Pthreads, se realizó una división de la matriz en strips horizontales (o grupos de filas) de igual tamaño para cada thread. Se realizó la paralelización de los recorridos de los diferentes fragmentos de matriz.

Dentro de cada una de estas fracciones se distinguen tres zonas (ver Figura 5.1.). Por un lado, la parte superior, constituida por los N+1 elementos que se “comparten” con el thread anterior al actual. Esto significa que si el identificador del thread actual es i, el thread con el cual se comparte dicha zona es el hilo i-1 ó, si la cantidad de threads es w y se trata del primer thread (al cual corresponde el identificador 0), se compartirán los datos con el thread identificado con el valor w-1. Luego, se tiene una parte central en la cual se encuentran las celdas o elementos que son propias del thread. Estas serán únicamente modificadas por el hilo al cual fueron asignadas. Por último, la parte inferior del fragmento contiene, al igual que la parte superior, celdas compartidas, pero con el thread siguiente al actual (el hilo con identificador i+1 ó 0 si se trata del thread w-1).

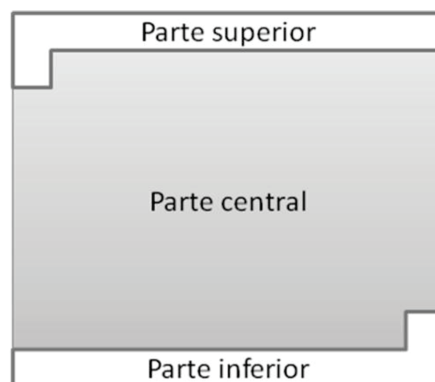


Figura 5.1. Descripción de las partes del strip o submatriz

Por lo tanto, se tendrán tantos fragmentos como threads se creen y se irá alternando el trabajo de los mismos en las diferentes zonas. Para un paso de tiempo dado de la simulación se producirán tres instancias de procesamiento (ver Figura 5.2.). Inicialmente, cada thread trabaja con la parte superior de su zona, reflejando, en caso de ser necesario,

modificaciones en la parte inferior del hilo anterior (a.). Más tarde, en una segunda fase, se procesa la parte central de cada submatriz por el thread correspondiente (b.). Finalmente, se procesa la parte inferior teniendo la posibilidad de alterar la parte superior del thread siguiente (c.).

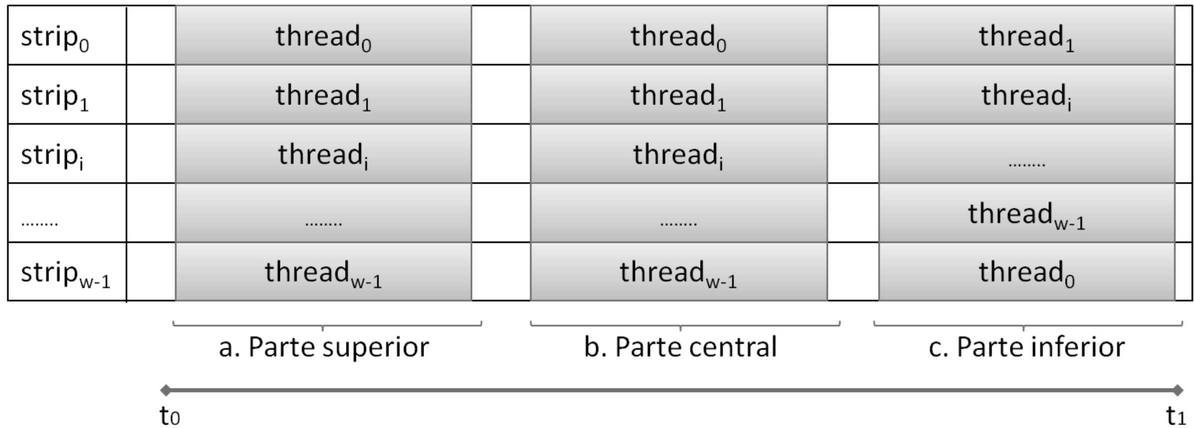


Figura 5.2. Fases de procesamiento en un paso de simulación

Este mecanismo se repite por cada paso de tiempo de la simulación permitiendo la ejecución a lo largo de toda la matriz. De manera que, en cada fragmento de la grilla y en un mismo instante de tiempo, un único thread puede estar realizando modificaciones.

Observaciones

Para la implementación de la presente solución, fue necesario reemplazar la función rand() utilizada en el algoritmo secuencial por rand_r(). Esto se debe a que la función rand() es **no reentrante**. Las funciones reentrantes son aquellas que pueden ser llamadas de forma segura cuando otra instancia de la misma se encuentra en ejecución. Si varios hilos comienzan la ejecución de la misma función en un mismo punto, una función no reentrante se comportaría de manera incorrecta.

En el algoritmo implementado, se advirtió que la función rand "secuencializa" sus llamadas, debido a que para el cálculo de cada valor que se genera es necesario el valor anterior. Por esta razón, no era visible el paralelismo de la aplicación. Cambiando la función rand por rand_r se puede lograr independizar los llamados y hacer que el algoritmo funcione correctamente.

Por otro lado, se utilizaron variables condición para proteger el acceso a las áreas limítrofes compartidas por los threads. De modo que, cuando un hilo necesita acceder a una submatriz, se verifica que le corresponda el “turno” para realizar el procesamiento. En caso de que no le sea otorgado el acceso se bloquea a la espera de la liberación del área compartida. Este mecanismo permite evitar las escrituras simultáneas en una misma celda de la grilla.

Para obtener un mejor rendimiento, se utilizaron las optimizaciones provistas por el compilador, haciendo uso de la opción de compilación -O3 de gcc.

Pruebas realizadas

En este caso, se utilizó una única hoja de la arquitectura descripta en el capitulo anterior. Las pruebas se realizaron sobre los mismos casos que en el algoritmo secuencial, con diferentes valores para los tamaños de matriz y la cantidad de pasos de tiempo. Y se plantearon los mismos casos de simulación con mapeo manual de hilos a cores. En el Capítulo 6 se detallan los resultados obtenidos.

Solución OpenMP

En este caso, se decidió particionar los datos, al igual que en la solución Pthreads, en grupos de filas o strips que se verán modificados por los diferentes threads generados. Mediante la utilización de múltiples directivas omp_for se buscó subdividir el recorrido de la matriz compartida y realizar la sincronización correspondiente para evitar que varios hilos modifiquen al mismo tiempo un fragmento determinado de la matriz.

De manera similar al algoritmo Pthreads, cada hilo realiza 3 fases de recorrido. Primero, se recorre el área compartida con el hilo anterior. Luego, el fragmento de submatriz central, que es independiente a los demás threads. Y, por último, se realiza el procesamiento de las últimas filas correspondientes al thread, las cuales son compartidas con el hilo siguiente.

Observaciones

Al ser OpenMP una herramienta que provee directivas, el uso de la directiva omp_for permite realizar la paralelización de la aplicación de manera simple. Por cada paso de tiempo de la simulación, se reproducen tantos recorridos como fases se describieron (3 recorridos, con sus límites correspondientes), y se les asignan tantas iteraciones a cada uno de los bucles for afectados como threads se crearon. Por ejemplo, en el caso de un único thread se generan 3 bucles for que recorren el inicio de la matriz, su parte central y las filas finales tantas veces como pasos de tiempo se realicen.

Cuando se realiza el procesamiento de las fracciones superiores e inferiores (haciendo referencia a los términos utilizados en la solución Pthreads), es necesario establecer barreras entre los hilos, para que en un mismo instante de tiempo solo un thread se encuentre en dicha zona. Esta sincronización se encuentra contemplada en las barreras implícitas de las directivas omp_for. La barrera implícita de omp_for permite que cada thread, al terminar con el procesamiento del bloque de instrucciones contenidos en el mismo, espere a que los demás threads finalicen la ejecución del mismo bloque de instrucciones contenido en la directiva.

Para el caso en que no es necesario esperar a que todos los threads terminen con el bloque, se agrega la clausula nowait al omp_for correspondiente. Este último es el caso del procesamiento del bloque central de cada submatriz, ya que no es necesario que todos los threads hayan finalizado con el procesamiento del bloque de celdas central para continuar con el bloque inferior.

Para obtener un mejor rendimiento, se utilizaron las optimizaciones provistas por el compilador, haciendo uso de la opción de compilación -O3 de gcc.

Pruebas realizadas

Al momento de ejecutar las pruebas se planteó la solución con mapeo manual de hilos a cores. Además, al igual que con la solución Pthreads, las pruebas se realizaron para los mismos casos que en el algoritmo secuencial, con diferentes valores para los tamaños de matriz y la cantidad de pasos de tiempo. También se utilizó una única hoja de la arquitectura, teniendo como máximo 8 cores para el procesamiento (que es el limite máximo de cores con memoria compartida de la arquitectura descripta). En el Capítulo 6 se detallan los resultados obtenidos.

Solución de Pasaje de Mensajes

Esta sección introduce una alternativa de solución que plantea el uso de múltiples procesadores (que pueden encontrarse en diferentes nodos o máquinas conectadas mediante una red de comunicación). Esta solución fue desarrollada bajo la suposición de que trabajar con la matriz por fragmentos sería una forma de disminuir el tiempo de ejecución de la simulación.

Inicialmente, al particionar la matriz, se encuentran varios ítems por definir:

1. Cómo se dividirá la matriz: optando por strips (ya sean horizontales o verticales) o por bloques;
2. Cómo se asignarán dichos fragmentos a los procesos.

En este caso, la solución implementada divide la matriz en **strips** horizontales, teniendo en cuenta el modo de almacenamiento de la misma. Se aprovecha el espacio de almacenamiento horizontal contiguo para no tener demoras en el armado de estructuras de datos para el envío.

Como consecuencia de este tipo de división, pueden suceder **conflictos** en los límites de los fragmentos, ya que se tendrían dos procesos trabajando en celdas contiguas y el funcionamiento original del algoritmo en una celda en particular necesita interactuar con las ocho celdas adyacentes.

En cuanto a la asignación de fragmentos a procesos, se determinó que corresponder uno a cada proceso sería una buena alternativa, ya que si se opera con un único fragmento a lo largo de toda la simulación no se necesita intercambiar gran cantidad de mensajes para la actualización de los límites.

Como se mencionaba anteriormente, existe la posibilidad de que sucedan conflictos en los límites de los segmentos de matriz entre procesos. Para su solución existen dos alternativas:

1. Utilizar un mecanismo que realice el recorrido de las celdas limítrofes y posteriormente verifique los valores modificados para hacer **rollback** de las operaciones conflictivas y resolver el paso de tiempo de algún modo.

2. Utilizar otro mecanismo que sincronice la información para evitar los conflictos.

En la solución desarrollada, se utilizó el segundo método, realizando el envío de la información a través de pasaje de mensajes (librería MPI). La operatoria del algoritmo se lleva a cabo mediante el uso de un esquema de comunicación circular de procesos que comunica mensajes en forma bidireccional. Según la cantidad de procesadores disponibles, se divide la matriz de manera uniforme entre los procesos que ocuparán cada uno de ellos.

Se tienen 2 tipos de procesos, el primero o proceso 0 reparte la matriz inicial, ejecuta la simulación en su fragmento de matriz y recopila el resultado final (matriz obtenida al final del algoritmo). Por otro lado, los procesos worker reciben su fracción de la matriz y mediante mensajes se comunican con los demás workers y también con el proceso 0, enviando los fragmentos necesarios para operar en los límites de las porciones. En la Figura 5.3. se puede observar cómo se lleva a cabo dicha comunicación. Por un lado, el proceso 0 envía la matriz inicial y recibe el resultado final; y por otro lado, cada proceso se comunica con su predecesor y sucesor para solucionar los límites.

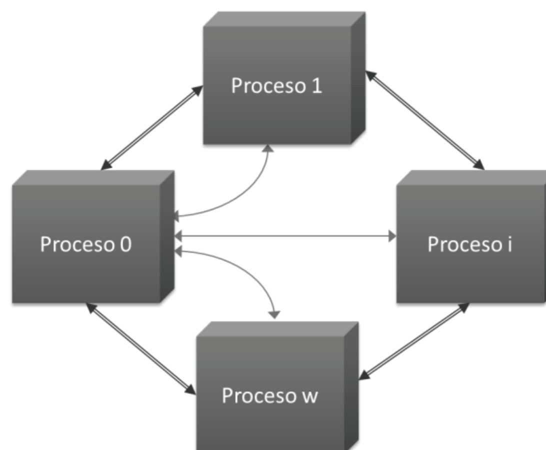


Figura 5.3.: Modo de comunicación

En una primera etapa, los procesos envían su fracción superior (ver Figura 5.4.) al proceso siguiente, para que éste trabaje con ella utilizando el mismo mecanismo de la solución secuencial. Posteriormente, una vez que fue procesado el fragmento enviado, el proceso que inicialmente lo recibió, lo envía a su antecesor y se continúa con el procesamiento hasta que sea necesaria la parte inferior del actual proceso, la cual a continuación se recibe, y así, se van completando los diferentes pasos de tiempo.

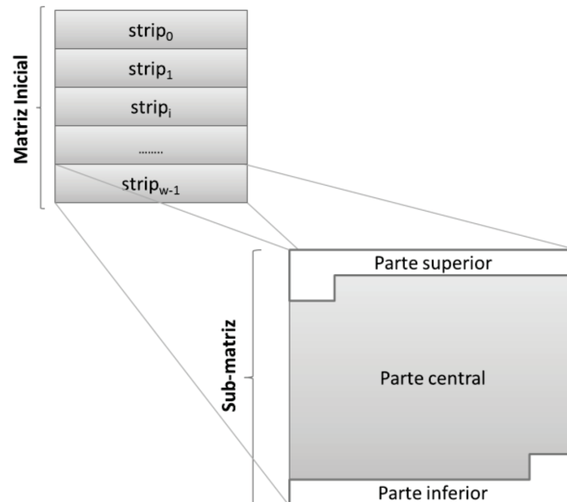


Figura 5.2.: Modo de división

La parte inferior de la sub-matriz de un proceso se encuentra compuesta por los primeros n+1 elementos del final de la fracción correspondiente a dicho proceso, en conjunto con una copia de los n+1 elementos del límite con el proceso sucesor (para cualquier proceso i, el sucesor será el proceso de identificador [(i+1) módulo w]). La parte superior de la sub-matriz, se encuentra conformada por los n+1 elementos del límite con el proceso predecesor (cuyo índice será (i+w-1) modulo w]) y por las n+1 celdas del inicio del fragmento del proceso i. Para una mejor apreciación, en la Figura 5.5. se observa la matriz dispuesta de a N+1 celdas por fila.

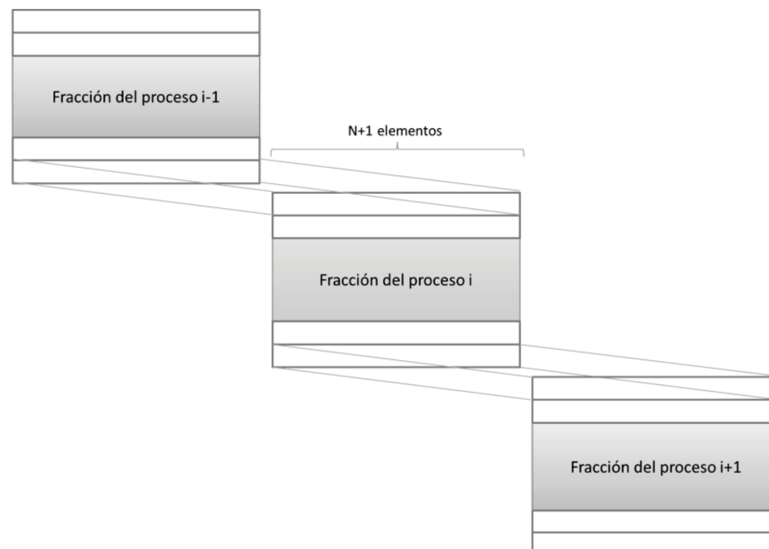


Figura 5.5.: Detalle de procesos

Observaciones

La mejora obtenida con este algoritmo está dada por la posibilidad de ejecutar varios recorridos de las sub-matrices, y principalmente, solapar el procesamiento de la parte central de las mismas.

Para obtener un mejor rendimiento, se utilizaron las optimizaciones provistas por el compilador, haciendo uso de la opción de compilación -O3 de gcc.

Pruebas realizadas

Contemplando los mismos casos que en el algoritmo secuencial, se realizaron las pruebas con diferentes tamaños de matriz y de pasos de tiempo. En cuanto a la arquitectura utilizada, se realizo la ejecución en cuatro hojas, totalizando 32 cores. En los próximos capítulos se detallan los resultados obtenidos y su comparación con las demás soluciones.

Soluciones Híbridas

Con el propósito de lograr un mejor aprovechamiento de la arquitectura y mejorar las prestaciones, se proponen las siguientes soluciones que combinan pasaje de mensajes con memoria compartida.

Solución MPI+Pthreads

Para su implementación, se evaluó el modo de distribución de los datos y la comunicación necesaria entre los procesos e hilos. Se optó por realizar un particionamiento de los datos al igual que en el algoritmo MPI, por lo que a cada proceso le corresponde un bloque de filas de la matriz original. Además, para que los hilos dentro de cada uno de los procesos puedan cooperar para completar la simulación, se repitió el esquema de división de datos de la solución Pthreads correspondiendo a cada hilo una submatriz, sólo que en este caso se le asignará un fragmento correspondiente a la porción del proceso.

Cuando se trabaja con algoritmos híbridos, la comunicación se lleva a cabo entre los hilos de los diferentes nodos de la arquitectura. Como se observa en la Figura 5.6., se tiene un grupo de procesos, donde cada uno de ellos se comunica con su antecesor y sucesor, y donde el proceso 0 comunica los datos iniciales mediante sentencias de comunicación colectivas, procesa y recibe el resultado final. La comunicación entre procesos se realiza para el intercambio de las filas limítrofes correspondientes, tal como sucede con el algoritmo MPI.

Dentro de cada proceso existe un conjunto de threads, los cuales resuelven los pasos de tiempo de la simulación para la submatriz del proceso que integran (ver Figura 5.6.). Debido al modelo híbrido de comunicación, son necesarios dos tipos de threads para esta solución.

Por un lado, se encuentran los **“threads de frontera”** que permiten la comunicación con los procesos adyacentes al proceso al cual pertenecen. Entre ellos se encuentran los hilos iniciales o de identificador 0 (cero), que se encargan de sincronizar el cómputo con los demás hilos de su mismo proceso pero, a su vez, se comunican mediante pasaje de mensajes con el último hilo del proceso anterior (en el caso del proceso 0, aquel con el cual se establece la comunicación es el proceso p-1). También se encuentran en la frontera los hilos finales o de identificador th-1, que interactúan con el hilo inicial del proceso siguiente para realizar el envío y recepción de los límites de la matriz (en el caso del proceso p-1, se interactuará con el proceso 0) y, además, alternan el procesamiento de su área con los demás threads del mismo proceso.

Como segundo tipo, surgen los **“threads intermedios”**, que únicamente interactúan con los hilos de su mismo proceso (en la Figura 5.6., los threads 1 y j). De la misma manera que en la solución Pthreads llevan a cabo la simulación alternando el procesamiento en las zonas correspondientes.

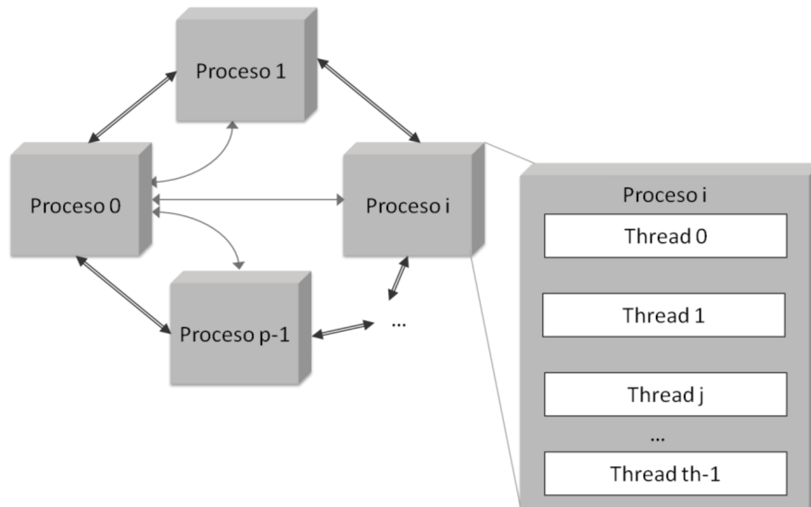


Figura 5.6. Esquema de procesos e hilos de ejecución

En una primera etapa, el proceso 0 realiza la distribución de los datos iniciales y los demás procesos reciben dichos datos. Luego, cada proceso crea e inicializa las estructuras necesarias para los hilos o threads que llevarán a cabo la simulación. En otra nueva fase, los threads trabajan para completar los pasos de tiempo comunicándose y sincronizando unos con otros. Y por último, se destruyen los hilos generados y se recopilan los resultados de la simulación.

Observaciones

Como herramientas para la programación del algoritmo se utilizaron Pthreads en lo que concierne a operaciones entre threads de un mismo proceso, y OpenMPI en la comunicación y sincronización entre hilos de diferentes procesos. Al utilizar un entorno MPI, es necesario tener habilitado el soporte para la ejecución de múltiples hilos en los procesos [INS10].

Al igual que en las soluciones de memoria compartida, fue necesario reemplazar la función rand() por rand_r() que permite la ejecución de varios llamados en un mismo instante de tiempo, por varios threads de un mismo proceso. Además, fueron aprovechadas las ventajas del mapeo de hilos a cores dentro de la implementación y de las comunicaciones colectivas donde fuera posible su aplicación (distribución y recopilación de datos).

Por último, es necesario aclarar que para lograr una mejor performance de la solución se utilizaron las optimizaciones provistas por gcc, compilando los archivos fuentes con la opción -O3.

Pruebas realizadas

Se realizaron las pruebas correspondientes para luego poder realizar una comparación entre los algoritmos híbridos y el de pasaje de mensajes. En el capitulo siguiente se detallan los resultados obtenidos.

Solución MPI+OpenMP

Al igual que en la solución híbrida explicada anteriormente, se utiliza el particionamiento y distribución de datos del algoritmo MPI puro. El master distribuye los datos entre todos los procesos que forman parte de la ejecución (incluyendo una parte para sí mismo). Una vez que se dispone de la porción inicial de océano necesario, cada proceso llama a la directiva *omp_parallel* para generar el conjunto de hilos solicitados (siendo el mismo proceso el hilo 0), que resolverá el problema.

Dentro del bloque paralelo, en cada caso de la simulación, se trabaja de manera similar que en la solución con OpenMP. La porción de océano de la que debe encargarse el proceso se resuelve por los distintos hilos trabajando sobre secciones diferentes de la porción.

Para esto, se ejecutan los *omp_for* para calcular y sincronizar la parte superior y central de la sección correspondiente a cada hilo. Cuando esto se ha realizado, cada proceso MPI debe enviar su parte superior al proceso anterior, y recibir su parte inferior del proceso siguiente, para lo cual se utilizan dos *omp_single* que permite ejecutar ambas acciones en simultáneo por dos hilos diferentes (utilizando la clausula *nowait*). Una vez realizada la comunicación, los hilos ejecutan la parte inferior de la sección correspondiente a cada uno por medio del tercer *omp_for* de la solución OpenMP pura.

Cuando se ha resuelto la parte inferior de la porción del océano correspondiente al proceso MPI, entonces la envía al proceso siguiente, y recibe del anterior la parte superior de su porción (de la misma manera que la comunicación antes mencionada) para comenzar a resolver un nuevo paso de la simulación.

Una vez que se han realizado todos los pasos de la simulación, el proceso master recolecta las porciones de océano que resolvió cada uno de los procesos (incluyendo su parte) por medio de una comunicación colectiva, y de esta manera se queda con el estado actual del océano completo.

Observaciones

Se debe recordar que cuando se menciona que un proceso i se debe comunicar con el siguiente, se refiere al proceso *(i+1) mod p* (siendo p la cantidad de procesos MPI), y en caso de tener que comunicarse con el anterior, es el proceso *(i-1+p) mod p*.

Como herramientas para la programación del algoritmo se utilizaron OpenMP en lo que concierne a operaciones en Memoria Compartida, y OpenMPI en la comunicación y sincronización entre hilos de diferentes procesos. Al utilizar un entorno MPI, es necesario tener habilitado el soporte para la ejecución de múltiples hilos en los procesos [INS10].

Al igual que en las soluciones de memoria compartida, fue necesario remplazar la función rand() por rand_r() que permite la ejecución de varios llamados en un mismo instante de tiempo, por varios threads de un mismo proceso. Además, fueron aprovechadas las ventajas del mapeo de hilos a cores dentro de la implementación y de las comunicaciones colectivas donde fuera posible su aplicación (distribución y recopilación de datos).

Por último, es necesario aclarar que para lograr una mejor performance de la solución se utilizaron las optimizaciones provistas por gcc, compilando los archivos fuentes con la opción -O3.

Pruebas realizadas

Se realizaron las mismas pruebas que la versión hibrida antes mencionada, para luego poder realizar una comparación entre ellas y la de Pasaje de Mensajes. En el capitulo siguiente se detallan los resultados obtenidos.

Capítulo 6: Resultados obtenidos

Descripción de la Plataforma y Pruebas Realizadas

En esta tesina se utilizó el lenguaje C con las librerías OpenMPI, Pthreads y OpenMP, la primera de ellas para pasaje de mensajes mientras que las dos restantes para memoria compartida.

Para realizar la experimentación se ha utilizado una arquitectura Blade de 8 hojas, con 2 procesadores quad core Intel Xeón e5405 de 2.0 GHz en cada una de ellas. Cada hoja posee 2Gb de memoria RAM (compartido entre ambos procesadores) y cache L2 de 2 x 6Mb entre par de núcleos.

Además, para analizar el comportamiento de cada una de las soluciones paralelas implementadas en este trabajo, se mide el tiempo de ejecución, speedup, y eficiencia de las mismas. También se analiza la escalabilidad desde tres puntos de vista: el primero variando el tamaño del océano (1024x1024; 2048x2048; 4096x4096; 8192x8192), el segundo modificando el lapso de tiempo de la simulación (1024; 2048; 4096; 8192 momentos), y el último utilizando diferentes cantidades de procesos y/o hilos.

Las combinaciones mencionadas anteriormente se agrupan en 16 escenarios de prueba los cuales se encuentran detallados en la Tabla 6.1.

Escenario	Momento	Tamaño del Océano
1	1024	1024x1024
2	1024	2048x2048
3	1024	4096x4096
4	1024	8192x8192
5	2048	1024x1024
6	2048	2048x2048
7	2048	4096x4096
8	2048	8192x8192

Escenario	Momento	Tamaño del Océano
9	4096	1024x1024
10	4096	2048x2048
11	4096	4096x4096
12	4096	8192x8192
13	8192	1024x1024
14	8192	2048x2048
15	8192	4096x4096
16	8192	8192x8192

Tabla 6.1. Detalle de los escenarios de prueba

Resultados Obtenidos

Solución Secuencial

El algoritmo secuencial se ejecutó en un único núcleo de la arquitectura descripta anteriormente. Los resultados del mismo se muestran en la Tabla 6.2.

Escenario	TS
1	80,24
2	306,48
3	1109,01
4	4243,48
5	165,16
6	645,56
7	2448,55
8	9211,07
9	334,07
10	1322,86
11	5169,97
12	20003,03
13	673,44
14	2680,60
15	10618,03
16	41805,37

Tabla 6.2. Resultados del algoritmo secuencial

Soluciones paralelas con Memoria Compartida

Como se indicó, para las pruebas de memoria compartida se desarrollaron dos soluciones: una de ellas utilizaba OpenMP mientras que la otra utilizaba Pthreads para así poder comparar los resultados obtenidos.

Ambas pruebas utilizan la arquitectura Blade mencionada anteriormente.

En la Tabla 6.3. se muestran los tiempos de ejecución de las primera solución (con OpenMP), donde para cada escenario fue probado con 2, 4, 6 y 8 hilos respectivamente. Además en el Anexo I se encuentra detallada la información respecto al speedup y a la eficiencia.

Escenario	2	4	6	8
1	40,30	20,39	13,56	10,29
2	153,85	78,98	53,43	40,25
3	589,05	302,02	202,70	155,14
4	2215,78	1128,10	758,79	575,44
5	82,89	41,59	27,80	20,84
6	323,65	163,52	109,98	82,85
7	1254,24	640,45	427,81	325,08
8	4705,26	2408,96	1640,13	1235,83
9	167,48	83,94	56,17	42,13
10	663,52	332,74	223,26	168,08
11	2604,97	1317,73	880,76	665,17
12	10078,94	5114,22	3446,07	2592,32
13	337,58	168,69	112,98	84,62
14	1345,88	671,88	449,56	338,60
15	5324,91	2671,85	1787,15	1345,02
16	20962,01	10528,40	7058,79	5306,90

Tabla 6.3. Tiempos de la solución paralela de memoria compartida con OpenMP.

Para analizar el comportamiento del algoritmo al escalar tanto el problema como la arquitectura, se muestra en forma gráfica la eficiencia lograda por las pruebas realizadas. En la Figura 6.1. se grafican las eficiencias logradas fijando el tamaño de océano en 4096x4096 y la cantidad de hilos en 8; y a su vez variando el lapso de tiempo de la simulación.

Por otro lado en la Figura 6.2., se grafica la comparación al variar sólo el tamaño del océano y mantener el lapso de tiempo (4096) y la cantidad de hilos (8). Finalmente, en la Figura 6.3. se muestra la eficiencia al modificar únicamente la cantidad de hilos y mantener tanto el tamaño del océano (4096x4096) como el lapso de tiempo (4096).

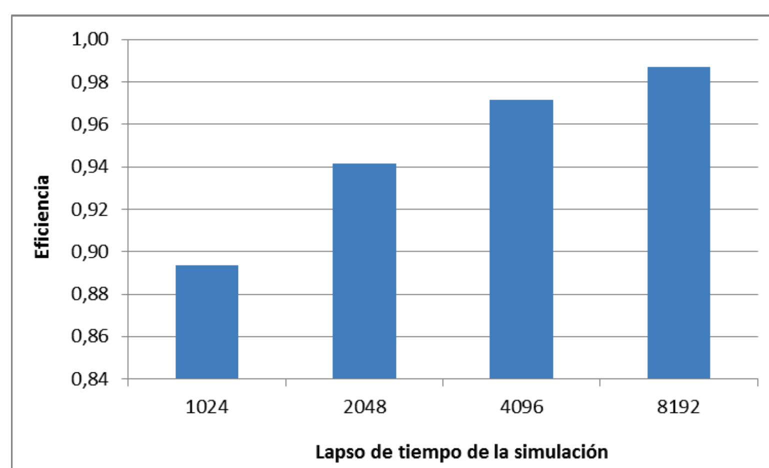


Figura 6.1. Eficiencia lograda en las pruebas con OpenMP usando 8 hilos, en un océano de 4096x4096, variando el lapso de tiempo de la simulación.

Como se puede observar, al aumentar el tamaño del problema (en este caso marcado por la cantidad de momentos a simular), la eficiencia crece levemente. En esto influyen aspectos de inicializaciones y fallos de cache en el algoritmo secuencial que al crecer el cómputo tienen menos peso.

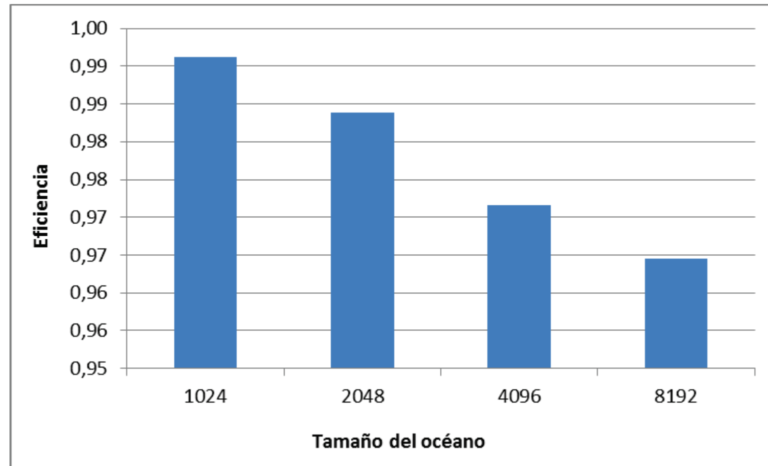


Figura 6.2. Eficiencia lograda en las pruebas con OpenMP usando 8 hilos, en una simulación de 4096 momentos, y variando el tamaño del océano.

En la Figura 6.2. puede observarse que al aumentar el tamaño del problema (en este caso marcado por las dimensiones del océano), la eficiencia decae muy levemente (por los mismos motivos que en el caso anterior).

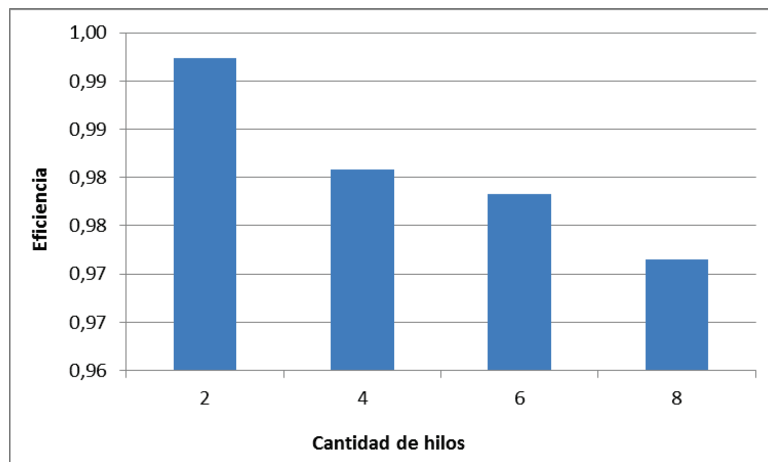


Figura 6.3. Eficiencia lograda en las pruebas con OpenMP en una simulación de 4096 momentos en un océano de 4096x4096, y variando la cantidad de hilos.

De la Figura 6.3. se puede observar, que al aumentar el tamaño de la arquitectura (o cantidad de hilos utilizados dado que se ejecuta un único hilo por núcleo), la eficiencia decae levemente debido a la creación y sincronización de hilos.

Para poder realizar la comparación entre las soluciones con OpenMP y Pthreads es que se ejecutaron las mismas pruebas que las detalladas en OpenMP con la solución que utiliza Pthreads. En la Tabla 6.4. se muestran los tiempos de ejecución del algoritmo con Pthreads. Al igual que para la solución con OpenMP en el Anexo I se encuentra detallada la información respecto al speedup y a la eficiencia.

Escenario	2	4	6	8
1	42,58	21,46	14,32	10,79
2	164,07	81,68	54,79	41,33
3	620,83	315,12	207,68	161,11
4	2220,38	1152,22	767,54	596,95
5	87,27	43,93	29,35	22,02
6	342,68	171,33	114,79	86,44
7	1335,80	673,09	446,76	341,10
8	4733,48	2510,60	1685,00	1300,90
9	176,68	88,90	59,48	44,45
10	700,71	350,67	235,04	176,69
11	2769,66	1390,90	927,14	702,23
12	10343,60	5377,93	3596,63	2740,38
13	355,30	178,87	119,84	89,29
14	1414,70	709,66	475,06	357,37
15	5635,74	2827,94	1886,50	1423,26
16	21813,90	11112,76	7425,62	5621,50

Tabla 6.4. Tiempos de la solución paralela de memoria compartida con Pthreads

Para estudiar la escalabilidad de esta solución también se analizan los mismos datos que para el de OpenMP. En la Figura 6.4. se grafican las eficiencias logradas usando el tamaño de océano 4096x4096, 8 hilos, y variando el lapso de tiempo de la simulación.

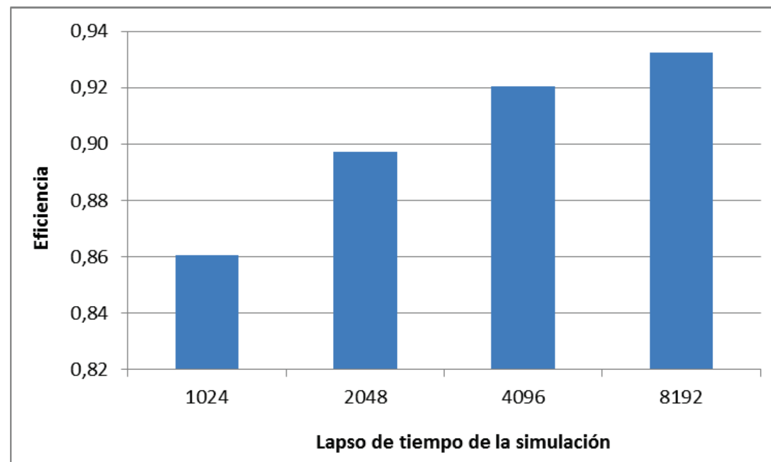


Figura 6.4. Eficiencia lograda en las pruebas con Pthreads usando 8 hilos, en un océano de 4096x4096, variando el lapso de tiempo de la simulación.

De la figura anterior se puede observar, al aumentar el tamaño del problema (en este caso marcado por la cantidad de momentos a simular), la eficiencia se va incrementando.

En la Figura 6.5., se grafica la comparación al variar sólo el tamaño del océano y mantener el lapso de tiempo (4096) y la cantidad de hilos (8).

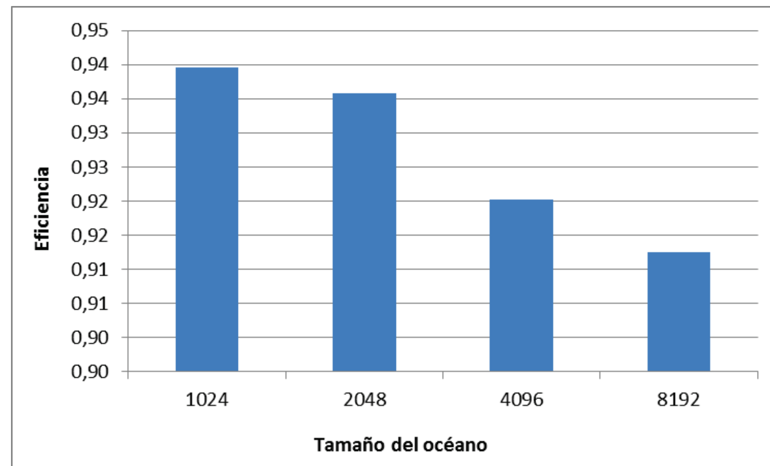


Figura 6.5. Eficiencia lograda en las pruebas con Pthreads usando 8 hilos, en una simulación de 4096 momentos, y variando el tamaño del océano.

Como se puede observar, al aumentar el tamaño del problema (en este caso marcado por las dimensiones del océano), la eficiencia decae levemente.

Finalmente, en la Figura 6.6. se muestra la eficiencia al modificar únicamente la cantidad de hilos y mantener tanto el tamaño del océano (4096x4096) como el lapso de tiempo (4096).

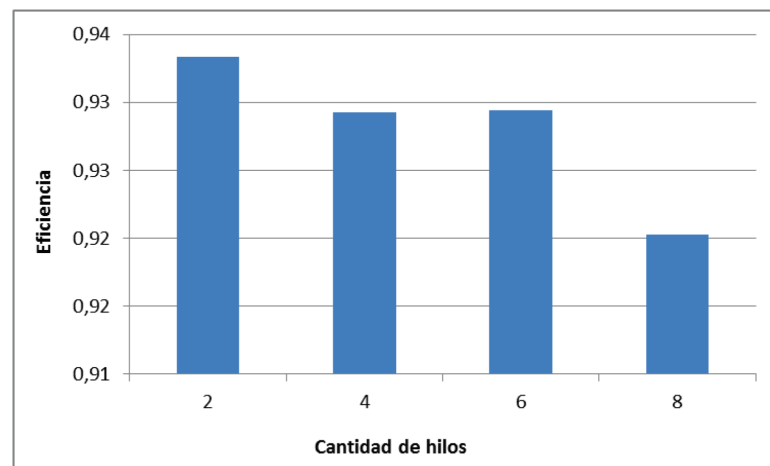


Figura 6.6. Eficiencia lograda en las pruebas con Pthreads en una simulación de 4096 momentos en un océano de 4096x4096, y variando la cantidad de hilos.

Se observa que al aumentar la cantidad de hilos utilizados dado que se ejecuta un único hilo por núcleo), la eficiencia decae muy levemente.

Solución paralela con Pasaje de Mensajes

Para realizar las pruebas de la solución con Pasaje de Mensajes (realizada con la librería OpenMPI), se utilizaron 2 hojas de la arquitectura mencionada anteriormente.

En la Tabla 6.5. se muestran los tiempos de ejecución de la solución con Pasaje de Mensajes. Cada escenario fue probado con 2, 4, 6, 8, 16 y 32 procesos. En el Anexo I se encuentra detallada la información respecto al speedup y a la eficiencia.

Escenario	2	4	6	8	16	32
1	40,43	21,48	16,03	13,92	12,41	11,51
2	157,04	82,97	58,52	49,18	40,38	29,32
3	577,54	303,01	207,74	169,84	118,62	95,65
4	2303,66	1170,60	785,94	610,85	363,37	305,78
5	83,79	43,53	32,35	27,02	25,44	24,32
6	323,91	169,12	120,72	99,65	81,70	72,84
7	1228,58	630,09	430,85	348,99	241,41	201,39
8	4794,10	2447,59	1627,06	1277,70	749,34	609,85
9	167,85	87,44	64,61	58,75	50,84	47,34
10	663,75	341,79	244,26	201,61	163,68	150,87
11	2592,34	1295,82	894,69	716,74	489,66	395,91
12	10010,95	5076,18	3392,44	2632,87	1541,93	1017,75
13	336,74	175,54	127,30	111,02	96,79	91,76
14	1341,13	685,92	493,05	408,55	327,91	298,76
15	5323,94	2654,94	1823,46	1454,17	989,03	758,29
16	20983,76	10463,95	6978,78	5363,42	3126,75	2876,24

Tabla 6.5. Tiempos de la solución paralela de Pasaje de Mensajes (MPI).

De la misma manera que para las soluciones de memoria compartida, se analiza la escalabilidad desde los tres puntos de vista antes mencionados, estos son variación en cuanto al lapso de tiempo de la simulación, en cuanto al tamaño del océano y por último en cuanto a la cantidad de procesos utilizados.

En a Figura 6.7. se muestra la eficiencia lograda usando un tamaño de océano de 4096x4096 y 16 procesos. Además en esta figura se puede observar que al aumentar el tamaño del problema (en este caso marcado por la cantidad de momentos a simular), la eficiencia se va incrementando.

Por su parte en la Figura 6.8. se muestra la eficiencia lograda usando el lapso de tiempo de 4096 y 16 procesos. Al igual que para la figura anterior se observa que al aumentar el tamaño del problema (en este caso marcado por las dimensiones del océano), la eficiencia se incrementa, a diferencia de lo que ocurría con las soluciones de memoria compartida.

Por último en a Figura 6.9. se muestra la eficiencia lograda usando el tamaño del océano de 4096x4096 y el lapso de tiempo de 4096. Como se puede observar en esta figura, al aumentar el tamaño de la arquitectura (o cantidad de procesos utilizados dado que se ejecuta un único proceso por núcleo), la eficiencia decae levemente.

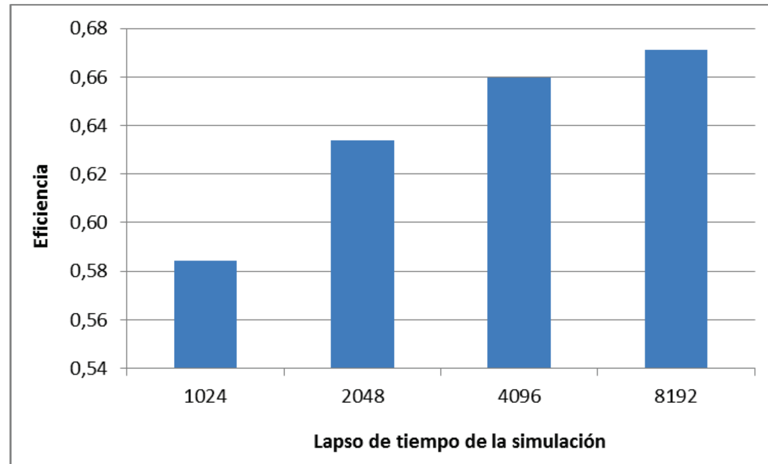


Figura 6.7. Eficiencia lograda en las pruebas con MPI usando 16 procesos, en un océano de 4096x4096, variando el lapso de tiempo de la simulación.

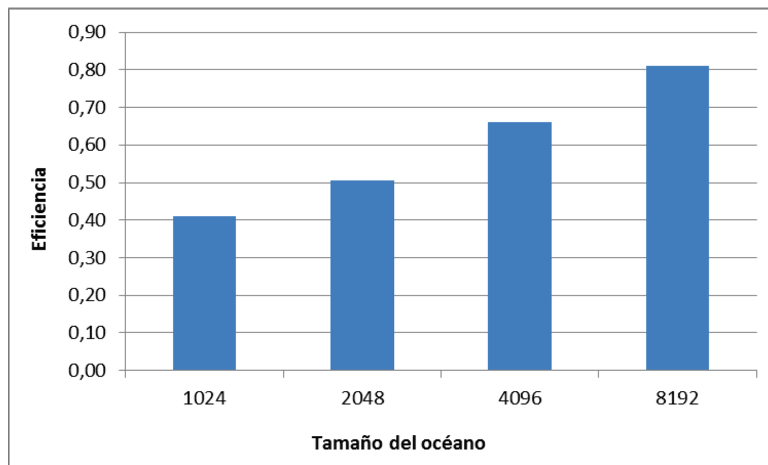


Figura 6.8. Eficiencia lograda en las pruebas con MPI usando 16 procesos, en una simulación de 4096 momentos, y variando el tamaño del océano.

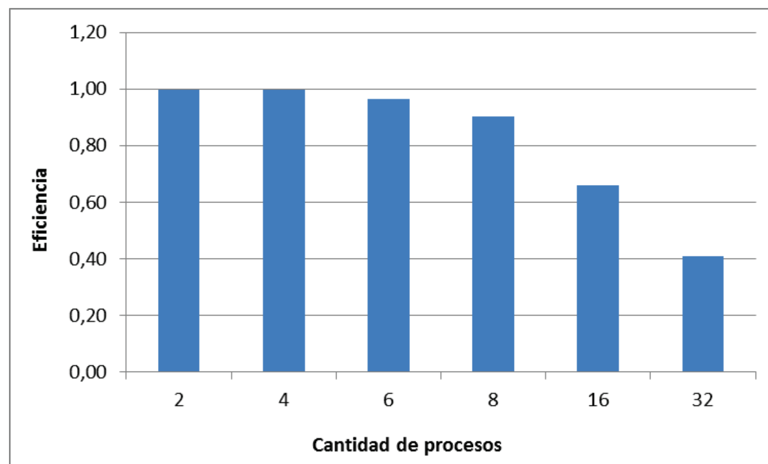


Figura 6.9. Eficiencia lograda en las pruebas con MPI en una simulación de 4096 momentos en un océano de 4096x4096, y variando la cantidad de procesos.

Comparación de las tres soluciones puras (OpenMP, Pthreads y MPI)

En esta Sección se realiza una comparación de la eficiencia que obtiene cada una de las soluciones anteriores. En la Figura 6.10. se muestra gráficamente las diferencia de eficiencia en los 16 escenarios y utilizando 4 hilos/procesos dependiendo el caso.

A su vez en la Figura 6.11. se presenta la misma comparación pero utilizando 8 hilos/procesos.

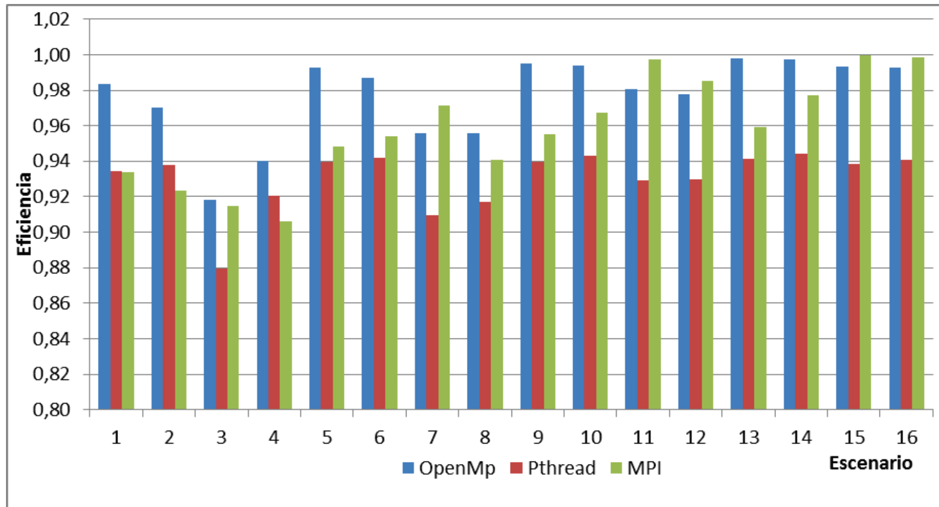


Figura 6.10. Comparación de la eficiencia en los tres algoritmos puros usando 4 hilos/procesos.

Observando el gráfico de la Figura 6.10. se deduce que cuando la cantidad de hilos/procesos no es grande (2 y 4), en general la eficiencia lograda con OpenMP y MPI es parecida y muy superior a la de Pthreads.

Además cuanto más grande es el problema (tamaño del océano y lapso de tiempo de la simulación) más se nota la aproximación de los resultados de MPI a los de OpenMP, y en algunos casos incluso la superioridad.

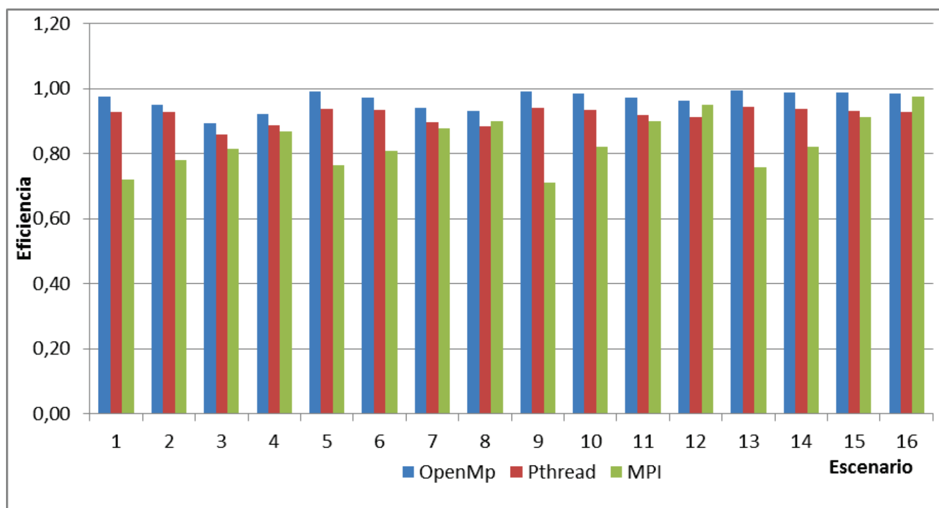


Figura 6.11. Comparación de la eficiencia en los tres algoritmos puros usando 8 hilos/procesos.

Como se puede notar en el gráfico de la Figura 6.11, las soluciones de memoria compartida se comportan (en general) mejor que las de Pasaje de Mensajes cuando se trabaja con una mayor cantidad de hilos/procesos (6 y 8). Esto se debe en parte a que se puede aprovechar eficientemente los niveles de memoria compartida de la arquitectura teniendo en cuenta que este tipo de problema requiere excesivo uso de memoria, evitando de esta manera las comunicaciones y sincronizaciones implícitas del PM.

Teniendo en cuenta estos resultados, sobre todo al usar mayor cantidad de hilos/procesos, donde es más eficiente la solución con OpenMP, y la imposibilidad de esta de ejecutarse en más de una hoja, da relevancia a la posibilidad de mejorar el rendimiento utilizando un algoritmo híbrido que aproveche la eficiencia de la solución con OpenMP dentro de cada multicore, y la comunicación entre ellos por medio de MPI.

Soluciones paralelas Híbridas MPI + (OpenMP o Pthreads)

En este punto se analizan los resultados obtenidos por las soluciones híbridas implementadas con OpenMPI para la comunicación entre procesos, y dentro de cada uno de estos OpenMP o Pthreads para el manejo de los hilos.

La Tabla 6.6. muestra los tiempos de ejecución del algoritmo híbrido que usa Pthreads. En esta tabla se indica la cantidad de proceso MPI usados, y la cantidad de hilos dentro de cada uno de ellos (2-4 significa 2 procesos MPI con 4 hilos en cada uno de ellos).

Escenario	2-2	2-4	2-8	4-8
1	21,82	10,77	5,60	18,93
2	135,72	51,21	33,58	51,04
3	498,78	248,57	122,73	165,25
4	1858,15	933,45	481,48	546,17
5	43,66	22,09	11,44	36,13
6	284,93	110,54	70,66	94,83
7	1090,21	542,92	271,56	311,44
8	4088,69	2024,60	1045,54	1030,22
9	90,19	44,70	22,94	71,71
10	583,49	202,37	145,62	182,06
11	2285,55	1141,42	573,51	619,09
12	8864,88	4389,49	2238,59	2082,02
13	181,17	90,04	46,03	139,25
14	1179,47	415,65	292,63	360,30
15	4679,22	2339,48	1182,45	1236,81
16	18410,40	9169,29	4633,28	4198,71

Tabla 6.6. Tiempo de ejecución de las pruebas de la solución híbrida con Pthreads.

Para esta solución se analiza la escalabilidad desde los tres puntos de vista antes mencionados para la solución anterior: variación en cuanto al lapso de tiempo de la simulación, variación en el tamaño del océano y variación en cuanto a la cantidad de procesos.

En la Figura 6.12. se muestra la eficiencia lograda usando el mismo tamaño de océano de 4096x4096, 2 procesos y 8 hilos.

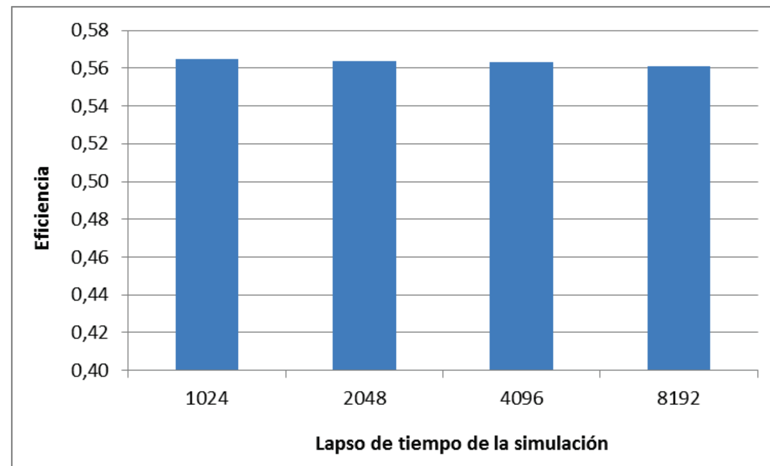


Figura 6.12. Eficiencia lograda en las pruebas de la solución híbrida con Pthreads usando 2 procesos y 8 hilos por cada uno, en un océano de 4096x4096, variando el lapso de tiempo de la simulación.

Como se puede observar, al aumentar el tamaño del problema (en este caso marcado por la cantidad de momentos a simular), la eficiencia se mantiene con una diferencia menos al 0,2 %.

Por otro lado en la Figura 6.13. se muestra la eficiencia lograda usando el lapso de tiempo de 4096, 2 procesos y 8 hilos.

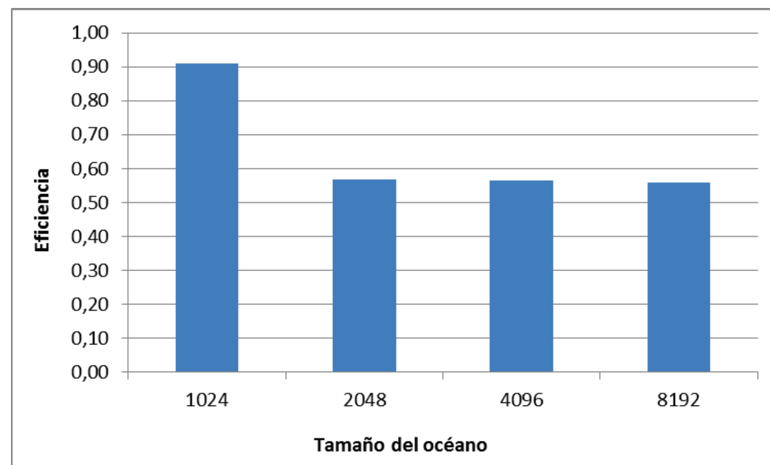


Figura 6.13. Eficiencia lograda en las pruebas de la solución híbrida con Pthreads usando 2 procesos y 8 hilos por cada uno, en una simulación de 4096 momentos, y variando el tamaño del océano.

Como se puede observar, al aumentar el tamaño del problema (en este caso marcado por las dimensiones del océano), la eficiencia se decrementa, al igual que lo hacía en Memoria Compartida.

Por último en la Figura 6.14. se muestra la eficiencia lograda usando el tamaño de océano de 4096x4096 y un lapso de tiempo de 4096.

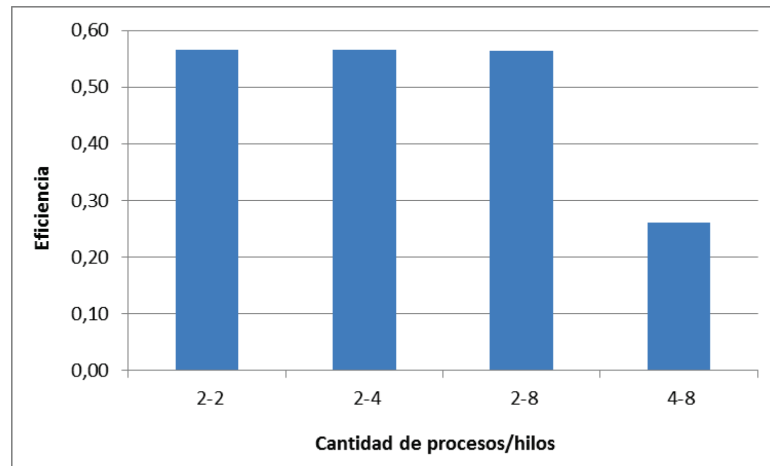


Figura 6.14. Eficiencia lograda en las pruebas de la solución híbrida con Pthreads en una simulación de 4096 momentos en un océano de 4096x4096, y variando la cantidad de procesos e hilos.

En este caso, la eficiencia se mantiene dentro de un 2% de diferencia, sin seguir una función ascendente o descendente, sino que altera la suba y baja del resultado.

La Tabla 6.7. muestra los tiempos de ejecución del algoritmo híbrido que usa OpenMP. En esta tabla se indica la cantidad de proceso MPI usados, y la cantidad de hilos dentro de cada uno de ellos (2-4 significa 2 procesos MPI con 4 hilos en cada uno de ellos).

Escenario	2-2	2-4	2-8	4-8
1	31,45	15,92	8,48	4,86
2	123,08	62,48	32,12	17,36
3	467,59	233,86	119,82	64,93
4	1791,78	923,67	465,99	243,95
5	63,88	32,02	16,78	9,71
6	251,06	126,89	64,72	34,36
7	992,22	481,51	243,80	128,54
8	3747,90	1907,12	955,91	489,55
9	127,93	64,18	33,35	18,59
10	512,67	263,75	131,79	67,77
11	1998,49	990,77	498,47	257,61
12	7807,75	3940,43	1971,57	987,15
13	258,59	130,61	67,26	36,90
14	1019,69	513,48	261,05	135,46
15	4046,11	2020,58	1011,76	517,89
16	15967,70	8016,77	4015,06	2018,84

Tabla 6.7. Tiempo de ejecución de las pruebas de la solución híbrida con OpenMP.

Para esta solución se analiza la escalabilidad desde los tres puntos de vista antes mencionados para la solución anterior: variación en cuanto al lapso de tiempo de la simulación, variación en el tamaño del océano y variación en cuanto a la cantidad de procesos.

En la Figura 6.15. se muestra la eficiencia lograda usando el mismo tamaño de océano de 4096x4096, 2 procesos y 8 hilos. Por otro lado en la Figura 6.16. se muestra la eficiencia lograda usando el lapso de tiempo de 4096, 2 procesos y 8 hilos. Por último en la Figura 6.17. se muestra la eficiencia lograda usando el tamaño de océano de 4096x4096 y un lapso de tiempo de 4096.

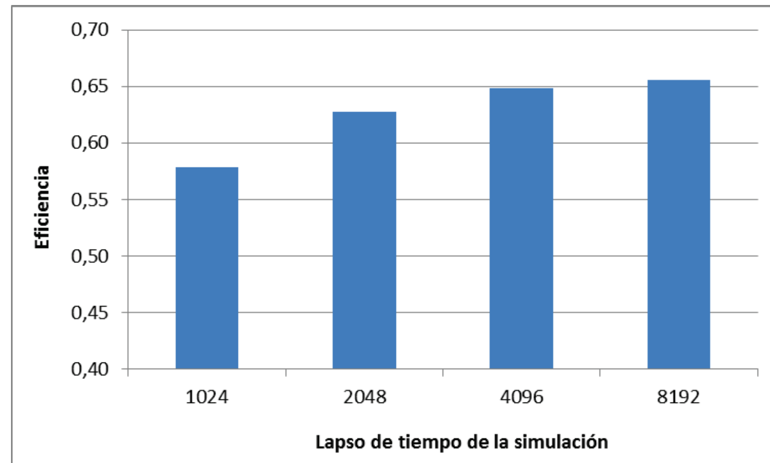


Figura 6.15. Eficiencia lograda en las pruebas de la solución híbrida con OpenMP usando 2 procesos y 8 hilos por cada uno, en un océano de 4096x4096, variando el lapso de tiempo de la simulación.

Como se puede observar, al aumentar el tamaño del problema (en este caso marcado por la cantidad de momentos a simular), la eficiencia se incrementa, tendiendo a estabilizarse en 0,65 (en este caso).

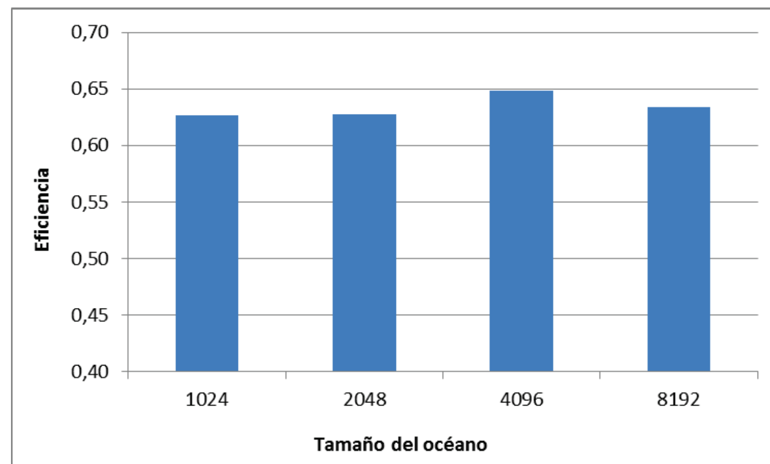


Figura 6.16. Eficiencia lograda en las pruebas de la solución híbrida con OpenMP usando 2 procesos y 8 hilos por cada uno, en una simulación de 4096 momentos, y variando el tamaño del océano.

Como se puede observar, al aumentar el tamaño del problema (en este caso marcado por las dimensiones del océano), la eficiencia se mantiene prácticamente estable con un rango de variación de 2%.

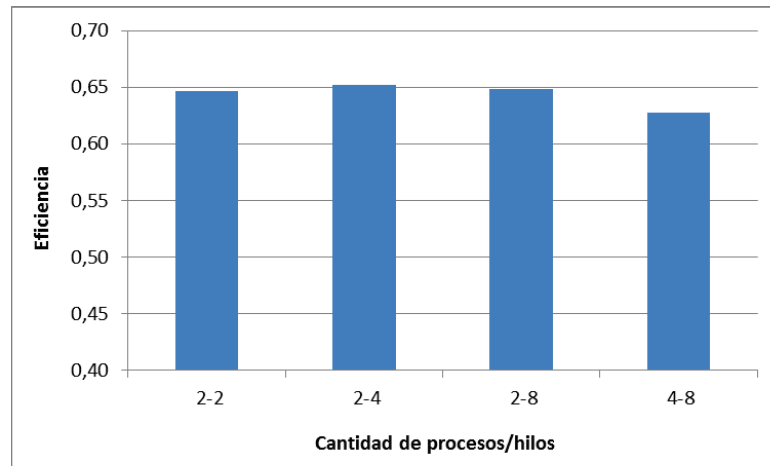


Figura 6.17. Eficiencia lograda en las pruebas de la solución híbrida con OpenMP en una simulación de 4096 momentos en un océano de 4096x4096, y variando la cantidad de procesos e hilos.

En este caso, la eficiencia se mantiene dentro de un 2% de diferencia, sin seguir una función ascendente o descendente.

Comparación de las soluciones híbridas y la de Pasaje de Mensajes

En esta Sección se compara la eficiencia lograda por los algoritmos híbridos y el de pasaje de mensajes para analizar la diferencia entre ellos. En la Figura 6.18 se muestra gráficamente esta diferencia en los 16 escenarios y utilizando 4 núcleos (cantidad total de procesos e hilos dependiendo el caso). En las figuras 6.19, 6.20 y 6.21 se presenta la misma comparación pero utilizando 8, 16 y 32 núcleos respectivamente.

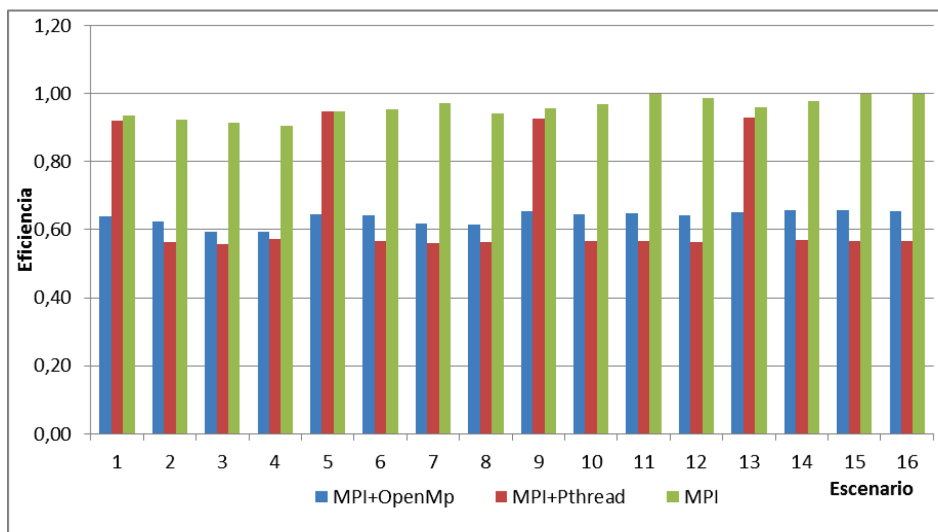


Figura 6.18. Comparación de la eficiencia en los tres algoritmos usando 4 núcleos.

Como se ve en el gráfico anterior, la solución con pasaje de mensajes se comporta más eficientemente que las soluciones híbridas. Esta diferencia es muy marcada, excepto en los tamaños de océano más chicos. Entre las otras dos soluciones la que usa OpenMp es mejor, excepto en los casos donde el océano es más chico (1024x1024).

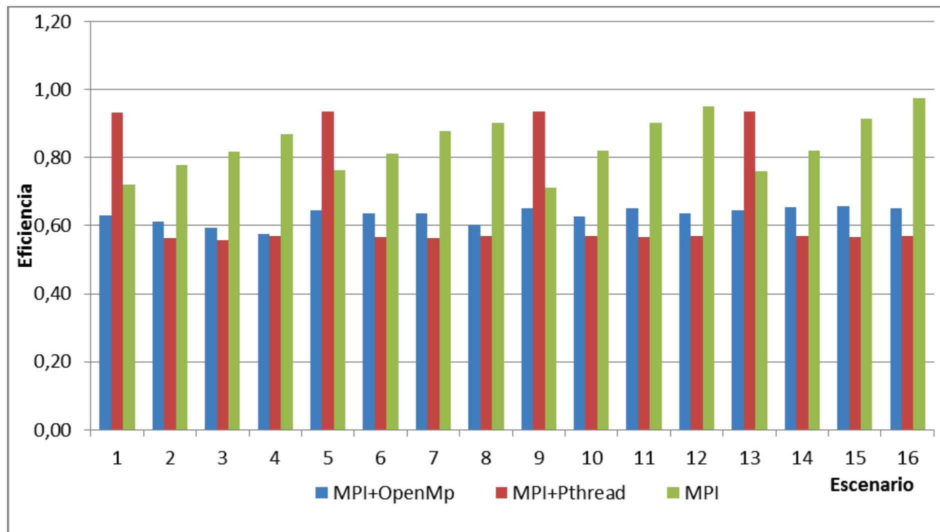


Figura 6.19. Comparación de la eficiencia en los tres algoritmos usando 8 núcleos.

Como se observa en el gráfico anterior, la solución con pasaje de mensajes sigue comportándose (en general) de forma más eficientemente que las híbridas. Esta diferencia es menor que al trabajar con 4 núcleos. Mientras, entre ambas soluciones híbridas, la diferencia se mantiene igual que con 4 núcleos.

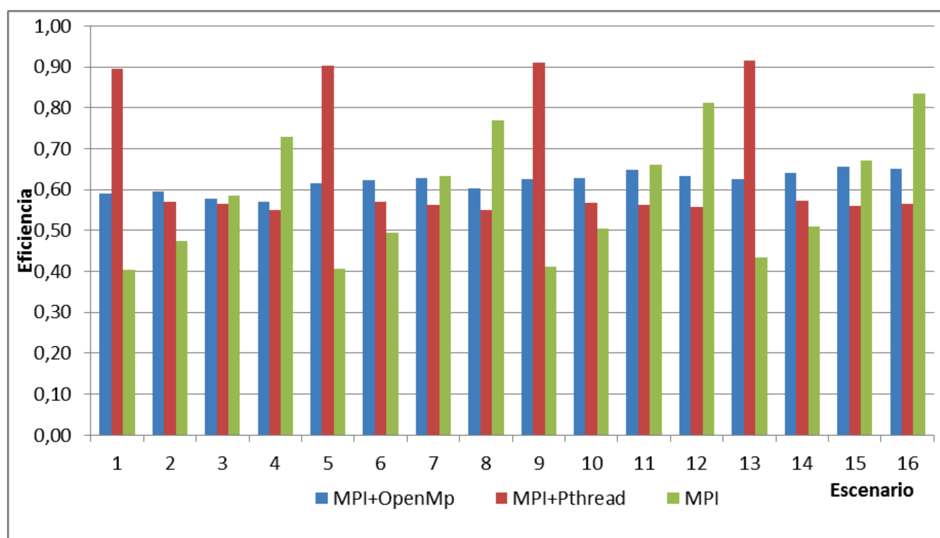


Figura 6.20. Comparación de la eficiencia en los tres algoritmos usando 16 núcleos.

Al trabajar con mayor cantidad de núcleos, se sigue la tendencia anterior, reduciendo cada vez más la diferencia de eficiencias entre la solución de pasaje de mensajes y la híbrida con OpenMP. La solución con MPI supera a la hibrida con OpenMP en los tamaños de océanos mayores, mientras que en los tamaños menores se da la inversa.

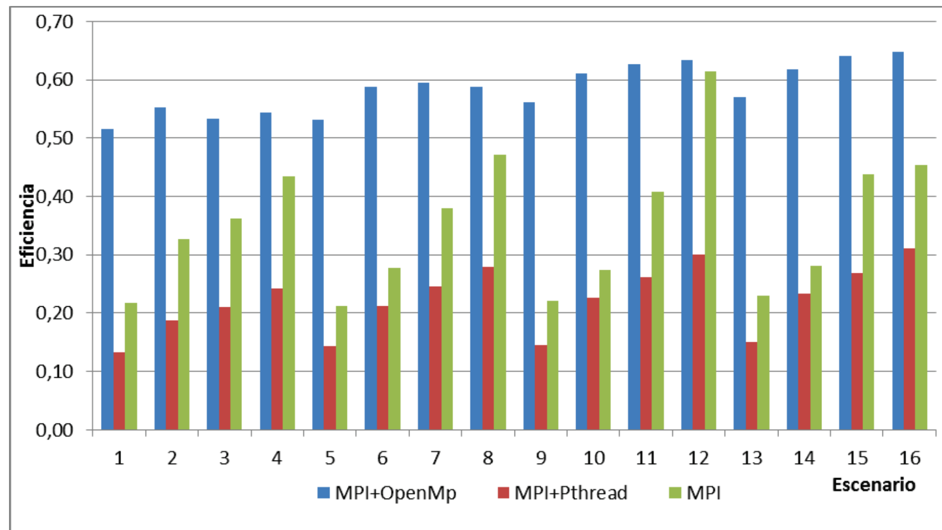


Figura 6.21. Comparación de la eficiencia en los tres algoritmos usando 32 núcleos.

Al trabajar con los 32 núcleos de la arquitectura, se verifica la tendencia de los gráficos anteriores donde en todos los escenarios se comporta mejor el algoritmo híbrido con OpenMp que el de Pasaje de Mensajes puro, y a su vez este último obtiene mejores resultados que el híbrido con Pthreads.

Capítulo 7: Conclusiones y Trabajos Futuros

En este trabajo se realizó Investigación y Desarrollo de algoritmos paralelos sobre arquitecturas de soporte actuales de múltiples núcleos (como multicores y cluster de multicores), con un ámbito de aplicación como los problemas de simulaciones con alta demanda computacional.

Actualmente, la simulación de sistemas ocupa una amplia etapa en los proyectos de varias disciplinas, ya sea porque se necesita tener una predicción de una situación futura y los altos costos de realizar el proyecto sin tener una orientación en ese aspecto lo requieren, o bien porque se necesita tener una representación de un sistema con ciertas condiciones preestablecidas.

La simulación de sistemas es un área en constante evolución, y existen varias razones para pensar en el desarrollo de modelos de simulación mediante herramientas paralelas (entre ellas el alto tiempo de duración de la simulación y la gran necesidad de cómputo para simular escenarios complejos).

El caso de estudio elegido fue un problema conocido como lo es el de *sharks and fishes*, cuyo patrón algorítmico es representativo de otros problemas de cómputo científico.

Se plantearon diferentes soluciones paralelas utilizando distintos modelos de comunicación: memoria compartida, pasaje de mensajes e híbrido combinando ambos esquemas. Asimismo, se evaluó la performance de las soluciones obtenidas en cuanto a tiempo de ejecución, speedup y eficiencia. También se analizó la escalabilidad desde tres puntos de vista: variando el tamaño del océano (1024x1024; 2048x2048; 4096x4096; 8192x8192), modificando el lapso de tiempo de la simulación (1024; 2048; 4096; 8192 momentos), y utilizando diferentes cantidades de procesos y/o hilos.

En cuanto a las implementaciones, se utilizó el lenguaje C con las librerías OpenMPI, Pthreads y OpenMP, sobre una arquitectura Blade de 8 hojas, con 2 procesadores quad core Intel Xeón e5405 de 2.0 GHz en cada una de ellas. Se desarrollaron 2 soluciones con memoria compartida (Pthreads y OpenMP), una solución con pasaje de mensajes (OpenMPI) y 2 soluciones híbridas (OpenMPI + Pthreads y OpenMPI + OpenMP).

Entre los resultados más significativos pueden mencionarse:

- En cuanto a las soluciones con memoria compartida, la implementación OpenMP se comportó con mejores tiempos de ejecución que la de Pthreads. Esto se debe a que el primero se encuentra optimizado para este tipo de problemas. Asimismo,

en la solución OpenMP, al crecer el tamaño del problema dado por la cantidad de pasos a simular y mantener fija el número de hilos, la eficiencia aumenta muy levemente. Por otra parte, manteniendo fijo el tamaño del problema y variando la cantidad de hilos, la pérdida de eficiencia es mínima.

- En cuanto a la solución con pasaje de mensajes, en la experimentación realizada siempre que crece el tamaño del problema la eficiencia aumenta.

- Si se comparan las soluciones “puras”, cuando la cantidad de hilos/procesos no es grande (2 y 4), en general la eficiencia lograda con OpenMP y MPI es similar y superior a la de Pthreads. Además cuanto más grande es el problema (tamaño del océano y lapso de tiempo de la simulación) más se nota la aproximación de los resultados de MPI a los de OpenMP, y en algunos casos incluso la superioridad.

- Las soluciones de memoria compartida se comportan (en general) mejor que las de Pasaje de Mensajes cuando se trabaja con una mayor cantidad de hilos/procesos (6 y 8). Esto se debe en parte a que se puede aprovechar eficientemente los niveles de memoria compartida de la arquitectura teniendo en cuenta que este tipo de problema requiere extensivo uso de memoria, evitando de esta manera las comunicaciones explícitas y sincronizaciones implícitas del PM.

- En cuanto a las soluciones híbridas, como era esperable la que utiliza OpenMPI + OpenMP se comporta de manera más eficiente que la de OpenMPI + Pthreads.

- Si se comparan las soluciones híbridas con las de Pasaje de Mensajes, a medida que se utiliza una arquitectura con más núcleos, en la solución híbrida se observa que si bien en ambos casos la eficiencia cae, este decremento es mayor en el caso de Pasaje de Mensajes. De esta forma, la solución híbrida se comporta mejor.

Como trabajos futuros se plantea:

- Extender el estudio de escalabilidad para tamaños de problema más grandes

- Estudiar alternativas de solución mediante estrategias optimistas.

- Extender la aplicación de las estrategias y los resultados obtenidos a otros problemas de cómputo científico que presenten características similares.

- Incorporar heterogeneidad en la arquitectura, estudiando el impacto sobre el mapeo de procesos y datos a procesadores.

Bibliografía

[AMD67] Amdahl, G.M. “Validity of the single-processor approach to achieving large scale computing capabilities”. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485

[AND00] Andrews G. “Foundations of Multithreaded, Parallel and Distributed Programming” Addison Wesley Higher Education 2000. ISBN-13:9780201357523 .

[BAR10] Barney B, “Introduction to parallel computing”, Lawrence Livermore National Laboratory, 2010.

[CHA07] Lei Chai, Qi Gao, Dhabaleswar K Panda, “Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System”, IEEE International Symposium on Cluster Computing and the Grid 2007 (CCGRID 2007), pp. 471-478, 2007.

[COS05] Coss Bu R., “Simulación Un enfoque práctico”, 2da Edición, Limusa, 2005.

[DEG08] De Giusti L. “Mapping sobre Arquitecturas Heterogéneas”. Tesis Doctoral, Universidad Nacional de La Plata (2008).

[DON03] Dongarra J. , Foster I., Fox G., Gropp W., Kennedy K., Torzcon L., White A. “Sourcebook of Parallel computing”. Morgan Kaufmann Publishers – ISBN 155860 871 0 (Capítulo 3).

[FER95] Ferscha A., “Parallel and Distributed Simulation of Discrete Event Systems”, Handbook of Parallel and Distributed Computing, McGraw-Hill, 1995.

[FUJ99] Fujimoto R.M., “Parallel and Distribution Simulation Systems”, 1st Edition, John Wiley & Sons Inc., 1999.

[GCC] http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[GRA03] Grama A, Gupta A, Karypis G, Kumar V, “Introduction Parallel Computing”, Pearson Addison Wesley, 2da Edición, 2003.

[GUS88] Gustafson, J.L. “Reevaluating Amdahl's Law”. CACM, 31(5), 1988. pp. 532-533.

[INS10] Instituto de Investigación en Informática III-LIDI, “Reporte técnico: Soluciones híbridas - MPI + OpenMP”, Junio de 2010.

[LEO01] Leopold C., “Parallel and Distributed Computing. A Survey of Models, Paradigms and Approaches”. Wiley,2001. ISBN: 0471358312 (Capítulos 1, 2 y 3).

[LIN] http://www.linuxjournal.com/article/7269

[McC07] Mc Cool M, “Programming models for scalable multicore programming”, 2007, http://www.hpcwire.com/features/17902939.html.

[MIS86] Misra J., “Distributed Discrete-Event Simulation”, ACM Computing Surveys, Vol. 18, Nº 1, 1986.

[OHI96] Ohio Supercomputer Center, “MPI Primer / Developing with LAM”, The Ohio State University, 1996.

[OPE] https://computing.llnl.gov/tutorials/openMP

[PTH] https://computing.llnl.gov/tutorials/pthreads

[RAU10] Rauber T, Rünger G, “Parallel programming for Multicore and Cluster Systems”, Springer, 2010.

[WIL05] Wilkinson B, Allen M, “Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers”, 2da Edición, Pearson Prentice Hall, 2005.

Anexo I: Resultados completos

Resultados Obtenidos de la solución con OpenMP

En la Tabla AI.1., AI.2. y AI.3. se muestran los tiempos de ejecución, speedup y eficiencia respectivamente de la solución de memoria compartida con OpenMP.

Escenario	Momento	Tamaño	2	4	6	8
1	1024	1024x1024	40,30	20,39	13,56	10,29
2	1024	2048x2048	153,85	78,98	53,43	40,25
3	1024	4096x4096	589,05	302,02	202,70	155,14
4	1024	8192x8192	2215,78	1128,10	758,79	575,44
5	2048	1024x1024	82,89	41,59	27,80	20,84
6	2048	2048x2048	323,65	163,52	109,98	82,85
7	2048	4096x4096	1254,24	640,45	427,81	325,08
8	2048	8192x8192	4705,26	2408,96	1640,13	1235,83
9	4096	1024x1024	167,48	83,94	56,17	42,13
10	4096	2048x2048	663,52	332,74	223,26	168,08
11	4096	4096x4096	2604,97	1317,73	880,76	665,17
12	4096	8192x8192	10078,94	5114,22	3446,07	2592,32
13	8192	1024x1024	337,58	168,69	112,98	84,62
14	8192	2048x2048	1345,88	671,88	449,56	338,60
15	8192	4096x4096	5324,91	2671,85	1787,15	1345,02
16	8192	8192x8192	20962,01	10528,40	7058,79	5306,90

Tabla AI.1. Tiempos de la solución paralela de memoria compartida con OpenMP.

Escenario	Momento	Tamaño	2	4	6	8
1	1024	1024x1024	1,99	3,93	5,92	7,80
2	1024	2048x2048	1,99	3,88	5,74	7,61
3	1024	4096x4096	1,88	3,67	5,47	7,15
4	1024	8192x8192	1,92	3,76	5,59	7,37
5	2048	1024x1024	1,99	3,97	5,94	7,92
6	2048	2048x2048	1,99	3,95	5,87	7,79
7	2048	4096x4096	1,95	3,82	5,72	7,53
8	2048	8192x8192	1,96	3,82	5,62	7,45
9	4096	1024x1024	1,99	3,98	5,95	7,93
10	4096	2048x2048	1,99	3,98	5,93	7,87
11	4096	4096x4096	1,98	3,92	5,87	7,77
12	4096	8192x8192	1,98	3,91	5,80	7,72
13	8192	1024x1024	1,99	3,99	5,96	7,96
14	8192	2048x2048	1,99	3,99	5,96	7,92
15	8192	4096x4096	1,99	3,97	5,94	7,89
16	8192	8192x8192	1,99	3,97	5,92	7,88

Tabla AI.2. Speedup logrado en la solución paralela de memoria compartida con OpenMP.

Escenario	Momento	Tamaño	2	4	6	8
1	1024	1024x1024	1,00	0,98	0,99	0,97
2	1024	2048x2048	1,00	0,97	0,96	0,95
3	1024	4096x4096	0,94	0,92	0,91	0,89
4	1024	8192x8192	0,96	0,94	0,93	0,92
5	2048	1024x1024	1,00	0,99	0,99	0,99
6	2048	2048x2048	1,00	0,99	0,98	0,97
7	2048	4096x4096	0,98	0,96	0,95	0,94
8	2048	8192x8192	0,98	0,96	0,94	0,93
9	4096	1024x1024	1,00	0,99	0,99	0,99
10	4096	2048x2048	1,00	0,99	0,99	0,98
11	4096	4096x4096	0,99	0,98	0,98	0,97
12	4096	8192x8192	0,99	0,98	0,97	0,96
13	8192	1024x1024	1,00	1,00	0,99	0,99
14	8192	2048x2048	1,00	1,00	0,99	0,99
15	8192	4096x4096	1,00	0,99	0,99	0,99
16	8192	8192x8192	1,00	0,99	0,99	0,98

Tabla AI.3. Eficiencia lograda en la solución paralela de memoria compartida con OpenMP.

Resultados Obtenidos de la solución con Pthreads

En la Tabla AI.4., AI.5. y AI.6. se muestran los tiempos de ejecución, speedup y eficiencia respectivamente de la solución de memoria compartida con Pthreads.

Escenario	Momento	Tamaño	2	4	6	8
1	1024	1024x1024	42,58	21,46	14,32	10,79
2	1024	2048x2048	164,07	81,68	54,79	41,33
3	1024	4096x4096	620,83	315,12	207,68	161,11
4	1024	8192x8192	2220,38	1152,22	767,54	596,95
5	2048	1024x1024	87,27	43,93	29,35	22,02
6	2048	2048x2048	342,68	171,33	114,79	86,44
7	2048	4096x4096	1335,80	673,09	446,76	341,10
8	2048	8192x8192	4733,48	2510,60	1685,00	1300,90
9	4096	1024x1024	176,68	88,90	59,48	44,45
10	4096	2048x2048	700,71	350,67	235,04	176,69
11	4096	4096x4096	2769,66	1390,90	927,14	702,23
12	4096	8192x8192	10343,60	5377,93	3596,63	2740,38
13	8192	1024x1024	355,30	178,87	119,84	89,29
14	8192	2048x2048	1414,70	709,66	475,06	357,37
15	8192	4096x4096	5635,74	2827,94	1886,50	1423,26
16	8192	8192x8192	21813,90	11112,76	7425,62	5621,50

Tabla AI.4. Tiempos de la solución paralela de memoria compartida con Pthreads.

Escenario	Momento	Tamaño	2	4	6	8
1	1024	1024x1024	1,88	3,74	5,60	7,44
2	1024	2048x2048	1,87	3,75	5,59	7,42
3	1024	4096x4096	1,79	3,52	5,34	6,88
4	1024	8192x8192	1,91	3,68	5,53	7,11
5	2048	1024x1024	1,89	3,76	5,63	7,50
6	2048	2048x2048	1,88	3,77	5,62	7,47
7	2048	4096x4096	1,83	3,64	5,48	7,18
8	2048	8192x8192	1,95	3,67	5,47	7,08
9	4096	1024x1024	1,89	3,76	5,62	7,52
10	4096	2048x2048	1,89	3,77	5,63	7,49
11	4096	4096x4096	1,87	3,72	5,58	7,36
12	4096	8192x8192	1,93	3,72	5,56	7,30
13	8192	1024x1024	1,90	3,76	5,62	7,54
14	8192	2048x2048	1,89	3,78	5,64	7,50
15	8192	4096x4096	1,88	3,75	5,63	7,46
16	8192	8192x8192	1,92	3,76	5,63	7,44

Tabla AI.5. Speedup logrado en la solución paralela de memoria compartida con Pthreads.

Escenario	Momento	Tamaño	2	4	6	8
1	1024	1024x1024	0,94	0,93	0,93	0,93
2	1024	2048x2048	0,93	0,94	0,93	0,93
3	1024	4096x4096	0,89	0,88	0,89	0,86
4	1024	8192x8192	0,96	0,92	0,92	0,89
5	2048	1024x1024	0,95	0,94	0,94	0,94
6	2048	2048x2048	0,94	0,94	0,94	0,93
7	2048	4096x4096	0,92	0,91	0,91	0,90
8	2048	8192x8192	0,97	0,92	0,91	0,89
9	4096	1024x1024	0,95	0,94	0,94	0,94
10	4096	2048x2048	0,94	0,94	0,94	0,94
11	4096	4096x4096	0,93	0,93	0,93	0,92
12	4096	8192x8192	0,97	0,93	0,93	0,91
13	8192	1024x1024	0,95	0,94	0,94	0,94
14	8192	2048x2048	0,95	0,94	0,94	0,94
15	8192	4096x4096	0,94	0,94	0,94	0,93
16	8192	8192x8192	0,96	0,94	0,94	0,93

Tabla AI.6. Eficiencia lograda en la solución paralela de memoria compartida con Pthreads.

Resultados Obtenidos de la solución con Pasaje de Mensajes

En la Tabla AI.7., AI.8. y AI.9. se muestran los tiempos de ejecución, speedup y eficiencia respectivamente de la solución de pasaje de mensajes (MPI).

Escenario	Momento	Tamaño	2	4	6	8	16	32
1	1024	1024x1024	40,43	21,48	16,03	13,92	12,41	11,51
2	1024	2048x2048	157,04	82,97	58,52	49,18	40,38	29,32
3	1024	4096x4096	577,54	303,01	207,74	169,84	118,62	95,65
4	1024	8192x8192	2303,66	1170,60	785,94	610,85	363,37	305,78
5	2048	1024x1024	83,79	43,53	32,35	27,02	25,44	24,32
6	2048	2048x2048	323,91	169,12	120,72	99,65	81,70	72,84
7	2048	4096x4096	1228,58	630,09	430,85	348,99	241,41	201,39
8	2048	8192x8192	4794,10	2447,59	1627,06	1277,70	749,34	609,85
9	4096	1024x1024	167,85	87,44	64,61	58,75	50,84	47,34
10	4096	2048x2048	663,75	341,79	244,26	201,61	163,68	150,87
11	4096	4096x4096	2592,34	1295,82	894,69	716,74	489,66	395,91
12	4096	8192x8192	10010,95	5076,18	3392,44	2632,87	1541,93	1017,75
13	8192	1024x1024	336,74	175,54	127,30	111,02	96,79	91,76
14	8192	2048x2048	1341,13	685,92	493,05	408,55	327,91	298,76
15	8192	4096x4096	5323,94	2654,94	1823,46	1454,17	989,03	758,29
16	8192	8192x8192	20983,76	10463,95	6978,78	5363,42	3126,75	2876,24

Tabla AI.7. Tiempos de la solución paralela con MPI.

Escenario	Momento	Tamaño	2	4	6	8	16	32
1	1024	1024x1024	1,98	3,74	5,00	5,76	6,47	6,97
2	1024	2048x2048	1,95	3,69	5,24	6,23	7,59	10,45
3	1024	4096x4096	1,92	3,66	5,34	6,53	9,35	11,59
4	1024	8192x8192	1,84	3,63	5,40	6,95	11,68	13,88
5	2048	1024x1024	1,97	3,79	5,10	6,11	6,49	6,79
6	2048	2048x2048	1,99	3,82	5,35	6,48	7,90	8,86
7	2048	4096x4096	1,99	3,89	5,68	7,02	10,14	12,16
8	2048	8192x8192	1,92	3,76	5,66	7,21	12,29	15,10
9	4096	1024x1024	1,99	3,82	5,17	5,69	6,57	7,06
10	4096	2048x2048	1,99	3,87	5,42	6,56	8,08	8,77
11	4096	4096x4096	1,99	3,99	5,78	7,21	10,56	13,06
12	4096	8192x8192	2,00	3,94	5,90	7,60	12,97	19,65
13	8192	1024x1024	2,00	3,84	5,29	6,07	6,96	7,34
14	8192	2048x2048	2,00	3,91	5,44	6,56	8,17	8,97
15	8192	4096x4096	1,99	4,00	5,82	7,30	10,74	14,00
16	8192	8192x8192	1,99	4,00	5,99	7,79	13,37	14,53

Tabla AI.8. Speedup logrado en la solución paralela con MPI.

Escenario	Momento	Tamaño	2	4	6	8
1	1024	1024x1024	0,99	0,93	0,83	0,72
2	1024	2048x2048	0,98	0,92	0,87	0,78
3	1024	4096x4096	0,96	0,92	0,89	0,82
4	1024	8192x8192	0,92	0,91	0,90	0,87
5	2048	1024x1024	0,99	0,95	0,85	0,76
6	2048	2048x2048	1,00	0,95	0,89	0,81
7	2048	4096x4096	1,00	0,97	0,95	0,88
8	2048	8192x8192	0,96	0,94	0,94	0,90
9	4096	1024x1024	1,00	0,96	0,86	0,71
10	4096	2048x2048	1,00	0,97	0,90	0,82
11	4096	4096x4096	1,00	1,00	0,96	0,90
12	4096	8192x8192	1,00	0,99	0,98	0,95
13	8192	1024x1024	1,00	0,96	0,88	0,76
14	8192	2048x2048	1,00	0,98	0,91	0,82
15	8192	4096x4096	1,00	1,00	0,97	0,91
16	8192	8192x8192	1,00	1,00	1,00	0,97

Tabla AI.9. Eficiencia lograda en la solución paralela con MPI.

Resultados Obtenidos de la solución híbrida con Pthreads

En la Tabla AI.10., AI.11. y AI.12. se muestran los tiempos de ejecución, speedup y eficiencia respectivamente de la solución híbrida con Pthreads (MPI+Pthreads).

Escenario	Momento	Tamaño	2-2	2-4	2-8	4-8
1	1024	1024x1024	21,82	10,77	5,60	18,93
2	1024	2048x2048	135,72	68,08	33,58	51,04
3	1024	4096x4096	498,78	248,57	122,73	165,25
4	1024	8192x8192	1858,15	933,45	481,48	546,17
5	2048	1024x1024	43,66	22,09	11,44	36,13
6	2048	2048x2048	284,93	142,14	70,66	94,83
7	2048	4096x4096	1090,21	542,92	271,56	311,44
8	2048	8192x8192	4088,69	2024,60	1045,54	1030,22
9	4096	1024x1024	90,19	44,70	22,94	71,71
10	4096	2048x2048	583,49	290,61	145,62	182,06
11	4096	4096x4096	2285,55	1141,42	573,51	619,09
12	4096	8192x8192	8864,88	4389,49	2238,59	2082,02
13	8192	1024x1024	181,17	90,04	46,03	139,25
14	8192	2048x2048	1179,47	587,06	292,63	360,30
15	8192	4096x4096	4679,22	2339,48	1182,45	1236,81
16	8192	8192x8192	18410,40	9169,29	4633,28	4198,71

Tabla AI.10. Tiempos de la solución paralela híbrida con Pthreads.

Escenario	Momento	Tamaño	2-2	2-4	2-8	4-8
1	1024	1024x1024	3,68	7,45	14,33	4,24
2	1024	2048x2048	2,26	4,50	9,13	6,00
3	1024	4096x4096	2,22	4,46	9,04	6,71
4	1024	8192x8192	2,28	4,55	8,81	7,77
5	2048	1024x1024	3,78	7,48	14,44	4,57
6	2048	2048x2048	2,27	4,54	9,14	6,81
7	2048	4096x4096	2,25	4,51	9,02	7,86
8	2048	8192x8192	2,25	4,55	8,81	8,94
9	4096	1024x1024	3,70	7,47	14,56	4,66
10	4096	2048x2048	2,27	4,55	9,08	7,27
11	4096	4096x4096	2,26	4,53	9,01	8,35
12	4096	8192x8192	2,26	4,56	8,94	9,61
13	8192	1024x1024	3,72	7,48	14,63	4,84
14	8192	2048x2048	2,27	4,57	9,16	7,44
15	8192	4096x4096	2,27	4,54	8,98	8,58
16	8192	8192x8192	2,27	4,56	9,02	9,96

Tabla AI.11. Speedup logrado en la solución paralela híbrida con Pthreads.

Escenario	Momento	Tamaño	2-2	2-4	2-8	4-8
1	1024	1024x1024	0,92	0,93	0,90	0,13
2	1024	2048x2048	0,56	0,56	0,57	0,19
3	1024	4096x4096	0,56	0,56	0,56	0,21
4	1024	8192x8192	0,57	0,57	0,55	0,24
5	2048	1024x1024	0,95	0,93	0,90	0,14
6	2048	2048x2048	0,57	0,57	0,57	0,21
7	2048	4096x4096	0,56	0,56	0,56	0,25
8	2048	8192x8192	0,56	0,57	0,55	0,28
9	4096	1024x1024	0,93	0,93	0,91	0,15
10	4096	2048x2048	0,57	0,57	0,57	0,23
11	4096	4096x4096	0,57	0,57	0,56	0,26
12	4096	8192x8192	0,56	0,57	0,56	0,30
13	8192	1024x1024	0,93	0,93	0,91	0,15
14	8192	2048x2048	0,57	0,57	0,57	0,23
15	8192	4096x4096	0,57	0,57	0,56	0,27
16	8192	8192x8192	0,57	0,57	0,56	0,31

Tabla AI.12. Eficiencia lograda en la solución paralela híbrida con Pthreads.

Resultados Obtenidos de la solución híbrida con OpenMP

En la Tabla AI.13., AI.14. y AI.15. se muestran los tiempos de ejecución, speedup y eficiencia respectivamente de la solución híbrida con OpenMP (MPI+OpenMP).

Escenario	Momento	Tamaño	2-2	2-4	2-8	4-8
1	1024	1024x1024	31,45	15,92	8,48	4,86
2	1024	2048x2048	123,08	62,48	32,12	17,36
3	1024	4096x4096	467,59	233,86	119,82	64,93
4	1024	8192x8192	1791,78	923,67	465,99	243,95
5	2048	1024x1024	63,88	32,02	16,78	9,71
6	2048	2048x2048	251,06	126,89	64,72	34,36
7	2048	4096x4096	992,22	481,51	243,80	128,54
8	2048	8192x8192	3747,90	1907,12	955,91	489,55
9	4096	1024x1024	127,93	64,18	33,35	18,59
10	4096	2048x2048	512,67	263,75	131,79	67,77
11	4096	4096x4096	1998,49	990,77	498,47	257,61
12	4096	8192x8192	7807,75	3940,43	1971,57	987,15
13	8192	1024x1024	258,59	130,61	67,26	36,90
14	8192	2048x2048	1019,69	513,48	261,05	135,46
15	8192	4096x4096	4046,11	2020,58	1011,76	517,89
16	8192	8192x8192	15967,70	8016,77	4015,06	2018,84

Tabla AI.13. Tiempos de la solución paralela híbrida con OpenMP.

Escenario	Momento	Tamaño	2-2	2-4	2-8	4-8
1	1024	1024x1024	2,55	5,04	9,46	16,49
2	1024	2048x2048	2,49	4,91	9,54	17,65
3	1024	4096x4096	2,37	4,74	9,26	17,08
4	1024	8192x8192	2,37	4,59	9,11	17,39
5	2048	1024x1024	2,59	5,16	9,84	17,01
6	2048	2048x2048	2,57	5,09	9,97	18,79
7	2048	4096x4096	2,47	5,09	10,04	19,05
8	2048	8192x8192	2,46	4,83	9,64	18,82
9	4096	1024x1024	2,61	5,20	10,02	17,97
10	4096	2048x2048	2,58	5,02	10,04	19,52
11	4096	4096x4096	2,59	5,22	10,37	20,07
12	4096	8192x8192	2,56	5,08	10,15	20,26
13	8192	1024x1024	2,60	5,16	10,01	18,25
14	8192	2048x2048	2,63	5,22	10,27	19,79
15	8192	4096x4096	2,62	5,25	10,49	20,50
16	8192	8192x8192	2,62	5,21	10,41	20,71

Tabla AI.14. Speedup logrado en la solución paralela híbrida con OpenMP.

Escenario	Momento	Tamaño	2-2	2-4	2-8	4-8
1	1024	1024x1024	0,64	0,63	0,59	0,52
2	1024	2048x2048	0,62	0,61	0,60	0,55
3	1024	4096x4096	0,59	0,59	0,58	0,53
4	1024	8192x8192	0,59	0,57	0,57	0,54
5	2048	1024x1024	0,65	0,64	0,62	0,53
6	2048	2048x2048	0,64	0,64	0,62	0,59
7	2048	4096x4096	0,62	0,64	0,63	0,60
8	2048	8192x8192	0,61	0,60	0,60	0,59
9	4096	1024x1024	0,65	0,65	0,63	0,56
10	4096	2048x2048	0,65	0,63	0,63	0,61
11	4096	4096x4096	0,65	0,65	0,65	0,63
12	4096	8192x8192	0,64	0,63	0,63	0,63
13	8192	1024x1024	0,65	0,64	0,63	0,57
14	8192	2048x2048	0,66	0,65	0,64	0,62
15	8192	4096x4096	0,66	0,66	0,66	0,64
16	8192	8192x8192	0,65	0,65	0,65	0,65

Tabla AI.15. Eficiencia lograda en la solución paralela híbrida con OpenMP.