UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: "Análisis de estrategias de distribución dinámica de trabajo, en el paradigma master worker sobre un cluster de multicore"
Autores: Torres, Rocío Nahime y Pantaleo, Facundo Adrián
Directores: De Giusti, Laura, y Chichizola, Franco
Carrera: Licenciatura en Informática

Resumen

Como objetivo de la tesina, se encuentra comparar el rendimiento del paradigma Master-Worker con distribución dinámica de trabajo usando uno y dos niveles de master sobre cluster de multicores, empleando diferentes modelos de comunicación (Pasaje de Mensajes e Híbrido). Para esto se utilizaron dos aplicaciones donde la carga de trabajo es variable en función de características de los datos y con alta complejidad computacional. Una de ellas es el clásico problema "N-reinas", y la otra es "Búsqueda de máxima similitud en Bases de Datos de secuencias de ADN". Ambas se diferencian entre sí por el tamaño de los datos con los que deben trabajar, y por consiguiente el tamaño de las comunicaciones en las soluciones paralelas. Para ambas aplicaciones, se analizará el rendimiento de la solución Master-Worker con un nivel de master, usando el modelo de comunicación pasaje de mensajes, en una instancia posterior, se analizará el rendimiento de las soluciones Master-Worker con dos niveles de master, tanto con el modelo de comunicación pasaje de mensajes, como con un modelo híbrido. Finalmente, sobre las pruebas experimentales, se aplicarán métricas de performance, en base a las cuales, se podrán comparar las distintas soluciones, para obtener las conclusiones.

Palabras Claves

N-reinas, Similitud de secuencias de ADN, cluster multicore, paradigma Master-Worker, modelo comunicación por pasaje de mensajes, modelo de comunicación híbrida,

Conclusiones

Finalizado el trabajo experimental, se analizaron y compararon los comportamientos de las soluciones implementadas. Puede verse que el algoritmo Master-Worker clásico de un nivel obtiene el mejor rendimiento para los dos problemas analizados. Esto puede deberse al hecho de que esta solución aprovecha mejor la arquitectura, y en el tamaño que esta tiene, el master no llega a convertirse en un cuello de botella ante los pedidos de todos los workers.

Trabajos Realizados

En este trabajo se realizó un repaso de la historia del procesamiento paralelo y se estudiaron los conceptos esenciales para poder modelar, desarrollar, analizar y evaluar diferentes alternativas de implementación de sistemas paralelos, en particular para los problemas de "N-reinas" y "Búsqueda de máxima similitud en Bases de Datos de secuencias de ADN", empleando el paradigma Master-Worker y los modelos de comunicación pasaje de mensajes y memoria compartida.

Trabajos Futuros

Resulta particularmente interesante estudiar el comportamiento de ambas aplicaciones en arquitecturas de mayor tamaño para poder analizar la escalabilidad de las distintas soluciones implementadas. Además se podría estudiar estas soluciones sobre una arquitectura heterogénea, para poder hacer un análisis del balance de carga de los workers.

Fecha de la presentación: Diciembre 2014

UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA EN INFORMÁTICA

Análisis de estrategias de distribución dinámica de trabajo, en el paradigma master worker sobre un cluster multicore

Alumnos: APU Torres, Rocío Nahime y APU Pantaleo, Facundo Adrián.

Directores: Dra. De Giusti, Laura y Esp. Chichizola Franco

Agradecimientos

A mis papás y a mi familia por apoyarme y ayudarme en estos años de carrera y durante toda mi vida, mis logros van dedicados a ustedes.

A mis amigos de facultad, sin ustedes no hubiera disfrutado tanto el trayecto y sin duda hubiera sido más difícil, especialmente gracias a Pía.

A mis amigas de la vida por escucharme y también acompañarme todos estos años.

A cada uno de los profesores de la carrera, de todos me llevo algo bueno y en especial de aquellos que me transmitieron no sólo conocimiento sino también motivación por lo que hacemos.

A Viviana H., por darme un lugar en su proyecto de extensión que me permitió desarrollar otra parte de mí la cual me hizo crecer como persona.

A Laura y Franco, nuestros directores, por su buena predisposición, paciencia y consejo, gracias por aceptar ser parte de esto.

A Facundo, mi novio y compañero de tesina, por su enorme paciencia y amor, porque este es el primero de muchos logros que vamos a alcanzar juntos.

"Lo maravilloso de aprender algo es que nadie puede arrebatárnoslo". B. B. King.

Nahime

Agradecimientos

A mis padres y a mi familia por transmitirme sus valores, principios, conocimientos, cariño, por haberme guiado, ayudado y formado siempre. Sin ustedes el camino hubiera sido mucho más difícil. Este logro está dedicado a ustedes.

A mis abuelas, por el cariño y apoyo que siempre me brindaron.

A mi tío por sus consejos, afecto y su apoyo.

A mis primos que estuvieron siempre.

A Carla y su familia. Por el cariño, apoyo y confianza al elegirme como padrino.

A mi gran amigo Martín y a su familia. Por abrirme siempre las puertas de su casa, tratarme como a un miembro más y estar en todos los momentos junto a mí.

A mis amigos de la vida y de la facultad que me acompañaron e hicieron más liviano el camino.

A Fruss, Pablo, Martín y otros compañeros de trabajo que me dieron una mano cuando estuve complicado entre el mundo laboral y el académico.

A los directores Laura y Franco, por su confianza al embarcarse en este proyecto, todos los consejos, ayuda y acompañamiento que nos brindaron durante el mismo.

A la facultad por la formación brindada y al departamento de postgrado por prestarnos sus instalaciones para realizar las pruebas de este trabajo.

A mi novia y compañera de tesina Nahime por su confianza y amor. Por su hacerme partícipe de este proyecto y de otro mucho más grande, por haberme acompañado en otra conquista más de las que están por venir.

"Locura es hacer lo mismo una y otra vez esperando resultados diferentes." Albert Einstein.

Facundo

Contenido

1. INTRODUCCIÓN 1

1.1 Procesamiento Paralelo 1
1.1.1 Concurrencia y paralelismo 1
1.1.2 Cómputo distribuido y paralelismo 2

1.2 Ventajas del procesamiento paralelo 2
1.2.1 Resolver problemas más grandes 2
1.2.2 Resolver problemas con límite de tiempo 3
1.2.3 Resolver problemas con mayor precisión 3
1.2.4 Límites en el cómputo serial 3

1.3 Limitaciones del procesamiento paralelo 3

1.4 Límites en el sistema de memoria 4
1.4.1 Mejorando la latencia de memoria mediante el uso de chaches 4
1.4.2 Impacto del ancho de banda 4

1.5 Definiciones y conceptos básicos 5

2. ARQUITECTURAS PARALELAS 7

2.1 Memoria compartida 7
2.1.1 Multicores 8
2.1.2 Modelo de comunicación 9
2.1.2.1 Hilos 9
2.1.2.2 Pthread 10

2.2 Memoria distribuida 11
2.2.1 Cluster 11
2.2.2 Modelo de comunicación 12
2.2.2.1 Operaciones send y receive 12
2.2.2.2 Costo de la comunicación 14
2.2.2.3 MPI 14

2.3 Memoria compartida distribuida 15
2.3.1 Cluster de multicores 16
2.3.2 Modelo de comunicación 17

3. DISEÑO DE APLICACIONES 18

3.1 Etapas de diseño 18
3.1.1 Etapa de particionamiento 19
3.1.2 Etapa de comunicación 29
3.1.3 Etapa de aglomeración 30
3.1.3.1 Incrementando la granularidad 30
3.1.4 Etapa de mapeo 30
3.1.4.1 Mapeo estático 31
3.1.4.1.1 Mapeos basados en el particionamiento de datos 32
3.1.4.2 Mapeo dinámico 32
3.1.4.2.1 Esquema centralizado 33
3.1.4.2.2 Esquema distribuido 33

3.2 Paradigmas de programación 34
3.2.1 Paradigma Master-Worker 35
3.2.1.1 Modelo Uno 37

3.2.1.2 Modelo Dos 38
3.2.1.3 Modelo Tres 39

4. EVALUACIÓN DE SISTEMAS PARALELOS 40

4.1 Fuentes de overhead 40
4.1.1 Interacción entre procesos 41
4.1.2 Ocio de los procesadores 41
4.1.3 Cómputo extra asociado a la paralelización 42

4.2 Métricas 42
4.2.1 Tiempo de ejecución 42
4.2.2 Desbalance de carga 42
4.2.3 Speedup 43
4.2.4 Eficiencia 44
4.2.5 Escalabilidad 45

5. PROBLEMA "N-REINAS" 47

5.1 Origen del problema 47

5.2 Descripción del problema 47

5.3 Algoritmo secuencial 49

5.4 Algoritmo paralelo 50
5.4.1 Solución con pasaje de mensajes 51
5.4.1.1 Un nivel de master 51
5.4.1.2 Dos niveles de master 52
5.4.2 Solución Hibrida 53

6. PROBLEMA "BÚSQUEDA DE SIMILITUD MÁXIMA EN SECUENCIAS DE ADN" 55

6.1 Bioinformática 55
6.1.1 ¿Qué es? 55
6.1.2 ¿Porque es importante? 55

6.2 Descripción del problema 55
6.2.1 Conceptos preliminares 55
6.2.2 Alineación de secuencias 56
6.2.3 Subsecuencias 58
6.2.4 Similitud local y global 59
6.2.5 Algoritmo Smith-Waterman 59

6.3 Algoritmo secuencial 61

6.4 Algoritmo paralelo 63
6.4.1 Solución con pasaje de mensajes 64
6.4.1.1 Un nivel de master 64
6.4.1.2 Dos niveles de master 65
6.4.2 Solución Hibrida 68

7. EXPERIMENTACIÓN 70

7.1 Arquitectura utilizada 70

7.2 N-reinas 70
7.2.1 Comparación de las tres soluciones 71
7.2.2 Análisis del comportamiento cuando crece la arquirectura 72

7.3 Búsqueda de similitud máxima entre secuencias de ADN 73
7.3.1 Comparación de las tres soluciones 73
7.3.2 Análisis del comportamiento cuando crece la arquirectura 74

8. CONCLUSIONES 76

9. APÉNDICE A - DETALLE DE LOS RESULTADOS PARA "N-REINAS" 78

10. APÉNDICE B-DETALLE DE LOS RESULTADOS PARA "BÚSQUEDA DE SIMILITUD MÁXIMA EN SECUENCIAS DE ADN" 86

11. APÉNDICE C - CÓDIGOS DEL PROBLEMA "N-REINAS" 109

12. APÉNDICE D – CÓDIGOS DEL PROBLEMA "BÚSQUEDA DE SIMILITUD MÁXIMA EN SECUENCIAS DE ADN" 128

13. ANEXO A – PHTREADS 145

14. ANEXO B – MPI 147

15. BIBLIOGRAFÍA 151

Capítulo 1

Introducción

Una computadora tradicional tiene una única unidad de procesamiento para ejecutar las instrucciones de un programa determinado. Una forma de incrementar el poder de cómputo es mediante el uso de múltiples unidades de procesamiento dentro de una única computadora, por otro lado se pueden usar múltiples computadoras trabajando en conjunto para resolver el problema. En ambos casos el problema es dividido en partes donde cada una es asignada a una unidad de procesamiento y resuelto por la misma en forma paralela. El desarrollo de programas usando este tipo de recursos computacionales es conocido como programación paralela.

Una computadora paralela es un conjunto de unidades de procesamiento que pueden trabajar de forma cooperativa para resolver un problema computacional. Esta definición es lo suficientemente amplia para incluir supercomputadoras paralelas que tienen cientos o miles de unidades de procesamiento, redes de computadoras, computadoras con múltiples procesadores y sistemas embebidos. Las computadoras paralelas son interesantes porque ofrecen el potencial de concentrar recursos computacionales como procesadores, memoria o ancho de banda de entrada/salida y enfocarlos para resolver importantes problemas computacionales.

El paralelismo ha sido visto en ocasiones como un área rara y exótica de la informática. Sin embargo, un estudio de las tendencias en el desarrollo de aplicaciones, las arquitecturas de computadoras y redes muestra que esta visión ya no es sostenible. El paralelismo se ha vuelto una parte fundamental en el desarrollo de aplicaciones informáticas. [Fos94]

1.1 Procesamiento Paralelo

Se define al procesamiento paralelo como el proceso mediante el cual se computa la solución de un problema de manera concurrente o simultánea sobre diferentes componentes físicos. El procesamiento del algoritmo paralelo para un problema dado se ejecuta de forma coordinada y descomponiéndolo en múltiples procesos que se ejecutan en varias unidades de procesamiento. Así, se busca realizar un procesamiento de la información de manera eficiente en el menor tiempo posible. [HB84]

Un sistema paralelo es la conjunción de un algoritmo paralelo con una máquina paralela. Como se mencionó anteriormente, el algoritmo paralelo se encuentra muy fuertemente ligado a la arquitectura donde se lo desarrolla, y es por esto que las optimizaciones y el rendimiento de la aplicación están íntimamente relacionados.

1.1.1 Concurrencia y paralelismo

Un programa concurrente comprende la ejecución de dos o más procesos que trabajan juntos para llevar adelante una tarea. Cada proceso es un programa secuencial, es decir, una secuencia de instrucciones que se ejecutan una después de la otra. Mientras que un programa secuencial tiene un único hilo de control, un programa concurrente tiene múltiples hilos de control. La concurrencia es un concepto de software, que implica la

determinación de la comunicación y sincronización entre los procesos en ejecución simultánea. [And00]

El paralelismo puede ser caracterizado como un caso particular de concurrencia, en el que los procesos que se ejecutan concurrentemente lo hacen sobre un conjunto de componentes físicos homogéneos o heterogéneos con un espacio de direcciones de memoria compartido o distribuido. [And00]

1.1.2 Cómputo distribuido y paralelismo

Aunque el procesamiento paralelo y el distribuido se encuentran relacionados, no se deben confundir los conceptos (a pesar de que muchos autores en la literatura lo hacen). El atributo distintivo de un sistema paralelo es que está escrito para resolver un problema en menos tiempo que el que le tomaría a un programa secuencial. En cambio, en un sistema distribuido las tareas que se ejecutan se encuentran en máquinas autónomas, y su fin es compartir información y recursos. El espacio de direcciones de memoria es distribuido y las aplicaciones que se ejecutan son de múltiples usuarios y dinámicas. [And00]

1.2 Ventajas del procesamiento paralelo

El uso del procesamiento paralelo provee un incremento significativo en la performance de un programa. Esta idea está basada en que *p* procesadores podrían proveer hasta p veces más el poder de cómputo que un sólo procesador, sin importar la velocidad del procesador, con el objetivo de que se encuentre la solución al problema en un 1/p del tiempo que tomaría de la forma convencional. Por supuesto que esto es una situación ideal que raramente se consigue en la práctica. En general los problemas no se pueden dividir de forma perfecta en partes independientes, además es necesaria la interacción entre las partes para la transferencia de datos y la sincronización del cómputo. Sin embargo, se puede alcanzar un incremento sustancial en la performance dependiendo del problema y su capacidad para ser paralelizable.

Lo que hace que la computación paralela se encuentre siempre vigente son los continuos desarrollos y mejorías en la velocidad de ejecución de los procesadores haciendo a las computadoras paralelas cada vez más rápidas. Además, siempre habrá problemas pertenecientes a la familia "Grand Challenge" los cuales no pueden ser resueltos en una cantidad de tiempo razonable con las computadoras estándares, debido a su tamaño o complejidad. Este tipo de problemas suelen requerir enormes cantidades de cálculos repetitivos sobre grandes cantidades de datos para obtener los resultados. [Bar05]

1.2.1 Resolver problemas más grandes

Además de obtener un incremento en la velocidad en que se consigue la resolución de un problema, los sistemas paralelos permiten frecuentemente resolver un problema de mayor tamaño en una cantidad razonable de tiempo. Por ejemplo, computar la predicción del clima involucra dividir el aire en una grilla tridimensional donde cada celda contiene una solución y el uso de múltiples computadoras o unidades de procesamiento permite que más celdas con soluciones en la grilla sean computadas en una franja de tiempo dada, obteniendo así una solución más precisa.

1.2.2 Resolver problemas con límite de tiempo

Algunos problemas tienen un tiempo límite muy específico en sus computaciones como por ejemplo la predicción del clima. Emplear dos días para calcular el clima local en forma precisa para el día siguiente haría que la predicción no tenga utilidad práctica.

1.2.3 Resolver problemas con mayor precisión

Los sistemas paralelos permiten también obtener resultados más precisos que los experimentos prácticos o pueden ser realizados con un menor esfuerzo económico. Un ejemplo es el uso de simulaciones para determinar la resistencia al aire de un vehículo. Comparando la simulación con un experimento clásico de túnel de viento, una simulación puede otorgar resultados más precisos dado que el movimiento relativo del vehículo en relación al suelo puede ser incluido en la simulación. Esto no es posible en el túnel de viento, ya que el vehículo no se puede desplazar. La simulación de colisiones entre vehículos es un ejemplo obvio donde se pueden obtener resultados con menor esfuerzo económico. [Rau10]

1.2.4 Límites en el cómputo serial

Existen razones físicas y prácticas que imponen límites significativos al hecho de desarrollar solamente computadoras seriales más rápidas. Entre estas razones se encuentran:

- Velocidades de transmisión. La velocidad de una computadora serial depende directamente de que tan rápido se pueden transmitir los datos a través del hardware. Como límites absolutos se encuentran la velocidad de la luz (30 cm/nanosegundo) y el límite de transmisión de los cables de cobre (9 cm/nanosegundo). Indefectiblemente el incremento de la velocidad tiene que estar acompañado de un incremento en la proximidad de los elementos de procesamiento.
- Límites a la miniaturización. La tecnología de los procesadores está permitiendo un incremento en el número de transistores que se pueden integrar en un chip. Sin embargo, incluso con componentes a nivel molecular o atómico, se alcanzará un límite en el tamaño mínimo de los componentes.
- Límites económicos. Cada vez es más difícil hacer que un sólo procesador sea más rápido. Usar un gran número de procesadores moderadamente rápidos para alcanzar la misma (o inclusive mejor) performance es más barato.
- Durante los últimos 10 años las tendencias marcadas por redes cada vez más rápidas, sistemas distribuidos y arquitecturas de multiprocesadores (inclusive al nivel de las computadoras personales) muestran que el futuro del procesamiento se encuentra en el desarrollo del paralelismo. [Bar10]

1.3 Limitaciones del procesamiento paralelo

Así como los sistemas paralelos brindan diversas ventajas, también tienen varias limitaciones a considerar:

- Los procesadores requieren realizar comunicaciones entre sí.
- Los procesos deben esperarse entre sí para poder continuar con la ejecución de la aplicación (sincronización).

- El máximo grado de concurrencia alcanzable se encuentra determinado por el problema concreto a paralelizar.
- Alto grado de dependencia con el hardware subyacente.
- Los métodos de debugging del procesamiento secuencial no sirven en el procesamiento paralelo y la codificación requiere un mayor grado de dificultad.
- El desarrollador de algoritmos paralelos debe tener en cuenta muchos más factores que en un algoritmo secuencial, tales como no determinismo, sincronización, comunicación, división y distribución de los datos, balance de carga, tolerancia a fallos, manejo de memoria compartida y distribuida y deadlock entre otros.

1.4 Límites en el sistema de memoria

El rendimiento real de un programa en una computadora no se encuentra solamente en la velocidad del procesador sino también en la capacidad del sistema de memoria para "alimentar" con datos al procesador. A nivel lógico, un sistema de memoria que podría incluir múltiples niveles de cache, recibe una petición para una palabra de memoria y devuelve un bloque de datos de tamaño *b* que contiene la palabra solicitada en un lapso de *L* nanosegundos; dicho lapso es conocido como latencia de la memoria. La tasa a la que los datos pueden ser llevados desde la memoria al procesador determina el ancho de banda del sistema de memoria. [Ana03]

1.4.1 Mejorando la latencia de memoria mediante el uso de chaches

El manejo de la diferencia de velocidad entre los procesadores y las memorias DRAM ha dado lugar a numerosas innovaciones en los diseños de sistemas de memoria. Una de esas innovaciones consiste en ubicar una memoria más pequeña y más rápida entre el procesador y la memoria DRAM. Esta memoria que se conoce con el nombre de cache, actúa como un almacenamiento de baja latencia y alto ancho de banda. Los datos que necesita el procesador primero se colocan en la cache y todos los accesos posteriores a datos que se encuentran en la cache son proporcionados por la misma.

Así, en principio, si un dato se accede varias veces, la latencia de memoria de este sistema se reduce por el uso de la cache. Todos los datos referenciados de forma exitosa por la cache se llaman porcentaje de éxito (hit ratio). La velocidad de ejecución de varias aplicaciones está limitada por la velocidad en la que los datos pueden ser entregados a la CPU y no por la velocidad de esta; este tipo de aplicaciones se conoce como limitadas por memoria (memory bound). El rendimiento en estas aplicaciones se encuentra impactado en forma crítica por el porcentaje de éxito de la cache.

1.4.2 Impacto del ancho de banda

El ancho de banda de memoria hace referencia a la tasa en la que se pueden mover los datos entre el procesador y la memoria. Se encuentra determinada por el ancho de banda del bus de memoria y las unidades de memoria. Una técnica muy usada para mejorar el ancho de banda de memoria es incrementar el tamaño de los bloques de memoria. A modo de ejemplo, se asume que un solo requerimiento a memoria retorna un bloque contiguo de cuatro palabras. Esa unidad de cuatro palabras se conoce en este caso como línea de cache (cache line). Las computadoras convencionales pueden traer entre dos y ocho palabras juntas a la cache.

Al incrementar el tamaño del bloque la unidad de procesamiento dispone de más datos al realizar un pedido de memoria, es decir de mayor ancho de banda, sin aumentar la latencia de memoria del sistema. Este incremento permite acelerar la ejecución de las aplicaciones que no reutilizan datos.

De lo anterior se entiende que a mayor ancho de banda se obtienen mayores tasas en el cómputo. Esto se debe a que se asume que la disposición de los datos en memoria es tal que instrucciones sucesivas usan palabras sucesivas en memoria. En otras palabras, desde el punto de vista del cómputo existe localidad espacial en el acceso a los datos. En cambio, desde un enfoque centrado en la distribución de los datos, el cómputo se ordena de modo que sucesivas operaciones requieran datos contiguos. Si las operaciones o el patrón de acceso a los datos no tienen localidad espacial, es muy probable que el ancho de banda del sistema sea pobre.

Mientras que la velocidad de las unidades de procesamiento ha crecido significativamente durante las últimas décadas, la latencia de memoria y el ancho de banda no han acompañado este crecimiento. Se presentaron anteriormente técnicas para suplir estas diferencias, reduciendo la latencia e incrementando el ancho de banda, de las cuales se pueden extraer los siguientes conceptos:

- Explotar la localidad espacial y temporal en las aplicaciones es crítico para amortizar la latencia de memoria e incrementar el ancho de banda efectivo.

- Ciertas aplicaciones tienen inherentemente mayor localidad temporal que otras y por esto mayor tolerancia a un ancho de banda bajo. La tasa entre la cantidad de operaciones y la cantidad de accesos a memoria es un buen indicador de la tolerancia anticipada al ancho de banda de memoria.

- La disposición en memoria y la organización apropiada de las operaciones puede tener un impacto significativo en la localidad espacial y temporal. [Ana03]

1.5 Definiciones y conceptos básicos

Al desarrollar un algoritmo paralelo para un problema dado se tienen que descomponer las computaciones en varias partes llamadas tareas, las cuales se intenta que puedan ser computadas en paralelo en los cores de una unidad de procesamiento.

Las tareas de una aplicación suelen estar codificadas en un lenguaje de programación paralelo y son asignadas a procesos o hilos los cuales luego son asignados a unidades de procesamiento físicas para su ejecución.

Las técnicas de descomposición de tareas y las de mapeo de dichas tareas a los procesadores son aspectos que tienen que ver con el diseño de los algoritmos paralelos, y dicho tema se abarca en el Capítulo 3.

A su vez, se puede clasificar a las computadoras paralelas de acuerdo a la organización de su memoria en computadoras de memoria compartida y computadoras de memoria distribuida. Frecuentemente el término hilo se asocia con memoria compartida y el término proceso con memoria distribuida. Para las computadoras de memoria compartida, una memoria global compartida guarda los datos de una aplicación y puede ser accedida por todas las unidades de procesamiento o cores del hardware.

El intercambio de información entre los hilos es realizado mediante variables compartidas que son escritas por uno y leídas por otro. El comportamiento correcto del programa se consigue mediante la sincronización entre los hilos de forma tal que el acceso a los datos compartidos sea coordinado entre ellos, es decir, un hilo no lee un dato hasta que una operación de escritura haya finalizado. Dependiendo del lenguaje de programación o del ambiente utilizado, la sincronización la realiza el sistema operativo o se encuentra a cargo del programador. En el caso de las máquinas con memoria distribuida, existe una memoria privada a cada unidad de procesamiento, que solamente puede ser accedida por ella misma y no se necesita sincronización en el acceso a memoria. El intercambio de datos se realiza mediante el envío de datos de una unidad de procesamiento a otra mediante una red de interconexión con operaciones de comunicación explícitas. [Rau10]

Los diferentes tipos de computadora tanto las que poseen memoria compartida como memoria distribuida, y los modelos de comunicación utilizados en cada una de ellas se pueden ver con mayor detalle en el Capítulo 2.

Como se menciona anteriormente uno de los principales objetivos de los sistemas paralelos es reducir el tiempo de ejecución de las aplicaciones o resolver problemas de mayor tamaño en un tiempo razonable. La performance de un sistema depende de muchos factores, tales como la arquitectura que se utiliza, el compilador, el sistema operativo, entre otros. Es decir que el tiempo de ejecución depende tanto de la arquitectura, como del modelo utilizado para desarrollar el algoritmo y es por eso que resulta interesante tener métricas que permitan evaluar los sistemas paralelos, más allá del tiempo de ejecución. Dichas métricas y otros temas acerca de la performance de los sistemas paralelos se abordan en el Capítulo 4.

Capítulo 2

Arquitecturas paralelas

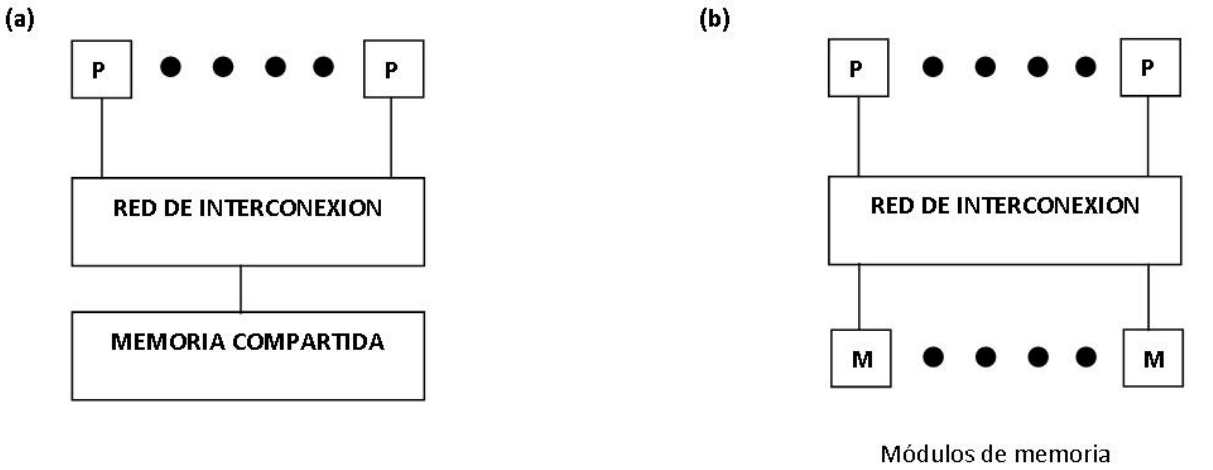
Una arquitectura paralela es una colección de elementos de procesamiento que se comunican y cooperan. Las aplicaciones paralelas aprovechan esto con el objetivo de resolver un problema común, tratando de reducir el tiempo de ejecución de la aplicación así también como para resolver problemas de mayor tamaño aprovechando las características de la arquitectura utilizada. A continuación se puede ver una clasificación de las arquitecturas paralelas, basada en su espacio de direcciones.

2.1 Memoria compartida

Desde el punto de vista de las plataformas paralelas, la memoria compartida es un espacio de datos común, accesible por todas las unidades de procesamiento, las cuales interactúan entre sí modificando los datos allí almacenados. Las plataformas de memoria compartida son atractivas para los programadores debido a esta posibilidad de compartir información entre los procesadores.

Las plataformas con memoria compartida, están formadas por una cantidad de procesadores, una memoria compartida físicamente y una red de interconexión entre los procesadores. Este esquema de memoria puede implementarse cómo un conjunto de módulos de memoria, donde cada procesador tiene acceso a cualquier módulo. Se suele utilizar un espacio de direcciones simple, lo que significa que cada posición en el sistema de memoria tiene una dirección única (que no se repetirá en otro módulo), la cual será usada por cada procesador para acceder a esa posición. [Bar05]

En la Figura 2.1 [Ana04] se puede ver el esquema de memoria compartida:



En (a) se observa el esquema de memoria compartida desde un punto abstracto y en (b) se ve la implementación con módulos de memoria.

Figura 2.1

Si el tiempo que toma acceder a cualquier dirección de memoria es idéntico para todas las unidades de procesamiento, entonces la plataforma es clasificada como acceso de memoria uniforme (UMA), por el contrario, si el tiempo que lleva acceder a ciertas

posiciones de memoria es mayor que a otras, la plataforma es clasificada como acceso de memoria no uniforme (NUMA). Lo ideal sería que el sistema provea acceso uniforme de memoria, pero en la práctica, en sistemas medianamente grandes, es difícil implementar hardware para tener un acceso rápido y uniforme a toda la memoria compartida para todas las unidades de procesamiento. Es por eso que muchos sistemas de memoria compartida tienen estructuras de memoria jerárquicas, así las unidades de procesamiento pueden acceder a las posiciones de memoria que están físicamente más cerca de ellos más rápido que a posiciones de memoria que están más distantes. [Bar05], [Ana03]

Las máquinas convencionales con un solo procesador, tienen una caché rápida, en la cual mantienen copias de los datos recientemente referenciados, esto reduce el acceso a la memoria principal. Frecuentemente se encuentran con dos niveles de memoria caché, los cuales se ubican entre el procesador y la memoria principal. La memoria caché se extiende a multiprocesadores de memoria compartida, dándole a cada procesador su propia memoria caché. Memorias caché rápidas en cada procesador pueden aliviar, de cierta forma, el hecho de que acceder a la memoria principal tome diferentes cantidades de tiempo para cada procesador; pero se debe estar seguro que las copias del mismo dato en distintas cachés sean idénticas para no tener problemas de consistencia de datos. Esto introduce un nuevo problema a tratar. Básicamente cuando un procesador modifica un dato en su caché, se requerirá que se tomen las medidas correspondientes para que las otras cachés se enteren de esta modificación y se actualicen, o invaliden el dato. Tanto la idea de propagar la actualización como invalidar el dato en las otras caches puede hacerse por hardware lo que lo hace completamente transparente al software o se puede utilizar una solución por software con un hardware específico que ayude a la tarea.

Es importante agregar que la transferencia de datos a las memorias caché se realiza por conjunto de palabras; lo cual presenta un nuevo problema, conocido como “falso sharing”. Este problema ocurre cuando dos o más procesadores comparten una línea de cache, y por ejemplo uno de ellos modifica un dato de dicha línea, que no era utilizado por otros procesadores, sin embargo, toda la línea de caché será tratada como inconsistente ejecutando el correspondiente protocolo de actualización o invalidación de datos. Estas invalidaciones y actualizaciones de líneas de cache hacen que todo el sistema se ejecute más lento, introduciendo un overhead con respecto a los tiempos de ejecución. La coherencia de caché no será vista con detalle en este documento, por lo cual se deja una referencia a quién le interese profundizar: [Fer98].

2.1.1 Multicores

En la década de los noventas y aproximadamente hasta el 2004, las arquitecturas de computadoras han buscado, sobre todo, una mayor velocidad del procesador, más memoria caché (dentro del procesador), más memoria principal y una mayor velocidad en la comunicación entre la memoria y el procesador. La Ley de Moore, publicada en 1995 [Gor98], y calibrada en 1975 por su autor, afirma que el número de transistores en un circuito integrado se duplica aproximadamente cada dos años. Esta ley, se mantiene técnicamente válida, la cantidad de transistores que pueden ponerse en un chip, continúa duplicándose cada dos años, pero la ventaja de hacerlo, alcanzó su punto máximo, y empezó a decaer, debido al calor que generan tantos transistores en un único chip, y al consumo que estos ocasionan. Por esta razón, en lugar de incrementar la velocidad de los procesadores, aumentando la frecuencia de estos, los desarrolladores de hardware han optado por incrementar el número de núcleos de procesamiento en cada chip, dando lugar a lo que se conoce como procesadores “multicores”.

Con este nuevo modelo, se requiere que la jerarquía de memoria pueda servir peticiones a más de un núcleo, por lo cual se agrega mayor complejidad, por ejemplo puede haber una memoria cache para cada uno de los núcleos y otra compartida entre ellos.

Mientras que aumentar la velocidad del procesador repercute de forma prácticamente directa en la disminución del tiempo de ejecución de un algoritmo, para tomar ventaja de los núcleos adicionales el algoritmo debe ser escrito de manera tal que exprese el uso de los núcleos de forma paralela, y para hacerlo se deben utilizar herramientas que lo permitan. [Eli13]

Ya desde la segunda mitad de la década del 2000, los procesadores con más de un núcleo se consideraron estándares y a medida que pasa el tiempo va a ser cada vez menos probable adquirir una computadora de escritorio con un único núcleo de procesamiento. Tal como ha sido la evolución del mercado, las computadoras de escritorio con más de un núcleo de procesamiento pasarán a ser las de menor costo (o de hecho las únicas) disponibles, y el costo variará en función de otras características, (tales como la frecuencia de reloj, el tamaño de memoria principal, los periféricos instalados, etc.). [Fer08]

2.1.2 Modelo de comunicación

Una forma de programar aplicaciones paralelas es mediante la definición de tareas independientes (o módulos) que pueden ser ejecutadas al mismo tiempo y que son mapeadas a los procesadores para ser ejecutadas en paralelo, dicho mapeo puede ser explícitamente declarado por el programador o puede dejarse a criterio de la librería y el sistema operativo utilizados, para que realice el mapeo bajo demanda en tiempo de ejecución.

En plataformas con memoria compartida la comunicación entre tareas está dada de forma implícita leyendo y escribiendo datos en la memoria que es accesible por todos los procesadores. Es por esta razón que los paradigmas de programación sobre este tipo de plataformas tienen su énfasis en ver cómo expresar concurrencia, exclusión mutua y sincronización con técnicas que aprovechen al máximo la arquitectura y eviten el overhead que estos aspectos podrían incluir.

La exclusión mutua es usada para evitar que dos procesos accedan al mismo tiempo a la ejecución de una sección crítica, la cual a su vez es una porción de código de un programa en la cual se accede a un recurso compartido (estructura de datos o dispositivo) que no debe ser accedido por más de un proceso o hilo en ejecución. Para que los programas logren una mejor performance deberán reducir las secciones críticas al mínimo posible dado que las mismas hacen que la ejecución sea serial. [Bar05]

2.1.2.1 Hilos

Uno de los modelos que se utilizan en memoria compartida es el basado en procesos, los procesos son unidades de ejecución que asumen que toda la información asociada a estos es privada, al menos que se especifique lo contrario. Si bien esta propiedad es importante para asegurar protección de datos en sistemas multiusuarios, no es útil con tareas que cooperan para resolver un mismo problema. El overhead asociado a la protección de los datos de cada proceso hace que este modelo sea menos atractivo para plataformas de memoria compartida. En contraste con esto, el modelo de hilos asume que

toda la memoria es global, haciendo que la manipulación de los datos sea más rápida que con procesos. Un hilo es una unidad básica de utilización de la CPU, y consiste en un contador de programa, un juego de registros y un espacio de pila. Los hilos dentro de una misma aplicación comparten la sección de código, la sección de datos y los recursos del SO (archivos abiertos y señales). Un proceso tradicional es igual a una tarea con un solo hilo. [Ana03]

A continuación se presentan algunas ventajas del uso de hilos:

- Portabilidad de software. Las aplicaciones construidas a base de hilos, pueden ser ejecutadas tanto en plataformas paralelas como secuenciales, sin necesidad de efectuar ningún cambio.
- Ocultamiento de latencia. Una de las principales fuentes de overhead en los programas (tanto secuencial como paralelo) es la latencia producida por el acceso a memoria, por operaciones de entrada salida y por las comunicaciones. Permitiendo la ejecución de múltiples hilos, este overhead puede ser ocultado, así mientras que un hilo está esperando por una operación de comunicación, otros hilos pueden utilizar la CPU.
- Planificación y balance de carga. Cuando se escriben programas paralelos para plataformas con memoria compartida, se debe expresar la concurrencia de tal forma que se minimice el overhead generado por las interacciones y el tiempo ocioso. Usualmente resulta difícil obtener una distribución del trabajo balanceada en aplicaciones poco estructuradas y dinámicas. Los hilos permiten al programador especificar un gran número de tareas concurrentes que serán mapeadas a unidades de procesamiento de forma dinámica. De esta manera, se libera al programador de la responsabilidad de la planificación explícita y del balance de carga.
- Facilidad de programación y amplio uso. Escribir programas con manejo de hilos no representa mayores dificultades que otro tipo de programas. Sin embargo, lograr igual rendimiento para ambos programas puede requerir un esfuerzo mayor. [Ana03]

2.1.2.2 Pthread

Históricamente los desarrolladores de hardware han implementado su propia versión de hilos. Estas implementaciones difieren entre sí dificultando la tarea de los programadores de desarrollar aplicaciones con manejo de hilos que fueran portables. Surgió así la necesidad de un estándar que permita la codificación de aplicaciones con manejo de hilos, y fue la IEEE quién en 1995 especificó el estándar POSIX Section 1003.1c, conocido como Pthreads.

Pthreads es una librería que provee funciones para la creación y manejo de hilos, y que entre otras cosas permite lo siguiente:

- Manejo de hilos. Provee rutinas que trabajan directamente sobre los hilos, por ejemplo creación de hilos, espera entre hilos, así como funciones que permitan asignar o consultar por los atributos de un hilo.
- Declaración de sección crítica. Provee variables y funciones para aplicar sobre estas, que permiten la sincronización de secciones críticas, y son conocidas como "*mutex*", lo cual en inglés es la abreviatura de "mutual exclusion" (exclusión mutua).

- Variables de condición. Provee funciones para el manejo de comunicaciones entre hilos que comparten un *mutex*, tales como creación y destrucción, y las clásicas funciones wait y signal.
- Sincronización. Provee funciones para el manejo de sincronización y la generación de barreras entre hilos.

Para más información sobre las funciones que provee Pthreads, ver Anexo A-Phreads.

2.2 Memoria distribuida

Una plataforma de memoria distribuida está compuesta por un cierto número de elementos de procesamiento, llamados nodos, y una red de interconexión que conecta estos nodos y soporta la transferencia de datos entre ellos. Cada nodo es una unidad independiente, cuya memoria local es privada, es decir, no puede ser accedida desde otros nodos. Cuando un proceso necesita información almacenada en la memoria local de otros nodos para realizar algún cómputo, se ejecuta un pasaje de mensajes a través de la red de interconexión. [Rau10]

Una de las ventajas de las plataformas de memoria distribuida, es que se pueden escalar físicamente más fácilmente que las de memoria compartida. [Bar05]

En la figura 2.2 [Ele04] se puede ver un esquema de cómo podría ser una plataforma de memoria distribuida:

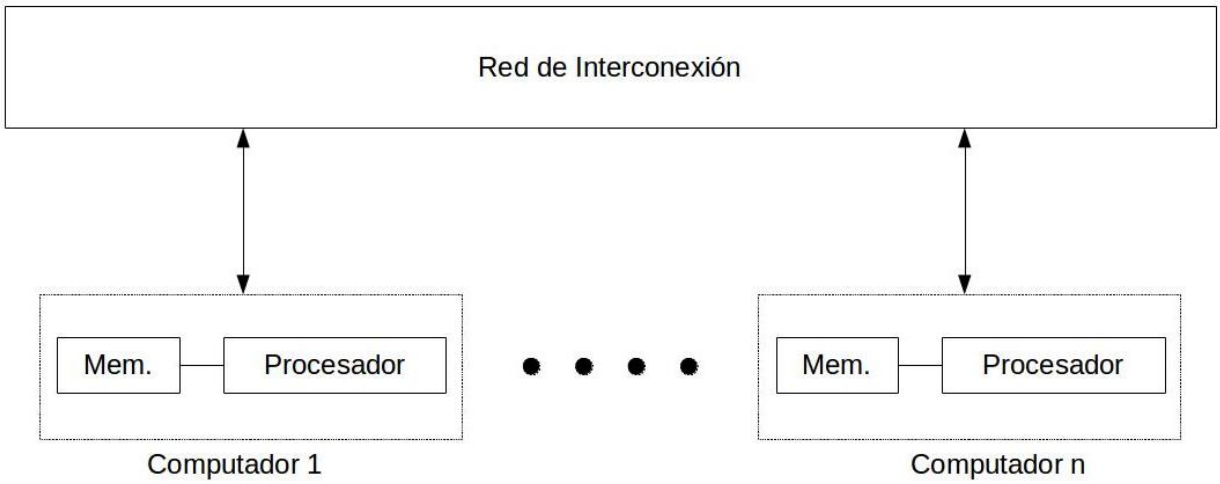


Figura 2.2

2.2.1 Cluster

Un cluster es un sistema de procesamiento paralelo o distribuido, que consiste en una colección de nodos autónomos interconectados, trabajando juntos, como un único recurso de procesamiento; donde cada nodo puede ser sistema mono/multiprocesador, con memoria, periféricos de E/S, y un sistema operativo. [Cla00]

Para que un cluster funcione como tal, no basta solo con conectar entre si los ordenadores, sino que es necesario proveer un sistema de manejo del cluster, el cual se encargue de interactuar con el usuario y los procesos que corren en él para optimizar el funcionamiento. [Lau08]

En muchos casos, los clusters están compuestos por procesadores de características similares conectados mediante una única arquitectura de red. Los tiempos de cómputo para todos los procesadores son iguales, así como también lo son los tiempos de comunicación entre ellos. A estos sistemas se los conoce como sistemas homogéneos. Sin embargo, debido a la arquitectura inherente a este tipo de sistema, es muy probable que al incrementar el número de procesadores en el cluster o al sustituir algún elemento de la máquina (procesador, memoria, interfaz de red, etc.) las características de la nueva máquina no coincidan con las características de los componentes originales, dando lugar a un sistema de naturaleza heterogénea. [Luz04]

Los clusters de PCs están jugando un papel importante en la Computación de Alto Rendimiento, esto es debido al bajo coste de estos sistemas en relación a su rendimiento. Si bien en principio los clusters se han aplicado y se aplican en lo que se denomina usualmente "cómputo científico", actualmente también se están utilizando satisfactoriamente en otras áreas por ejemplo servidores Web y comercio electrónico, bases de datos de alto rendimiento. [Fer04][Pau07]

2.2.2 Modelo de comunicación

En las plataformas de memoria distribuida la unidad de ejecución utilizada son los procesos y la interacción entre estos se realiza mediante el intercambio de mensajes. Es por eso que el modelo de comunicación en arquitecturas de memoria distribuida es conocido como “Pasaje de mensajes”.

Una máquina que soporta pasaje de mensajes consiste de *p* procesos, cada uno con su propio espacio de direcciones. De aquí surgen dos implicaciones, la primera es que los datos deben ser particionados y distribuidos explícitamente, y la segunda que todas las comunicaciones requieren la participación de dos procesos, el que requiere los datos y el que los tiene. Es tarea del programador descomponer el problema en tareas, especificar la concurrencia y la comunicación entre tareas así también como es el encargado de distribuir los datos. Si bien la programación con este modelo suele clasificarse como dificultosa, puede ofrecer un alto grado de performance y escalabilidad ya que el programador tiene gran control y flexibilidad para el manejo de los procesos.

2.2.2.1 Operaciones send y receive

Las operaciones básicas para el intercambio de mensajes entre procesos son el envío de mensajes (send) y la recepción de mensajes (receive). Existen distintos protocolos para estas operaciones los cuales se muestran a continuación:

- Bloqueante: cuando las sentencias se ejecutan en modo bloqueante, las rutinas devuelven el control una vez que el dato que se transmite está seguro. Este modelo puede ocasionar ociosidad en los procesadores. Existen dos posibilidades para este tipo de mensajes:
    - Sin buffer: en este caso tanto las rutinas *send* y *receive* devuelven el control sólo cuando el mensaje ha sido transmitido en su totalidad. Este modelo puede provocar ociosidad en los procesos, que se quedan esperando a que ambos estén listos para la transmisión y hasta su finalización.

- Con buffer: en este caso los procesos involucrados en la comunicación tienen un buffer para alojar los mensajes hasta que estos sean efectivamente transmitidos. En cuanto al proceso emisor, espera a que el mensaje sea copiado en el buffer y luego continúa su ejecución; por otro lado una vez que el proceso receptor está listo para ejecutar la operación *receive* se fija si el mensaje está en el buffer correspondiente para poder copiarlo desde ahí.

En la figura 2.3 [Ana03] se puede ver cómo se comportan los procesos con este modelo (a) sin buffer y (b) con buffer.

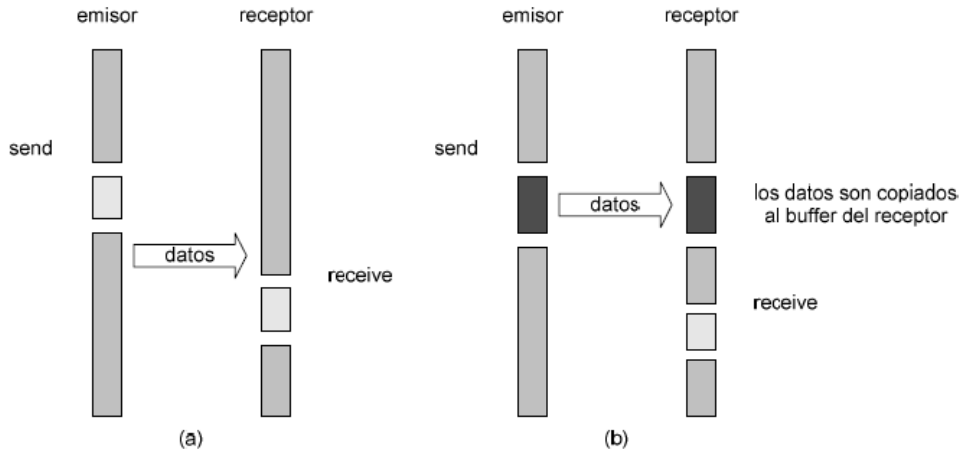


Figura 2.3

- No bloqueante: este término es usado para describir rutinas que devuelven el control sin importar si el mensaje ha sido recibido o no, con el fin de evitar la ociosidad de los procesadores ocasionada por la espera del mensaje. Sin embargo implica realizar un chequeo posterior para asegurar la finalización de la comunicación. También existen dos posibilidades:

  - Sin buffer: el proceso emisor simplemente le deja un mensaje pendiente al proceso receptor y continúa su trabajo. Cuando el proceso receptor alcanza la sentencia receive correspondiente, se inicia la transferencia.

  - Con buffer: se utiliza ADM (acceso directo a memoria) para copiar los datos a enviar en un buffer. Mientras la operación de copiado se está llevando a cabo el proceso emisor puede continuar con su trabajo pero hasta el envío los datos no deberían ser modificados. Cuando el proceso receptor está listo y el dato está en el buffer del emisor, puede iniciar la transferencia a su buffer local. [Ana03]

En la figura 2.4 [Ana03] se puede ver cómo se comportan los procesos con este modelo *(a)* sin buffer y *(b)* con buffer.

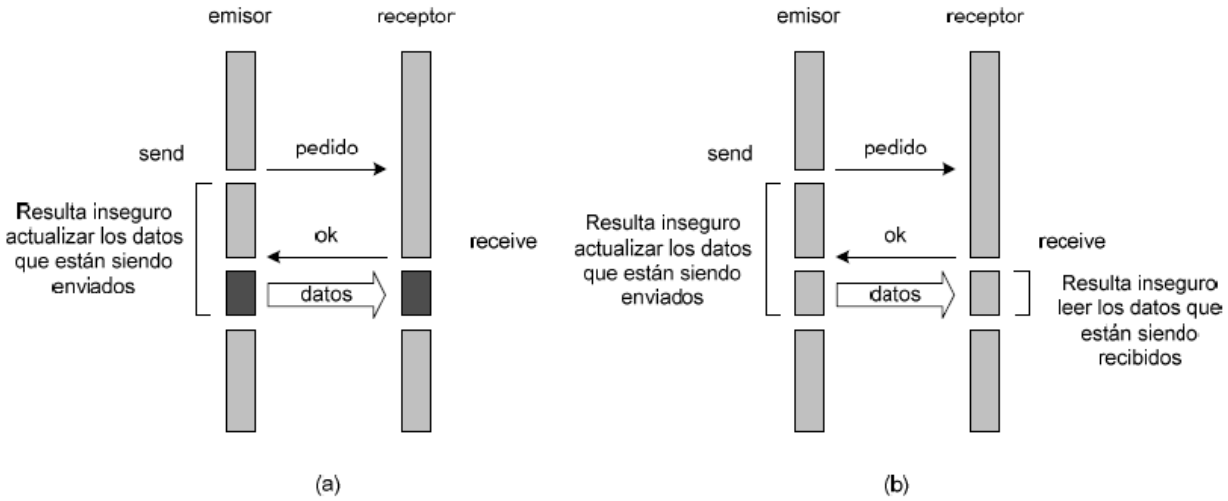


Figura 2.4

2.2.2.2 Costo de la comunicación

En los algoritmos paralelos una de las fuentes de overhead es el tiempo de comunicación entre nodos. Por esto resulta importante entender cuál es el costo de comunicación en el modelo de pasaje de mensajes.

El costo de la comunicación depende entonces del número de mensajes, del tamaño de cada mensaje, la estructura de interconexión y el modo de transferencia. A su vez el tiempo de cada mensaje depende de muchos factores, se puede pensar en una aproximación donde:

$T_{mensaje} = T_{startup} + W * T_{data}$, siendo $T_{startup}$ el tiempo de start up, *W* la cantidad de palabras que contiene el mensaje y T_{data} es el tiempo de transmisión de una palabra.

El tiempo de *start up* es esencialmente el tiempo necesario para enviar un mensaje vacío (y puede ser medido por esa simple acción) e incluye el tiempo que toma empaquetar el mensaje en el origen y desempaquetarlo en el destino, se asume que este tiempo es constante. El término T_{data} es el *tiempo de transmisión* de una palabra el cuál también se asume constante y usualmente es medido en bits/segundos. Muchos factores pueden afectar el *tiempo de comunicación* incluyendo por ejemplo la contención del medio usado para la transmisión. La ecuación mostrada anteriormente ignora el hecho de que el nodo origen y destino tal vez no estén directamente conectados y la necesidad de pasar el mensaje por nodos intermedios.

El *tiempo de comunicación total* puede verse como la suma del costo de cada mensaje $T_{mensaje}$. [Bar05]

2.2.2.3 MPI

Entre las principales herramientas para programar con pasaje de mensajes se encuentran “Message Passing Interface (MPI)" y anteriormente “Parallel Virutual Machine (PVM)”. Ambas son librerías para utilizar en lenguajes comunes como C, C++ y Fortran,

lo que las hace sencillas de manejar. La utilizada en esta tesina es la librería MPI la cual es un estándar que puede ser usado para desarrollar programas con pasaje de mensajes y define la sintaxis y la semántica de un conjunto de aproximadamente 125 rutinas pero no así su implementación. MPI fue desarrollado por un grupo de investigadores de academia e industria y es soportado por casi todos los fabricantes de hardware.

MPI provee entre otras funciones utilizadas para inicializar, administrar y finalizar comunicaciones, para transferir datos entre un par de procesos, se pueden clasificar las operaciones que provee MPI entre las siguientes categorías:

- Operaciones de seteo de ambiente: provee funciones para la inicialización y finalización de ambiente de MPI y para saber cuántos procesos MPI hay en ejecución así como el Rank o ID de cada uno dentro del ambiente.
- Comunicadores: un concepto clave en MPI es el de dominio de comunicación, el cual es un conjunto de procesos que tienen permitido comunicarse entre sí. Dentro de MPI los dominios de comunicación son conocidos como Comunicadores: MPI provee comunicadores y operaciones para manejarlos.
- Operaciones para enviar y recibir mensajes: provee las operaciones clásicas mencionadas anteriormente: *send* y *receive* tanto bloqueantes como no bloqueantes.
- Operaciones colectivas: MPI provee un conjunto de funciones para realizar operaciones colectivas sobre un grupo de procesos asociado con un comunicador para esto todos los procesos del comunicador deben llamar a la rutina colectiva. Ejemplos de estas operaciones son: *Broadcast* es usado para enviar un mensaje a todos los procesos del comunicador; la función *Gather* que recibe y concatena los datos enviados por todos los procesos dentro del comunicador; la operación *Reduce* que combina los mensajes de todos los procesos; entre otros. [Ana03]

Para más información sobre las funciones que provee MPI, ver Anexo B-MPI.

2.3 Memoria compartida distribuida

Como se menciona anteriormente las plataformas de memoria compartida son atractivas para los programadores debido a la facilidad de acceder a los datos y compartirlos entre diferentes unidades de procesamiento, sin necesidad de enviar información explícitamente de un procesador a otro. Además, se caracteriza por la necesidad de la sincronización para preservar la integridad de las estructuras de datos compartidas. Por otro lado, las plataformas de memoria compartida son atractivas debido a su fácil escalabilidad y la sincronización está dada por las primitivas para envío y recepción de mensajes. El objetivo de utilizar el modelo híbrido (memoria compartida distribuida) es aprovechar y aplicar las potencialidades de cada una de las estrategias que brindan tanto memoria compartida, como memoria distribuida, de acuerdo a la necesidad de la aplicación. Esta actualmente es un área de investigación de gran interés. [Bar05][Fab10]

Del mismo modo que en las plataformas de memoria distribuida, la memoria se encuentra físicamente distribuida entre todos los nodos del sistema, pero al igual que en memoria compartida, dentro de cada nodo, la memoria tiene un único espacio de direccionamiento, y puede ser accedida de forma directa por todas las unidades de procesamiento. [Bar05]

En la figura 2.5 [Ele04] se puede ver un esquema de memoria compartida distribuida:

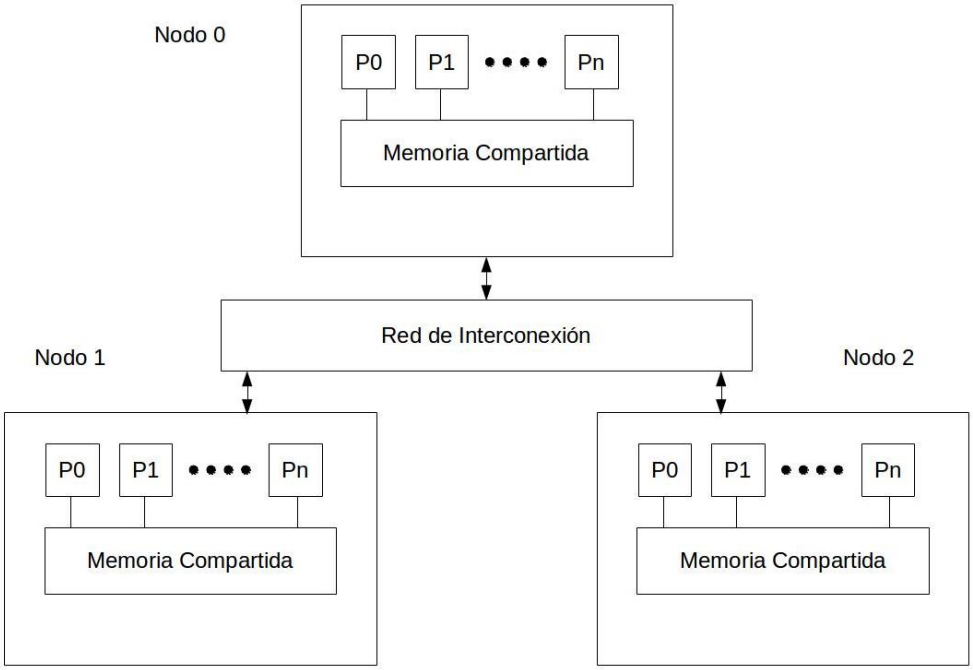


Figura 2.5

2.3.1 Cluster de multicores

Las arquitecturas paralelas han evolucionado en pos de obtener mejores tiempos de respuesta para las aplicaciones. En esta evolución pueden mencionarse los clusters, luego los multicores, y finalmente las arquitecturas de clusters de multicores. Estas últimas consisten básicamente en una colección de procesadores multicore interconectados mediante una red. [Fab10]

Los clusters de multicore introducen un nivel más en la jerarquía de memoria si se lo compara con los multicore: la memoria distribuida accesible vía red; que permite la interconexión de los diferentes procesadores que conforman el cluster. Si se enumera la jerarquía de memoria, la misma queda conformada de la siguiente manera: niveles de registros y caché L1 propio de cada núcleo, cache compartida de a pares de núcleos (L2), memoria compartida entre los cores de un procesador multicore y finalmente memoria distribuida vía red, tal como puede verse en la figura 2.6. [Fab12]

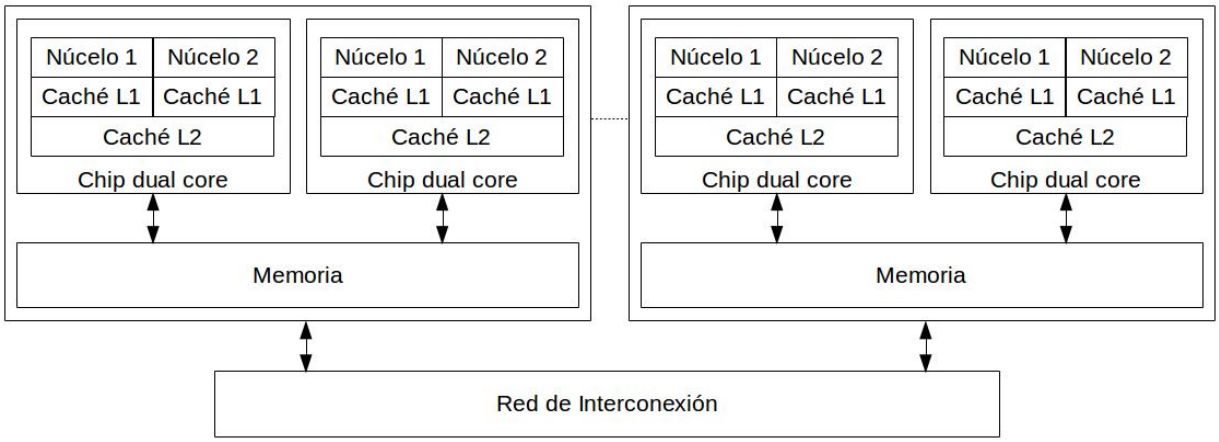


Figura 2.6

2.3.2 Modelo de comunicación

En las arquitecturas de memoria compartida distribuida el enfoque utilizado es el modelo híbrido, el cual combina los dos modelos de comunicación mencionados anteriormente tomando así lo mejor de cada uno.

Cuando se programa con memoria compartida se ve un gran beneficio y es la no necesidad de tener los datos replicados para cada procesador, sino que estos acceden a los datos a medida que lo necesitan; contrario a lo que pasa con pasaje de mensajes donde es necesario especificar el envío de un mensaje. La desventaja de memoria compartida es que para acceder a esos datos y que queden en un estado consistente es necesario coordinar el acceso mediante la programación de mutex o alguna otra herramienta.

Con el modelo de comunicación híbrido sobre un cluster de multicores se usa el modelo de pasaje de mensajes para mover información entre las distintas máquinas multicores y dentro de cada uno de ellos se utiliza el modelo de hilos, tal como se vio anteriormente. Es decir que por cada multiprocesador habrá un nodo MPI, y dentro de este habrá al menos tantos hilos como cores haya. Si bien la cantidad de hilos puede fijarse usando algún otro método, por ejemplo pensando en cuantos hilos son útiles para resolver la porción del problema dado, el tener un hilo por cada núcleo del multiprocesador suele ser la mejor opción para aprovechar al máximo la cantidad de tareas que pueden ejecutarse en paralelo, de todos modos esto depende de la naturaleza del problema, no sería la mejor opción si el mismo requiere mucho acceso a memoria o a recursos compartidos. También, podría utilizarse el modelo de pasaje de mensajes solamente, teniendo un nodo MPI por núcleo en cada una de las máquinas multicores, dando como resultado un modelo de pasaje de mensajes puro con las ventajas y desventajas de este, tal como se describe en la sección 2.1.2.2. [Cho09]

A continuación se ven algunas de las ventajas del modelo de comunicación híbrido:

- Dentro del mismo nodo el acceso a memoria es directo evitando el uso de mensajes, el tiempo que estos consumen y aliviando el tráfico en la red de interconexión por ejemplo cuando se hacen comunicaciones colectivas.
- Dentro de cada nodo se puede manejar de forma independiente el balance de carga de cada hilo.
- Optimización de la memoria utilizada, gracias a los hilos no es necesario replicar datos dentro de los procesos MPI.
- Cuando un nodo necesita enviar o recibir datos de otro nodo, un hilo puede ocuparse de la comunicación mientras que los otros pueden seguir trabajando en la solución del problema (si es que no necesitan ese dato para continuar).
- Es un modelo que se adecua mejor a la arquitectura, sobre un cluster de multicores usar solo MPI sería un modelado pobre.
- Reduce la utilización de memoria cuando el tamaño de ciertas estructuras dependen directamente del número de procesos MPI.

Este modelo también presenta algunas desventajas, por ejemplo que la complejidad del algoritmo puede resultar mayor al combinar dos modelos y librerías.

Capítulo 3

Diseño de aplicaciones

En el diseño de algoritmos paralelos se encuentran dos pasos fundamentales, en primera instancia dividir el trabajo en un conjunto de tareas que puedan ejecutarse en forma concurrente y luego de esto asignar estas tareas a diferentes elementos de procesamiento para su ejecución en paralelo. En este capítulo se describen diferentes técnicas para diseñar y desarrollar algoritmos paralelos. También, se describen los paradigmas de programación paralela, poniendo énfasis en el paradigma master-worker, debido a que es el que se utilizó para resolver los dos problemas que se abordan en los capítulos siguientes de esta tesina.

El desarrollo de algoritmos es una parte muy importante al momento de resolver problemas usando computadoras. Un algoritmo secuencial es básicamente una secuencia de pasos elementales para resolver problemas usando una computadora con un solo procesador. De la misma forma un algoritmo paralelo es una receta que indica cómo resolver un problema usando múltiples procesadores. No obstante, realizar un algoritmo paralelo involucra mucho más que especificar los pasos. Como mínimo un algoritmo paralelo acarrea la dimensión extra de la concurrencia, así el desarrollador tiene que especificar un conjunto de pasos que puedan ser ejecutados simultáneamente. En la práctica, realizar un algoritmo paralelo puede involucrar los siguientes procesos:

- Identificar porciones de trabajo (tareas) que pueden realizarse concurrentemente.
- Mapear tareas concurrentes a procesos en distintos procesadores.
- Distribuir los datos de entrada, intermedios y de salida asociados al programa.
- Manejar el acceso a los datos compartidos por múltiples procesadores.
- Sincronizar los procesos en varias etapas de la ejecución del programa paralelo.

Normalmente, hay varias opciones para cada uno de los pasos anteriores pero usualmente, pocas combinaciones de estas opciones desembocan en un algoritmo que alcance un rendimiento que sea acorde a los recursos empleados para resolver el problema. Frecuentemente ocurre que distintas de estas opciones obtienen el mejor rendimiento en distintas arquitecturas o bajo distintos paradigmas de programación. [Ana03]

3.1 Etapas de diseño

La mayoría de los problemas tienen varias soluciones paralelas y la mejor solución puede diferir totalmente de la sugerida por los algoritmos secuenciales existentes. Puede considerarse un enfoque metodológico en el diseño para maximizar el rango de opciones consideradas, brindar mecanismos para evaluar las alternativas y reducir el costo de backtracking por malas elecciones. También ofrece un enfoque exploratorio en el que cuestiones independientes tales como la concurrencia son considerados tempranamente y los aspectos específicos de la máquina se demoran hasta el final del proceso de diseño.

Esta metodología estructura el proceso de diseño en cuatro etapas distintas: particionamiento, comunicación, aglomeración y mapeo. En las primeras dos etapas, se enfoca en la concurrencia y la escalabilidad y busca describir algoritmos con estas

cualidades. En la tercer y cuarta etapa, la atención pasa a estar en la localidad y otras cuestiones relacionadas a la performance. A continuación se explican con más detalle.

3.1.1 Etapa de particionamiento

Uno de los pasos fundamentales para el desarrollo de algoritmos paralelos consiste en dividir el trabajo en un conjunto de tareas, de las cuales un subconjunto o todas puedan ser ejecutados en paralelo; este proceso es conocido como particionamiento o descomposición de tareas. Esta descomposición puede realizarse de muchos modos y debe tenerse en cuenta que no existe una única técnica para descomponer un problema y que sea siempre la mejor opción al momento de abordar la descomposición del mismo. En este sentido la elección de la técnica quedará supeditada a las características del problema que se pretenda resolver. Por ejemplo pensar en tareas de igual código (lo que se conoce normalmente como paralelismo de datos) y en tareas de diferente código (lo que se conoce como paralelismo funcional).

Estas técnicas se pueden clasificar de forma amplia en las siguientes categorías de descomposición: recursiva, de datos, descomposición exploratoria, especulativa y funcional, así también como se puede hacer una combinación de estas técnicas, resultando en una descomposición híbrida. Las técnicas de descomposición recursiva, descomposición funcional y de descomposición de datos son consideradas de propósito general y se pueden aplicar para resolver una amplia variedad de problemas. Por el contrario, las técnicas de descomposición exploratoria y descomposición especulativa son consideradas de propósito particular y por esta razón suelen aplicarse a una clase específica de problemas. [Ana03]

Esta etapa del diseño intenta exponer oportunidades para la ejecución paralela. De allí que el foco se encuentra en definir un gran número de pequeñas tareas con el objetivo de lograr una descomposición de grano fino del problema. En etapas posteriores del diseño, la evaluación de los requerimientos de comunicación, la arquitectura en la que se va a ejecutar o cuestiones de ingeniería de software pueden llevar a renunciar a oportunidades de paralelización identificadas en esta etapa. Luego se vuelve a rever la partición original y se aglomeran tareas para incrementar su tamaño o granularidad. Sin embargo, en esta primera etapa se desea evitar pensar en estrategias alternativas de particionamiento.

Un buen particionamiento consigue dividir en partes pequeñas el procesamiento asociado con un problema y los datos sobre los cuales opera esta computación. Existen varios enfoques para particionar, los cuales se describen a continuación. Estos enfoques pueden ser complementarios y aplicarse a diferentes componentes de un problema o inclusive pueden aplicarse al mismo problema para obtener algoritmos paralelos alternativos.

En esta primer etapa, se busca evitar replicar el procesamiento y los datos; o sea, definir tareas que particionen ambos, el procesamiento y los datos en conjuntos disjuntos. Como la granularidad, este es un aspecto del diseño que se debe rever luego. Sin embargo, puede valer la pena replicar alguno de los dos aspectos si eso permite reducir los requerimientos de comunicación. [Fos94]

3.1.1.1 Descomposición Funcional

La descomposición funcional representa una forma diferente y complementaria de pensar sobre los problemas. En este enfoque, el foco inicial se encuentra en el cómputo que se debe realizar más que en los datos utilizados en el cómputo. Si se tiene éxito en dividir el cómputo en tareas disjuntas, luego se procede a examinar los requerimientos de datos de estas tareas. Estos requerimientos de datos pueden ser disjuntos, en cuyo caso el particionamiento se encuentra completo. En caso contrario, podrían solaparse significativamente, lo que requeriría una comunicación considerable para evitar la replicación de datos. En general esto es frecuentemente un signo de que en realidad se necesita tomar un enfoque de particionamiento de datos.

La descomposición funcional tiene un rol importante como técnica de estructuración de programas. Una descomposición funcional que particiona el cómputo a realizarse y además el código que realiza dicho cómputo es muy probable que reduzca la complejidad de todo el diseño. Este es un caso frecuente en modelos computacionales de sistemas complejos, que pueden ser estructurados como colecciones de modelos más simples conectados por interfaces. Por ejemplo la simulación del clima puede involucrar componentes que representen a la atmósfera, el océano, hidrología, hielo, fuentes de dióxido de carbono y así sucesivamente. Mientras que cada componente puede ser paralelizado naturalmente usando técnicas de descomposición de datos, el algoritmo paralelo resulta mucho más simple si el sistema se descompone primero usando técnicas de descomposición funcional, aunque de este proceso no se obtenga un gran número de tareas.

En la figura 3.1 [Fos94] se puede ver una descomposición funcional de un modelo de computadoras del clima.

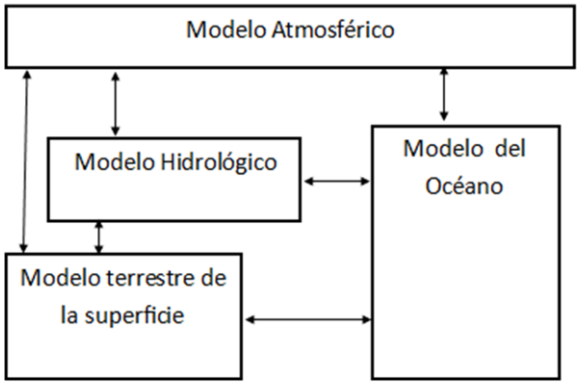


Figura 3.1

Cada componente del modelo puede ser pensado como una tarea por separado, a ser paralelizada por descomposición de datos. Las flechas representan intercambios de datos entre los componentes durante el cómputo: el modelo atmosférico genera datos de la velocidad del viento que son usados por el modelo del océano, este genera datos de la superficie del mar que son usados por otro modelo, y así sucesivamente. [Fos94]

3.1.1.2 Descomposición recursiva

El enfoque de dividir y conquistar es muy conocido en el desarrollo de algoritmos secuenciales. Un problema se divide en dos o más subproblemas. Cada uno de estos subproblemas se resuelve independientemente y sus resultados se combinan para obtener el resultado final. En general, los subproblemas son instancias más pequeñas o simples del problema original, dando lugar a una solución recursiva. Puede ser necesario

algún tipo de procesamiento para dividir el problema original o para combinar los resultados de los subproblemas.

Cuando se paraleliza esta estrategia, los subproblemas pueden ser resueltos al mismo tiempo, otorgando así suficiente paralelismo. El proceso de división y recombinación puede hacer uso de cierto paralelismo también, pero estas operaciones requieren comunicación. Sin embargo, dado que los subproblemas son independientes, no es necesaria la comunicación entre procesos que trabajan en diferentes subproblemas.

En general en esta descomposición se pueden identificar tres operaciones genéricas: dividir, procesar y combinar. La aplicación resultante se organiza en una especie de árbol donde algunos de los procesos crean subtareas y tienen que combinar los resultados de estos para producir un nuevo resultado. En general el procesamiento se realiza en los nodos hojas del árbol. [Mou99]

3.1.1.3 Descomposición de datos

La descomposición de datos es una técnica potente que se usa comúnmente para conseguir concurrencia en algoritmos que operan sobre estructuras de datos grandes. Con esta técnica la descomposición se realiza en dos pasos. En el primer paso, se particionan los datos sobre los que se trabajará y en el segundo paso el particionamiento que se obtuvo se usa para dividir en tareas el trabajo a realizar. Las operaciones que realizan estas tareas sobre diferentes particiones de datos son usualmente similares (por ejemplo una multiplicación de matrices) o son elegidas de un conjunto pequeño de operaciones (por ejemplo la factorización LU).

La partición de los datos puede realizarse de diferentes formas, en general uno debe explorar y evaluar todas las formas posibles de particionar los datos y determinar cuál es la que resulta en una descomposición natural y eficiente del problema. [Ana03]

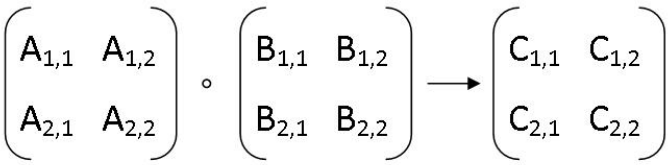
3.1.1.3.1 Particionamiento de datos de salida

Esta técnica puede emplearse en problemas donde cada elemento de la salida puede obtenerse a partir de los valores de entrada y se puede calcular de forma independiente de los demás. En estos tipos de problemas, un particionamiento de los datos de salida automáticamente resulta en una descomposición del problema en tareas, donde a cada una se le asigna el trabajo de computar parte de la salida.

Considérese el problema de multiplicar dos matrices A y B de n x n para obtener como resultado la matriz C. Por ejemplo, se podría descomponer este problema en 4 tareas, considerando que cada matriz está compuesta de 4 bloques o submatrices las cuales se definen al partir cada dimensión de la matriz por la mitad. Las 4 submatrices de C, cada una de tamaño $n/2$ x $n/2$, son procesadas de forma independiente por 4 tareas como la suma de cada producto correspondiente de las submatrices A y B.

En la figura 3.2 [Ana03] se pueden ver (a) las matrices *A* y *B* de dimensiones 4x4, que deberán multiplicarse para obtener la matriz resultado *C* y también se pueden ver (b) las posiciones de la matriz *C*, que debería calcular cada una de las 4 tareas, así también como los cálculos que debe realizar para obtener el valor de cada una de esas posiciones.

Figura 3.2



(a)

Tarea 1: $C_{1,1} = A_{1,1} B_{1,1} + A_{1,2} B_{2,1}$

Tarea 2: $C_{1,2} = A_{1,1} B_{1,2} + A_{1,2} B_{2,2}$

Tarea 3: $C_{2,1} = A_{2,1} B_{1,1} + A_{2,2} B_{2,1}$

Tarea 4: $C_{2,2} = A_{2,1} B_{1,2} + A_{2,2} B_{2,2}$

(b)

Debe notarse que la descomposición de datos es diferente de la descomposición en tareas. Si bien ambas están relacionadas y la primera frecuentemente sirve de ayuda a la segunda, una descomposición de datos no siempre resulta en una única descomposición de tareas. Por ejemplo si se toman las matrices de la Figura 3.2, pero ahora se quieren dividir los datos de salida en 8 en lugar de 4, se podrían obtener distintas tareas en base a la misma descomposición. Esto se puede ver en la figura 3.3. [Ana03]

Figura 3.3

Descomposición 1	Descomposición 2
Tarea 1: $C_{1,1} = A_{1,1} B_{1,1}$	**Tarea 1:** $C_{1,1} = A_{1,1} B_{1,1}$
Tarea 2: $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$	**Tarea 2:** $C_{1,1} = C_{1,1} + A_{1,2} B_{2,1}$
Tarea 3: $C_{1,2} = A_{1,1} B_{1,2}$	**Tarea 3:** $C_{1,2} = A_{1,1} B_{2,2}$
Tarea 4: $C_{1,2} = C_{1,2} + A_{1,2} B_{2,2}$	**Tarea 4:** $C_{1,2} = C_{1,2} + A_{1,1} B_{1,2}$
Tarea 5: $C_{2,1} = A_{2,1} B_{1,1}$	**Tarea 5:** $C_{2,1} = A_{2,1} B_{2,1}$
Tarea 6: $C_{2,1} = C_{2,1} + A_{2,2} B_{2,1}$	**Tarea 6:** $C_{2,1} = C_{2,1} + A_{2,1} B_{1,1}$
Tarea 7: $C_{2,2} = A_{2,1} B_{1,2}$	**Tarea 7:** $C_{2,2} = A_{2,1} B_{1,2}$
Tarea 8: $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$	**Tarea 8:** $C_{2,2} = C_{2,2} + A_{2,2} B_{2,2}$

3.1.1.3.2 Particionamiento de datos de entrada

Este tipo de particionamiento solo puede realizarse si cada dato de salida puede calcularse como función de los datos de entrada. En muchos algoritmos no es posible o deseable particionar los datos de salida. Por ejemplo, cuando se busca el mínimo, máximo o la suma de un conjunto de números, la salida es un valor único y desconocido.

En estos casos, a veces es posible particionar los datos de entrada y luego usar este particionamiento para conseguir concurrencia. Se crea una tarea por cada partición de los datos de entrada y esta tarea realiza tanto cómputo cómo es posible usando estos datos locales. Hay que notar que las soluciones a las tareas que se derivan del particionamiento de los datos de entrada podrían no resolver el problema original, sino una porción del mismo; en estos casos es necesario realizar procesamiento extra para combinar los resultados.

Otro ejemplo sería el problema de computar la frecuencia de conjuntos de ítems en base de datos de transacciones. El problema se describe de la siguiente manera. Dado un conjunto *T* que contiene *n* transacciones y un conjunto *I* que contiene *m* "conjuntos de ítems" (itemsets), cada transacción y "conjunto de ítems" contiene un subconjunto de ítems de todos los ítems posibles. Por ejemplo la base de datos que se puede ver en la figura 3.4 [Ana03] consiste de diez transacciones y resulta de interes computar la frecuencia de los ocho "conjuntos de ítems" de la segunda columna. En la tercera columna se puede ver la cantidad de frecuencias de los "conjuntos de ítems" en la base de datos. Por ejemplo, el "conjunto de ítems" {*D*, *K*} aparece dos veces, una en la primera transacción y otra en la novena. [Ana03]

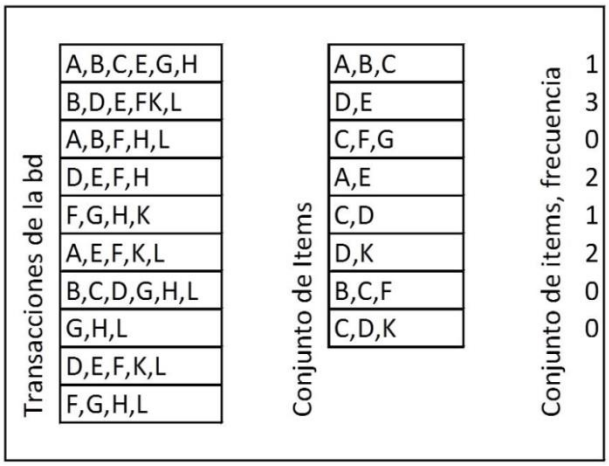


Figura 3.4

En la figura 3.5 [Ana03] se puede ver como el cómputo de las frecuencias de los conjuntos de items se puede descomponer en dos tareas mediante el particionamiento de los resultados en dos partes y hacer que cada tarea compute la mitad de las frecuencias. Notar que, en este proceso, la entrada de los conjuntos de items también ha sido particionada, aunque la motivación primaria para la descomposición que se puede ver en la figura es hacer que cada tarea compute independientemente el subconjunto de frecuencias que le fue asignado.

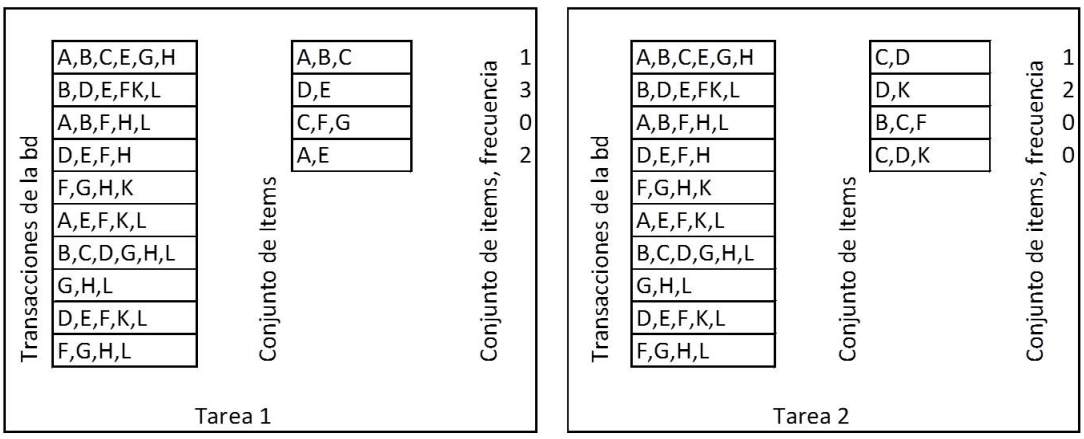


Figura 3.5

3.1.1.3.3 Particionamiento de datos de entrada/salida

En algunos casos, en los que es posible particionar los datos de salida, particionar los datos de entrada puede aportar concurrencia adicional.

3.1.1.3.4 Particionamiento de datos intermedios

Frecuentemente los algoritmos tienen una estructura de múltiples etapas de cómputo tal que la salida de una etapa es la entrada de la etapa siguiente. Una descomposición de un algoritmo de ese tipo puede obtenerse mediante el particionamiento de los datos de salida o entrada de una etapa intermedia del algoritmo.

En general, los datos intermedios no son generados explícitamente en la versión secuencial del algoritmo, por lo tanto el algoritmo original podría requerir algún tipo de reestructuración que permita aplicar este método.

El particionamiento de datos intermedios puede algunas veces llevar a una concurrencia mucho mayor que solamente particionar los datos de entrada o los datos de salida. [Ana03]

3.1.1.3.5 La regla del "dueño computa"

A una descomposición basada en el particionamiento de los datos de entrada o salida también se le conoce como la regla de "El dueño computa". La idea que engloba esta regla es que cada tarea está encargada de realizar todos los cálculos sobre los datos asociados a la partición que recibió. Esta regla puede tener distintos significados, dependiendo de la naturaleza del particionamiento. Por ejemplo, cuando se asignan particiones de datos de entrada a tareas, esta regla significa que una tarea debe hacer todos los cálculos posibles con los datos que le corresponden. En cambio, si el particionamiento se realiza sobre los datos de salida, la regla significa que una tarea debe realizar todos los cálculos para obtener el resultado de la partición que le fue asignada. [Ana03]

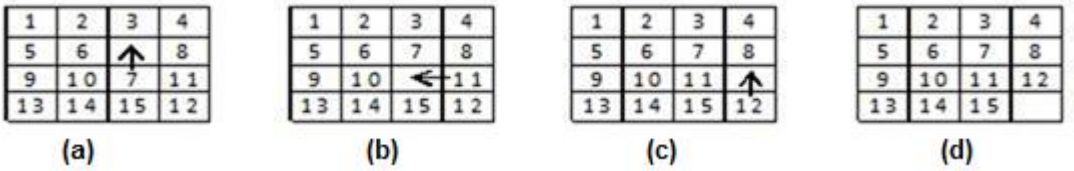
3.1.1.4 Descomposición exploratoria

La descomposición exploratoria es usada para descomponer problemas cuya solución yace en la exploración de un espacio de posibles soluciones que va evolucionando en tiempo real con su ejecución. Básicamente, la técnica consiste en particionar el espacio de búsqueda en partes más pequeñas y luego de forma concurrente buscar la solución en cada una de estas partes, si es necesario particionándolas nuevamente. Esta técnica podría generar speed-up superlineal, el cual se explica más adelante. Como ejemplos clásicos de problemas que se pueden resolver con esta técnica se pueden mencionar los problemas de optimización, redes neuronales evolutivas y juegos, entre otros.

Se toma como ejemplo el juego de "Puzzle-15", cuyos pasos a la solución final se pueden ver en la figura 3.6 [Ana03]. El juego consiste de 16 casilleros 15 de ellos, numerados del 1 al 15 distribuidos de forma aleatoria y el casillero restante en blanco, todos estos casilleros se encuentran dentro de una matriz de 4 x 4. Cada casillero se puede mover a la posición en blanco desde una posición adyacente a este, creando así un blanco en la posición donde se encontraba. Dependiendo de la disposición inicial de la matriz hay hasta 4 movimientos posibles: arriba, abajo, derecha e izquierda. El objetivo del juego, es una vez especificadas la configuración inicial (tradicionalmente una donde todos sus casilleros se encuentren mezclados sin seguir un orden dado) y la configuración final de la matriz (tradicionalmente con todos sus casilleros consecutivos en orden

ascendente), determinar una secuencia o la menor secuencia de movimientos posibles que lleve desde la configuración inicial de la matriz a la configuración final. En la siguiente figura se puede ver (a) una configuración inicial y (d) final, junto con una (b y c) serie de movimientos que lleva desde la primera a la última.

Figura 3.6



El “Puzzle-15” se resuelve típicamente usando técnicas de búsqueda sobre árboles. Comenzando desde la configuración inicial, todas las posibles configuraciones siguientes son generadas. Una configuración puede tener 2, 3 o 4 configuraciones siguientes, cada una correspondiente a la ocupación de la casilla vacía por uno de sus casilleros vecinos. La tarea de encontrar un camino desde la configuración inicial hasta la final ahora se convierte en encontrar un camino desde una de estas nuevas configuraciones hasta la configuración final. Ya que una ellas debe estar un movimiento más cerca de la solución (si la solución existe), ya se ha hecho un progreso en ese sentido. Al espacio de configuraciones generado por el árbol de búsqueda usualmente se lo conoce como grafo del espacio de estados. Cada nodo del grafo es una configuración y cada arista del grafo conecta configuraciones que pueden ser alcanzadas desde una configuración hacia otra mediante un solo movimiento.

El problema se podría resolver en paralelo de la siguiente forma. Primero, se generan de forma secuencial algunos niveles de configuración comenzando desde la posición inicial hasta que el árbol tiene la suficiente cantidad de nodos hojas (o sea, configuraciones del “Puzzle-15”). Ahora a cada nodo se le asigna una tarea encargada de explorar esa partición hasta que al menos una de ellas encuentre una solución. Cuando una de las tareas concurrentes encuentra una solución les informa a las demás que terminen su búsqueda. En la figura 3.7 [Ana03] se puede ver una descomposición en 4 tareas donde la tarea 4 encuentra la solución.

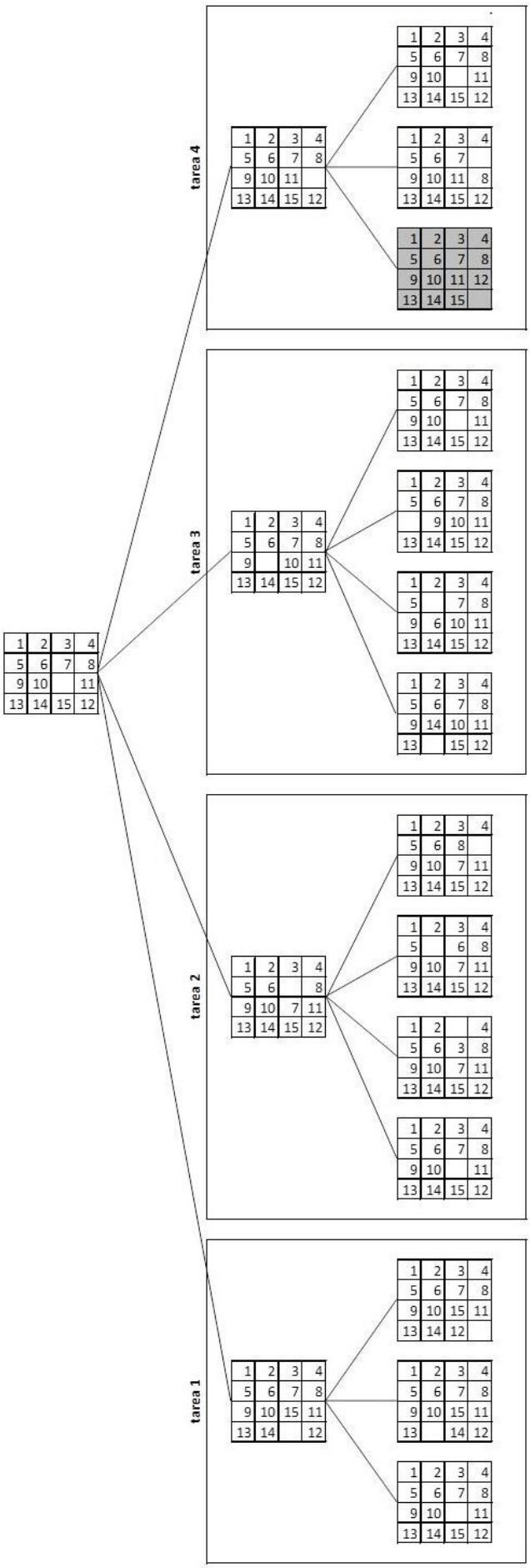


Figura 3.7

Es importante notar que si bien la descomposición exploratoria puede parecer similar a la descomposición de datos (puede pensarse el espacio de búsqueda como un particionamiento de los datos) tiene una diferencia fundamental en el siguiente sentido. Las tareas que se generan con la descomposición de datos se ejecutan por completo y cada una de ellas realiza cálculos útiles para encontrar la solución del problema. En cambio, con la descomposición exploratoria, las tareas que aún no han realizado todo el procesamiento que les correspondía, pueden terminar de ejecutarse ni bien alguna de las otras tareas encuentre la solución. De allí que la porción del espacio de búsqueda explorado (y el trabajo total realizado) por una ejecución paralela puede ser muy diferente de aquel explorado por un algoritmo secuencial. El trabajo realizado por una ejecución paralela puede ser o bien más pequeño o más grande que aquel realizado por un algoritmo secuencial. Por ejemplo, si se considera un espacio de búsqueda que fue particionado en 4 tareas concurrentes como se puede ver en la imagen a continuación. Si la solución se encuentra justo al principio del espacio de búsqueda correspondiente a la tarea 3 en el primer árbol, Figura 3.8, entonces será encontrada casi inmediatamente por la ejecución paralela. En este caso el algoritmo secuencial hubiera encontrado la solución solo después de haber calculado el espacio de búsqueda equivalente a las tareas 1 y 2 por completo. En cambio, si la solución se encuentra hacia el final del espacio de búsqueda de la tarea 1 en el segundo árbol, Figura 3.8, entonces la ejecución paralela realizará casi 4 veces la cantidad de trabajo que realiza el algoritmo secuencial y repercutirá negativamente en el speedup. Por ejemplo, en el primer caso se ve un claro ejemplo de speedup superlineal. [Ana03]

En la figura 3.8 [Ana03] se puede ver la distribución del trabajo.
Caso 1: *Tiempo secuencial total*: 2m +1, *tiempo paralelo total*: 1
Caso 2: *Tiempo secuencial total*: m, *tiempo paralelo total*: 4m

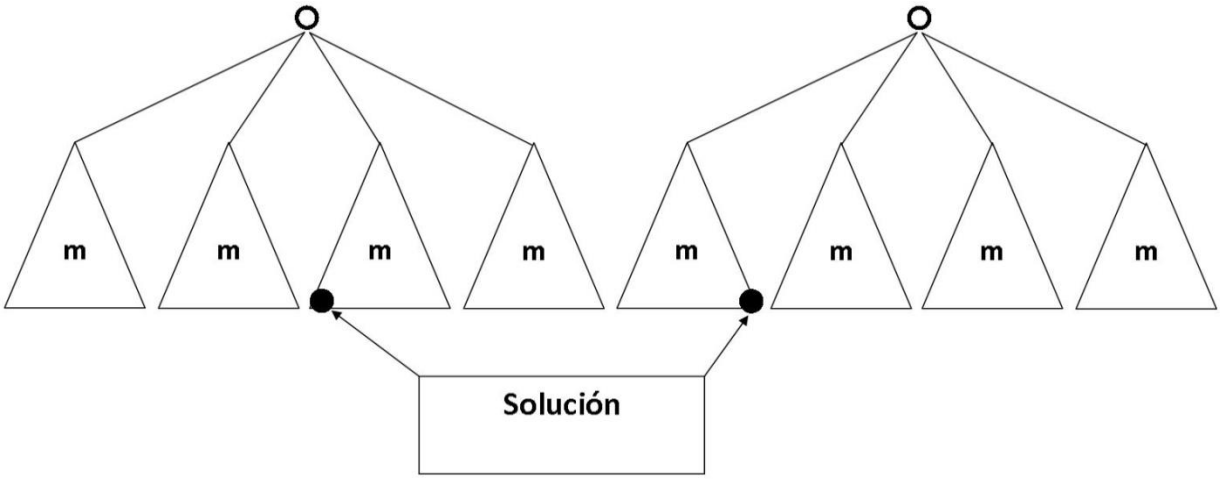


Figura 3.8

3.1.1.5 Descomposición especulativa

En muchos problemas las dependencias entre tareas no se conocen a priori y es difícil asegurar que se descompone en tareas totalmente independientes. La descomposición especulativa es usada cuando un programa puede tomar uno de varios caminos posibles de ejecución dependiendo de la salida de otras tareas que le preceden. Dada esta situación, mientras una tarea está realizando la computación que será usada para decidir la próxima computación, otras tareas pueden empezar las computaciones de la próxima etapa en forma concurrente. Para ver esto, se puede pensar en la sentencia switch de C:

SWITCH valor de entrada:

Condición A: acciones 1;
Condición B: acciones 2;
….
Condición N: acciones N;

Una descomposición especulativa sería muy similar a evaluar de una a más ramas de la sentencia en paralelo antes de que la entrada del switch se conozca. Mientras una tarea se encuentra calculando la entrada que resolverá el switch, otras tareas se pueden encargar de ir resolviendo las diferentes ramas del switch en paralelo. Finalmente cuando se haya computado el input del switch, se usará la computación correspondiente a la rama correcta mientras que las demás se descartarán. El tiempo de ejecución en paralelo es menor o igual que el tiempo de ejecución secuencial, ya que mientras se evalúa la condición de la que depende la próxima tarea, además este tiempo es utilizado para realizar computaciones valiosas para la próxima etapa de la ejecución paralela. Como contrapartida, este planteo paralelo del switch inevitablemente realiza cómputo que será descartado. Con el objetivo de minimizar computo que luego se descartará, se puede usar un planteo ligeramente diferente de la descomposición especulativa, especialmente en situaciones donde es más probable que se elija a una de las ramas del switch que a las otras. En este caso, debería calcularse solamente en paralelo la rama más probable y la computación precedente. Si finalmente se eligiera una rama diferente del switch a la que ya se había computado, entonces ese cómputo es descartado y se calcula la rama correcta del switch.

Es claro entonces que en este tipo de descomposición hay un compromiso entre el grado de concurrencia alcanzable y los procesos que se deban descartar y reiniciar. El speedup que se puede obtener con la descomposición especulativa dependerá y se incrementará de acuerdo a las múltiples etapas especulativas.

La descomposición especulativa es diferente de la descomposición exploratoria, ya que la entrada de una bifurcación que conduce a múltiples tareas es desconocida, mientras que por el contrario lo que se desconoce en la descomposición exploratoria, es la salida de múltiples tareas que se originan en una bifurcación. Otra diferencia que se encuentra es que en la descomposición especulativa, el algoritmo secuencial solamente ejecutaría una de las tareas en una etapa especulativa, porque cuando llega al comienzo de esa etapa, ya sabe exactamente que rama de la bifurcación debe tomar. De esto se desprende entonces que computar todas las posibles ramas de las cuales solo una se materializará y el resto se descartará, le agrega más trabajo al algoritmo paralelo que a su contraparte secuencial. Por el contrario, en la descomposición exploratoria, el algoritmo secuencial puede explorar diferentes alternativas una después de la otra porque la rama que lleva a la solución no se conoce de antemano. Por esta razón el algoritmo paralelo podría realizar más, menos o la misma cantidad de trabajo comparado con el algoritmo secuencial dependiendo de la ubicación de la solución en el espacio de búsqueda. [Ana03]

3.1.1.6 Descomposición híbrida

Todas las técnicas que se describieron anteriormente no necesariamente tienen que aplicarse de forma exclusiva para resolver un problema y pueden combinarse para obtener un grado mayor de concurrencia en las tareas. En general, una computación está estructurada en múltiples etapas y a veces es necesario aplicar diferentes tipos de descomposición en cada una de ellas. Por ejemplo, si se considera el problema de encontrar el número mínimo en un conjunto grande de *N* números, una descomposición puramente recursiva del problema podría generar muchas más tareas que el número *P* de

procesos disponibles. En cambio, una descomposición eficiente particionaría la entrada en *P* partes casi iguales y haría que cada tarea compute el mínimo de la secuencia que le fue asignada. El resultado final puede obtenerse encontrando el mínimo de los *P* resultados intermedios usando la descomposición recursiva. En la Figura 3.9 [Ana03] se puede ver un ejemplo en el que se debe calcular el mínimo entre 16 números, con *P*=4. Así cada una de las 4 tareas calcula el mínimo entre 4 números y luego de manera recursiva se calcula el menor entre esos 4 resultados intermedios.

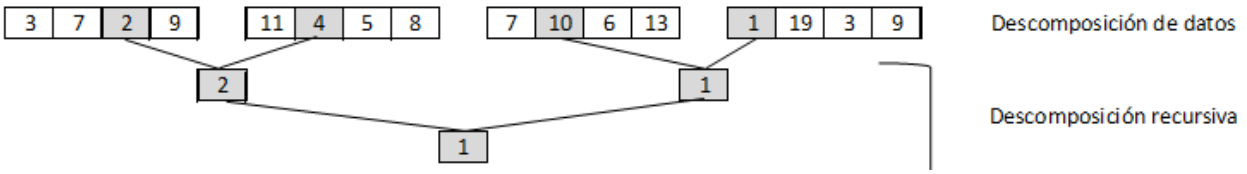


Figura 3.9

3.1.2 Etapa de comunicación

Se intenta que las tareas generadas por una partición se ejecuten concurrentemente pero en general no pueden ejecutar independientemente. El procesamiento que debe realizarse en una tarea requerirá usualmente datos asociados con otra tarea. Los datos entonces deben ser transferidos entre tareas para lograr que el procesamiento se lleve a cabo. Este flujo de información se especifica en la etapa de comunicación de la metodología de diseño. Así es que se conceptualiza la necesidad de comunicación entre tareas como un canal que las vincule, mediante el cual una tarea puede enviar y la otra recibir. De esta forma la comunicación en un algoritmo se puede especificar en dos fases. Primero, se define una estructura de canales que vincule, directa o indirectamente, a las tareas que necesiten datos (consumidores) con las tareas que proveen datos (productores). En segundo lugar, se especifican los mensajes que son enviados y recibidos en estos canales.

A continuación se ve una categorización de los patrones de comunicación en las siguientes categorías:

- Comunicación local. Cada tarea se comunica con un pequeño conjunto de otras tareas ("sus vecinos"); a diferencia de la comunicación global que requiere que cada tarea se comunique con muchas otras tareas.

- Comunicación estructurada. Una tarea y sus vecinas forman una estructura regular (como un árbol o una grilla), a diferencia de la comunicación desestructurada donde la estructura pueden ser grafos totalmente arbitrarios.

- Comunicación estática. La identidad de las tareas no cambia en el tiempo, a diferencia de la comunicación dinámica donde la identidad de las tareas puede ser determinada en tiempo de ejecución y puede ser altamente variable.

- Comunicación sincrónica. Los productores y consumidores ejecutan de forma coordinada, con pares de productores y consumidores cooperando en las operaciones de transferencia de datos, a diferencia de las comunicaciones asincrónicas que pueden requerir que un consumidor obtenga datos sin la cooperación del productor. [Fos94]

3.1.3 Etapa de aglomeración

En las etapas anteriores del proceso de diseño, se particiona la computación a realizar en un conjunto de tareas y luego se le agrega comunicación para proveer los datos requeridos por estas tareas. El algoritmo resultante todavía es abstracto en el sentido que no se encuentra especializado para ejecutarse eficientemente en cualquier computadora paralela.

En esta etapa, se pasa desde lo abstracto a lo concreto. Se revisan las decisiones tomadas durante las etapas de particionamiento y de comunicación con el objetivo de obtener un algoritmo que ejecute eficientemente en cierta clase de computadoras paralelas. Esta etapa tiene como objetivos considerar si es útil combinar o "aglomerar" tareas identificadas en la etapa de particionamiento, con el objetivo de tener un número menor de tareas pero cada una de mayor tamaño. También, determinar si merece la pena la replicación de datos y/o cómputo. En general las decisiones se ven afectadas por tres objetivos que entran en conflicto:

- Incrementar la granularidad al combinar varias tareas relacionadas en una única tarea reduciendo así los costos de las comunicaciones.
- Preservar la flexibilidad con respecto a la escalabilidad y las decisiones de mapeo; la capacidad de crear un número variante de tareas resulta crítica si se desea obtener un programa portable y escalable.
- La reducción de costos de ingeniería de software, se intenta evitar cambios extensivos por ejemplo mediante la utilización de rutinas existentes. [Fos94]

3.1.3.1 Incrementando la granularidad

En la etapa de particionamiento los esfuerzos se enfocan en definir tantas tareas como sea posible. Esto es útil ya que obliga a considerar un amplio rango de oportunidades para la paralelización. Sin embargo, tener un gran número de tareas paralelas no garantiza tener un algoritmo paralelo eficiente.

Uno de los problemas que más influye en la performance paralela son los costos de comunicación. En la mayoría de las computadoras paralelas se debe detener el procesamiento para poder enviar y recibir datos. Entonces se podría mejorar la performance al reducir el tiempo que el algoritmo emplea en comunicaciones. Claramente esto se puede lograr enviando menos datos. Tal vez, no de forma tan obvia, esto también se puede lograr enviando menor cantidad de mensajes, inclusive si se envía la misma cantidad de datos. Esto sucede porque no solo cada comunicación tiene un costo proporcional a la cantidad de datos a transferir sino lo costos asociados al manejo del canal de comunicación. Además, se deben tener en cuenta los costos asociados a la creación de tareas. [Fos94]

3.1.4 Etapa de mapeo

Una vez que el trabajo a realizar haya sido descompuesto en tareas es momento de asignar esas tareas a unidades de procesamiento con el objetivo de que sean ejecutadas en el menor tiempo posible, esto se conoce como mapeo de tareas. Con el objetivo de conseguir el menor tiempo de ejecución posible, es necesario minimizar el overhead asociado a la ejecución paralela de las tareas. Como se menciona en este trabajo, hay

dos fuentes principales de overhead, la comunicación entre tareas y que las mismas se encuentren en un estado ocioso. Una distribución desigual del trabajo hará que algunas tareas finalicen antes que otras. A veces, la dependencia entre tareas en el problema original obliga a que una tarea deba esperar el resultado o la terminación de otra tarea. Con frecuencia la interacción y el ocio ocurren en función del mapeo. Es por esta razón, que un buen mapeo de las tareas a unidades de procesamiento debe tratar de alcanzar dos objetivos: primero reducir el tiempo que los procesos pasan interactuando entre ellos y segundo reducir el tiempo total que algunos procesos se encuentran ociosos mientras otros se ejecutan.

Con frecuencia estos dos objetivos se contraponen el uno con el otro. Por ejemplo, el objetivo de minimizar las interacciones puede alcanzarse fácilmente asignando conjuntos de tareas que necesitan interactuar entre ellas, a la misma unidad de procesamiento. En la mayoría de los casos, esa decisión conducirá a un alto desbalance de carga entre las unidades de procesamiento, de hecho si se la lleva al límite esta estrategia terminará por mapear todas las tareas a una única unidad de procesamiento. Como resultado, los procesos con una carga de trabajo más baja estarán ociosos, mientras que aquellos que tengan una carga más pesada estarán tratando de finalizar su trabajo. De forma análoga, buscando balancear la carga entre las unidades de procesamiento, puede ser necesario asignar tareas que tienen una gran interacción a diferentes unidades de procesamiento.

El problema de mapeo es NP-completo, lo que significa que no existe un algoritmo computacionalmente tratable (de tiempo polinómico) para evaluar el mejor camino a tomar en un caso general. Sin embargo, se ha conseguido gran conocimiento en estrategias y heurísticas sobre la clase de problemas en las que estas son efectivas. Se puede ver entonces, que realizar un buen mapeo no es una tarea trivial.

Las técnicas de mapeo que se usan en los algoritmos paralelos pueden ser clasificadas en general en dos categorías: estáticas y dinámicas. El paradigma de programación y las características de las tareas y las interacciones entre ellas determinan si es más adecuado un mapeo estático o uno dinámico. [Ana03][Fos94]

A continuación se describen los distintos tipos de mapeo. Se debe tener en cuenta que estas técnicas se pueden combinar, dando como resultado lo que se denomina mapeo jerárquico.

3.1.4.1 Mapeo estático

Las técnicas de mapeo estático distribuyen las tareas entre las unidades de procesamiento previamente a la ejecución del algoritmo. Para tareas generadas estáticamente, puede usarse mapeo estático o mapeo dinámico. La elección de un buen mapeo en este caso depende de varios factores que incluyen el tamaño de las tareas, el tamaño de los datos asociados a las mismas, las características de interacción entre ellas e incluso el paradigma de programación paralela. Incluso cuando se desconoce el tamaño de las tareas, en general, el problema de obtener un mapeo óptimo es un problema NP-completo para tareas de diferente tamaño. No obstante, en la práctica, se encuentran heurísticas relativamente poco costosas que brindan soluciones casi óptimas con un grado de error aceptable al problema del mapeo estático. Los algoritmos que se valen de esta técnica en general son más sencillos de diseñar e implementar.

En general el mapeo estático suele usarse (aunque no de forma exclusiva) junto con la descomposición basada en los datos. También, se lo suele usar para mapear ciertos

problemas que son expresados naturalmente por un grafo de dependencias estático. A continuación se describen algunos esquemas de mapeo estático. [Ana03]

3.1.4.1.1 Mapeos basados en el particionamiento de datos

Dos de las formas más comunes de representar los datos en algoritmos son estructuras de tipo arreglo y los grafos. El particionamiento de datos en realidad desemboca en una descomposición, pero este particionamiento o descomposición son elegidos con un mapeo en mente. A continuación se describen los esquemas de distribución de arreglos y distribución de bloque aleatoria:

- Esquemas de distribución de arreglos. En una descomposición basada en el particionamiento de los datos, las tareas se encuentran fuertemente asociadas con porciones de los datos mediante la "regla del dueño computa". Es por eso, que mapear los datos relevantes a cada unidad de procesamiento es equivalente a mapear tareas a unidades de procesamiento. De esta forma se consigue un esquema de distribución de arreglos o matrices entre los procesos. Dentro de estos esquemas se encuentran las distribuciones en bloque que son una de las formas más simples de distribuir un arreglo. En este esquema se asignan diferentes porciones contiguas y uniformes de un arreglo a diferentes unidades de procesamiento. También, se encuentran la distribución de tipo cíclica, la cual se utiliza cuando la cantidad de trabajo a realizar difiere mucho para diferentes elementos de una matriz y una distribución de bloque podría conducir a desbalances de carga y ociosidad. Este esquema es una variación del esquema de distribución en bloque y la idea básica es particionar el arreglo en muchos más bloques que unidades de procesamiento disponible para luego asignarlos a las unidades de procesamiento siguiendo una estrategia round-robin de forma que cada unidad de procesamiento reciba varios bloques no adyacentes.

- Esquemas de distribución de bloque aleatoria. Podría suceder que una distribución de bloque-cíclico no sea capaz de balancear el trabajo cuando la distribución del mismo tenga algunos patrones especiales, como por ejemplo, sucede en una matriz esparcida (sparse) con una estructura diagonal (casi todos sus valores son cero excepto aquellos alrededor a la diagonal principal). Si en este caso se usara una distribución de tipo bloque-cíclico de dos dimensiones, se tendrían bloques con valores distintos de cero en los procesos diagonales (P0, P5, P10, P15) a diferencia de los demás. De hecho, algunos procesos como P12, no recibirán trabajo valioso. Esta distribución es una forma más general de distribución de bloques y puede ser usada en situaciones como la anterior. Aquí, el balance de carga se consigue particionando el arreglo en muchos más bloques que el número disponible de unidades de procesamiento. Sin embargo, los bloques se encuentran distribuidos de forma uniforme y aleatoria sobre las unidades de procesamiento.

3.1.4.2 Mapeo dinámico

El mapeo dinámico es necesario en situaciones donde el mapeo estático puede resultar en un alto desbalance de la distribución del trabajo entre los procesos o donde el grafo de dependencia de las tareas sea dinámico, descartando así el mapeo estático. Las técnicas de mapeo dinámico distribuyen el trabajo entre las unidades de procesamiento durante la ejecución del algoritmo. Si las tareas son generadas dinámicamente, entonces también deben ser mapeadas dinámicamente, además si se desconoce el tamaño de las tareas,

aplicar un mapeo estático podría conducir a serios desbalances de carga. En estos casos un mapeo dinámico es usualmente más efectivo.

Si la cantidad de datos asociados a las tareas es muy grande en relación al trabajo que realiza cada tarea, entonces usar un mapeo dinámico involucraría desplazar estos datos entre las unidades de procesamiento. El costo de desplazar estos datos puede ser aún mayor que las ventajas de usar un mapeo dinámico y finalmente podría ser más apropiado usar un mapeo estático en este caso (aunque genere cierto desbalance de trabajo). Sin embargo, el mapeo dinámico suele ofrecer un rendimiento aceptable en entornos de memoria compartida y datos de solo lectura.

Es importante notar que un paradigma de programación de memoria compartida no brinda inmunidad automática contra el costo que implica el desplazamiento de datos y debe contemplarse la arquitectura sobre la que se trabaja en cada caso particular. Los algoritmos que requieren un mapeo dinámico suelen ser más complicados de implementar, particularmente en un paradigma de programación con pasaje de mensajes. En general las técnicas de mapeo dinámico se clasifican en centralizadas o distribuidas. [Ana03]

3.1.4.2.1 Esquema centralizado

En un esquema de balance de carga dinámico centralizado, todas las tareas ejecutables son mantenidas en una estructura de datos central común, por un proceso especial o por un subconjunto de procesos. Si se designa un proceso especial para manejar el pool de tareas disponibles, entonces a este proceso se lo suele llamar el master y a los otros procesos que dependen del master para obtener trabajo se los llama esclavos. Cuando un proceso no tiene trabajo, toma una porción de trabajo disponible de una estructura de datos central o del proceso master. Cuando se genera una nueva tarea, se la agrega a esta estructura central o se la reporta al proceso master.

Los esquemas dinámicos centralizados en general son más simples de implementar que los esquemas distribuidos, pero su escalabilidad es limitada. A medida que más procesos son utilizados, el gran número de accesos a la estructura de datos central o al proceso master los vuelve un cuello de botella. [Ana03]

3.1.4.2.2 Esquema distribuido

En un esquema de balance de carga dinámico distribuido, el conjunto de tareas a ejecutar es distribuido entre los procesos los cuales intercambian tareas en tiempo de ejecución para balancear el trabajo. Cada proceso puede enviar trabajo o recibir trabajo desde cualquier otro proceso. Estos métodos no sufren de los cuellos de botella asociados con los esquemas centralizados. Al momento de analizar la viabilidad de un esquema distribuido es necesario tener ciertos parámetros críticos en cuenta:

- Cómo se encuentran relacionados los procesos que envían y reciben.
- Si la transferencia es iniciada por el emisor o el receptor.
- La cantidad de trabajo que se transfiere en cada comunicación. Si se transfiere poco trabajo, entonces el receptor podría no recibir el trabajo suficiente y realizar demasiadas transferencias podrían resultar en una interacción excesiva. En cambio, si se transfiere mucho trabajo, entonces el proceso emisor podría

quedarse sin trabajo en forma prematura, lo que nuevamente llevaría a realizar transferencias frecuentes.

- El momento apropiado para hacer la transferencia. Por ejemplo, en un balance de carga iniciado por el receptor, el trabajo puede ser requerido cuando el proceso se haya quedado sin trabajo o cuando al receptor le queda poco trabajo y anticipa que se quedará sin trabajo pronto. [Ana03]

3.2 Paradigmas de programación

Un modelo de programación paralela es una forma de estructurar un algoritmo paralelo mediante la elección de una técnica de descomposición y mapeo y aplicar la estrategia apropiada para reducir las interacciones entre tareas. En los sistemas paralelos existen algunos paradigmas que son usados para resolver distintos problemas, cuya solución requiere un modelado similar. Para cada paradigma podría escribirse un esqueleto algorítmico que define la estructura de control. A continuación se mencionan los paradigmas de programación paralela:

- Pipeline o Modelo productor-consumidor. En este modelo el problema se particiona en una secuencia de pasos y el flujo de datos pasa a través de una sucesión de una colección lineal de procesos. Aquí cada proceso consume la salida de su predecesor y produce una salida para su sucesor. Con la excepción de la tarea que inicia el pipeline, la llegada de nuevos datos dispara la ejecución de una nueva tarea por un proceso en el pipeline. O sea, que el pipeline puede verse como una cadena de productores y consumidores.
  Los procesos pueden formar pipelines de diferentes formas tales como, arreglos multidimensionales, árboles o grafos general con y sin ciclos. Por ejemplo un problema que se resuelve con este modelo es el filtrado, etiquetado y análisis de escena en imágenes.

- Divide y vencerás. El modelo de divide y vencerás se encuentra basado en la división repetida de problemas y datos en subproblemas más chicos hasta que estos se vuelven lo suficientemente pequeños y simples para ser resueltos directamente, lo que constituye la fase de dividir. Luego en la siguiente fase, conquistar, se trata la resolución independiente de cada uno de estos subproblemas. En general se consigue a través de la recursión en múltiples ramas. Por último en la fase de combinar, todas las soluciones parciales se combinan para obtener la solución global.
  En este modelo la subdivisión de tareas puede corresponderse con la descomposición entre los procesadores, lo que permite que cada subproblema pueda mapearse a un procesador. De esta forma cada proceso recibe una fracción de los datos, si llegó al caso base los procesa y en caso contrario crea la cantidad correspondiente de tareas con una instancia más simple de su subproblema y les distribuye los datos para que continúen buscando la solución. Por ejemplo problemas que se pueden resolver usando este paradigma son las búsquedas y ordenación de elementos en estructuras con naturaleza recursiva, como los algoritmos MergeSort y QuickSort.

A continuación se describe con más detalle el paradigma master-worker, dado que es el que se ha utilizado para resolver los problemas que esta tesina abarca. [Ana03] [Fos94] [Mou99]

3.2.1 Paradigma Master-Worker

En el modelo de programación Master-Worker o Master-Slave, se pueden distinguir dos tipos de entidades: los Maestros o Coordinadores (Masters) y los Esclavos o trabajadores (Workers). Siguiendo esta clasificación, los masters son los encargados de generar trabajo mediante la descomposición del problema en tareas más pequeñas, de distribuirlas entre los diferentes workers y de recibir los resultados parciales para componer la solución final del problema. Los workers realizan un procesamiento muy simple: deben recibir el trabajo que les envía el master y llevar adelante ese trabajo que les fue asignado, luego, una vez que no reciben más tareas y obtuvieron un resultado parcial deben enviárselo al master. Esto suele repetirse hasta que el master no tenga más tareas para resolver. Algunos modelos tienen una variante donde los masters se asignan el rol de worker una vez que distribuyeron las tareas lo que le permite aprovechar el tiempo que se encuentra ocioso esperando que los workers finalicen. [Ana03] [Fra13]

Los resultados que recibe el master pueden generar nuevas tareas para ser distribuidas entre los workers. De acuerdo a la independencia entre iteraciones pueden distinguirse dos casos básicos: [Fra13]

- Iteraciones dependientes. El master necesita en cada iteración los resultados enviados por todos los workers para poder generar un nuevo conjunto de datos a distribuir. De esta forma el trabajo tendrá que realizarse en etapas, debiendo finalizarse el trabajo de una etapa para poder comenzar con la siguiente. En este caso, es posible que el master requiera que todos los workers sincronicen luego de cada etapa.
- Entrada de datos independiente. Los datos se generan en el master sin necesitar los resultados anteriores para realizar una nueva distribución de los mismos.

Respecto al momento en que el master realiza la distribución de los datos se lo puede clasificar de dos formas básicas:

- Distribución estática. Consiste en distribuir todas las tareas entre los workers una vez que el master ha finalizado de hacer la descomposición de las mismas. Este tipo de distribución es válido cuando la cantidad de tareas para repartir y el tamaño de las mismas es conocido desde el comienzo de la aplicación. Como ventaja el master puede asumir el rol de worker realizando parte del trabajo una vez que ha repartido el trabajo a los workers. Cuando se trabaja sobre una arquitectura homogénea, se puede realizar una distribución directa de forma más simple, ya que las tareas pueden distribuirse en forma equitativa. En cambio, si la arquitectura fuera heterogénea, se tendría entonces que realizar una distribución predictiva. En esta distribución el master debe considerar la potencia de cómputo de cada procesador y en base a esto debería distribuir las tareas proporcionalmente a cada unidad de procesamiento. En la figura 3.10 [Ana03] se puede ver una representación esquemática de la distribución estática.

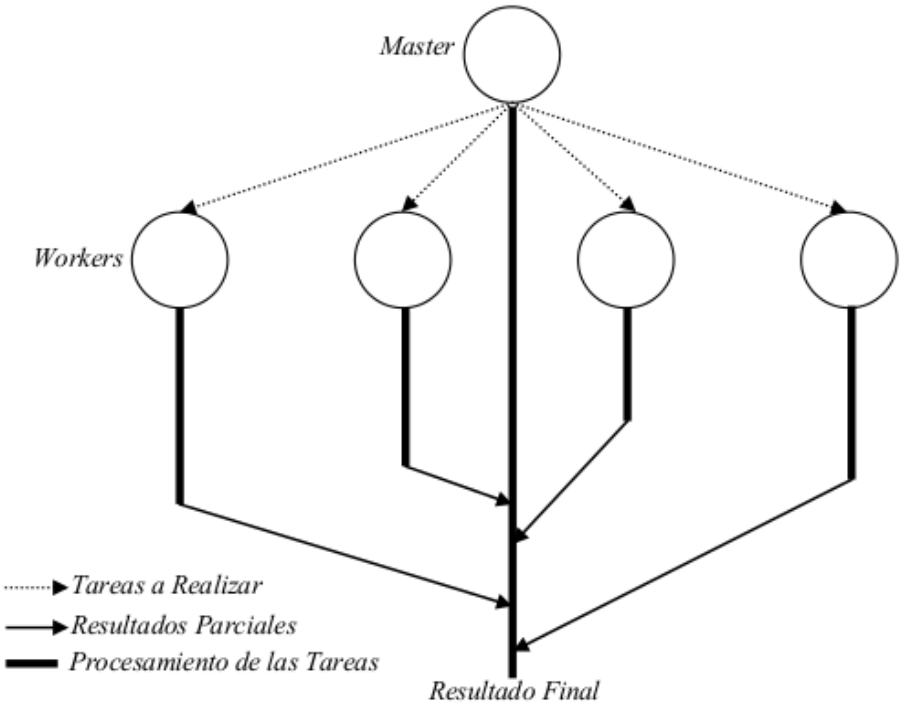


Figura 3.10

- Distribución dinámica bajo demanda. En una primera etapa se distribuye un cierto porcentaje de trabajo en forma estática (esto debería ser con distribución directa o predictiva de acuerdo a la arquitectura), y el resto se reparte bajo demanda, es decir que a medida que los workers consumieron los datos y se liberaron le piden más trabajo al master. Cuando a priori se desconoce la cantidad de tareas a realizar o el tamaño de las mismas, este método reduce el desbalance de carga, pero como contrapartida incrementa la comunicación entre tareas (master y workers) y la sincronización.

Este paradigma puede ser generalizado de forma jerárquica o multinivel con múltiples masters-workers, en el cual el master de más alto nivel se encuentra encargado de dividir el problema en grandes tareas las cuales envía a los masters del nivel más próximo, quien a su vez se encarga de subdividir estas tareas en tareas más pequeñas las cuales distribuye entre sus workers asociados y potencialmente estos masters de segundo nivel podrían realizar parte del trabajo. Este modelo en general puede ser implementado con éxito tanto en paradigmas de memoria compartida como en paradigmas de pasaje de mensajes ya que la interacción es naturalmente de dos vías: el master sabe a quién debe darle trabajo y los workers saben que deben recibir trabajo del master.

Cuando se usa este paradigma se debe tener especial cuidado de que el proceso master no genere un cuello de botella, lo que podría ocurrir si las tareas son muy pequeñas o los workers demasiado rápidos. Si esto sucediera los procesos workers terminarían inmediatamente y quedarían ociosos esperando que se les asigne más trabajo en lugar de estar procesando. Para evitar esta situación la granularidad de las tareas debe ser elegida de forma tal que el costo de procesar el trabajo sea superior al costo de transferirlo y de sincronizar entre ambos procesos (master y workers respectivos). [Ana03] [Fra13]

En esta tesina se utilizan tres diferentes implementaciones del modelo master-worker, los cuales se explican a continuación, y son referenciados en capítulos posteriores.

3.2.1.1 Modelo Uno

Solución Master / Worker con sólo un nivel de master: es el modelo clásico, en cual hay un master, y varios workers asociados a él, pidiéndole trabajo a medida que resuelven las asignaciones. En la figura 3.11 se puede ver un esquema del modelo:

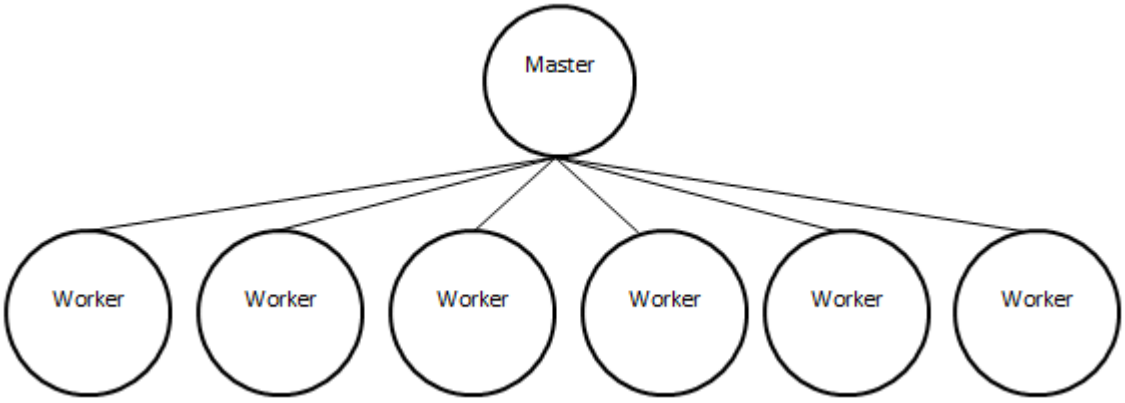


Figura 3.11

Una de las ventajas de este modelo es que sólo un proceso se encuentra atendiendo pedidos y todos los demás se encargan de realizar cómputo que realmente ayuda a la resolución del problema elegido. Por el otro lado dado que todos los worker tienen que pasar por el mismo master a pedir tarea, puede que este se convierta en un cuello de botella, no pudiendo responder los pedidos a la velocidad en que llegan.

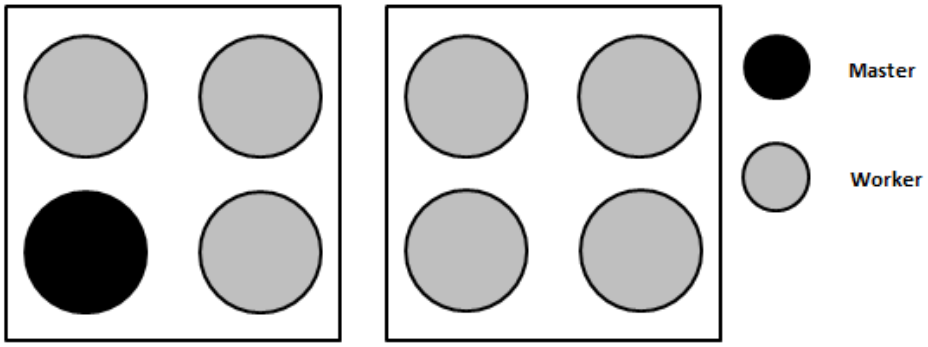


Figura 3.12

Cuando en la experimentación realizada para esta tesina se utiliza este modelo sobre un cluster homogéneo de multiprocesadores, sólo un núcleo de una máquina realiza el trabajo del master, y el resto de los núcleos de todos los multiprocesadores realizan trabajo de worker. En la figura 3.12 se puede ver el esquema explicado para una arquitectura con dos multiprocesadores de cuatro núcleos cada uno.

3.2.1.2 Modelo Dos

Solución Master/Worker con dos niveles de master: en este modelo se tiene un master principal que se encarga de resolver pedidos de trabajo, pero esta vez no se lo entrega directamente a los workers, sino a un segundo nivel de master, donde cada uno de estos tiene asociado workers que le piden trabajo. En la 3.13 se puede ver este esquema.

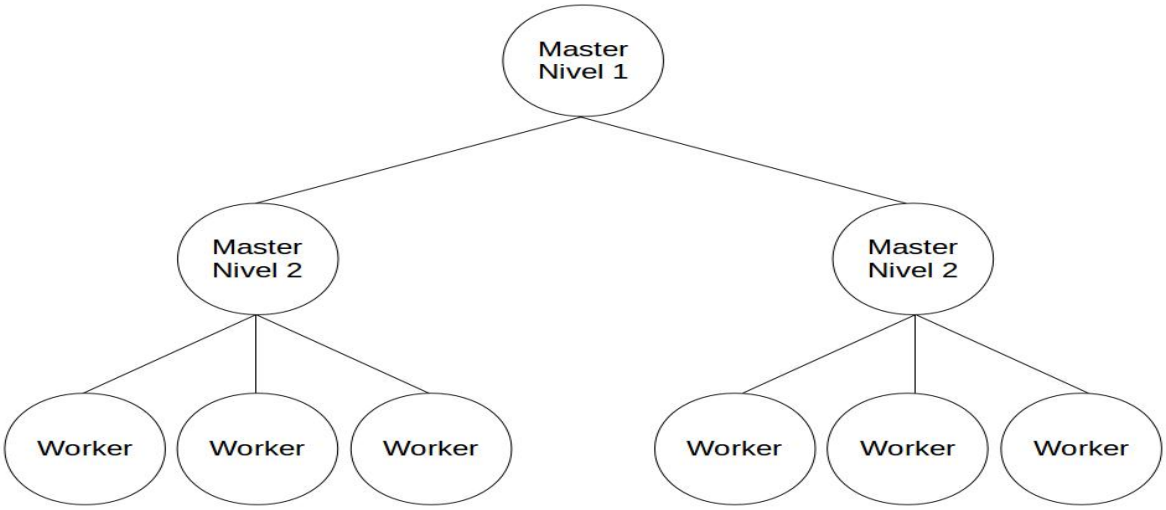


Figura 3.13

De este modo, cuando un worker necesita trabajo se lo pide al master de segundo nivel al que está asociado, y cuando este se queda sin más trabajo para asignar, le pide más trabajo al master primario. Una ventaja de este modelo es que el master principal o de nivel uno, atiende muchos menos pedidos y de una cantidad reducida de procesos, de modo de evitar que se forme un cuello de botella. Pero por el otro lado puede haber un overhead introducido por pasar el trabajo a un intermediario en lugar de directamente al proceso que lo necesita y además procesos que en el modelo anterior podían dedicarse a resolver cómputo asociado al problema, ahora ya no están disponibles para eso, reduciéndose así la cantidad de núcleos que procesan cómputo asociado a la resolución del problema.

Cuando en la experimentación realizada para esta tesina se utiliza este modelo, sólo un núcleo de una máquina realiza el trabajo del master, luego en cada uno de los multiprocesadores restantes, un núcleo toma el rol de master del segundo nivel y el resto toma el rol de worker pidiendo trabajo al master de nivel dos alocado en ese mismo multiprocesador. En la figura 3.14 se puede ver cómo se vería este esquema en una arquitectura con dos máquinas de cuatro núcleos cada una.

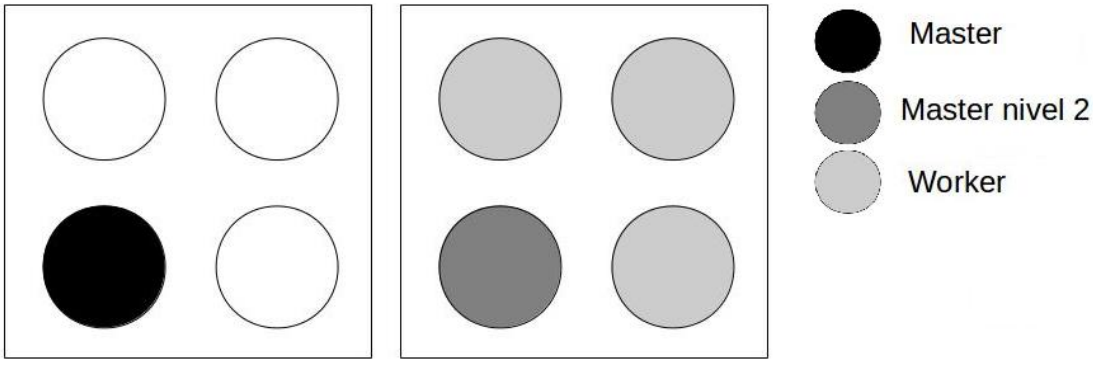


Figura 3.14

3.2.1.3 Modelo Tres

Master-Worker con dos niveles de master (híbrido): en este esquema también hay un master principal o de nivel uno, que recibe pedidos de “master secundarios”, pero a diferencia del esquema anterior, cada worker toma recursos de un buffer compartido, y el primero que vaya a buscar tarea a este recurso y se encuentre con que está vacío, toma el rol de master de nivel dos, pidiéndole tareas al master de nivel uno, y alojándolas en el buffer para volver a ocupar el rol de worker.

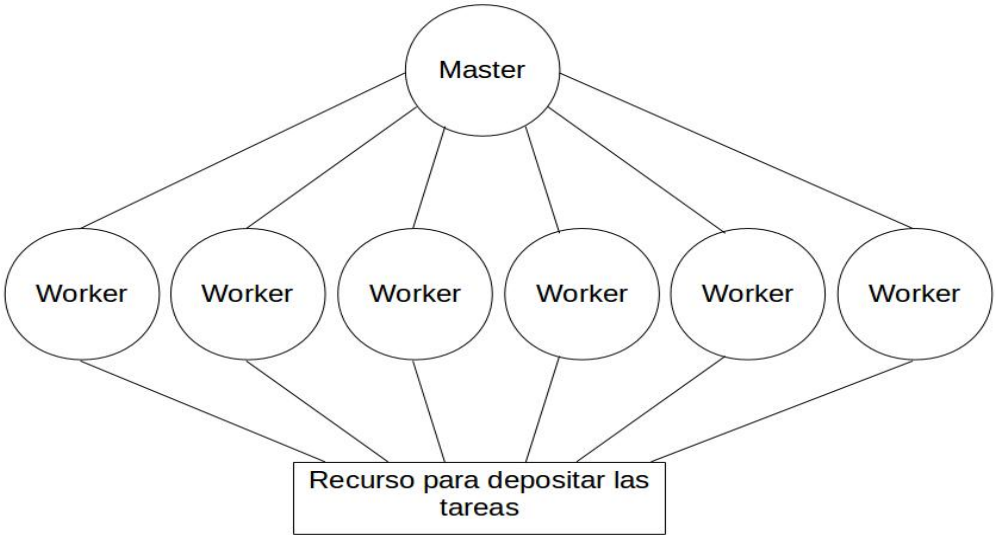


Figura 3.15

Es decir que en este esquema cualquier worker puede tomar el rol de master secundario cuando sea necesario. La ventaja con respecto al modelo anterior es que salvo por el master de nivel uno, todos los demás ejecutan cómputo importante para la resolución del problema, pero como desventaja el acceso a ese buffer compartido tiene que ser sincronizado.

Cuando en la experimentación realizada para esta tesina se utiliza este modelo, sólo un núcleo de una máquina realiza el trabajo del master, luego en cada uno de los multiprocesadores restantes, todos los núcleos desempeñan el rol de worker pero potencialmente cualquiera puede alterar su rol por un momento, a master de nivel dos. En la figura 3.16 se puede ver este esquema.

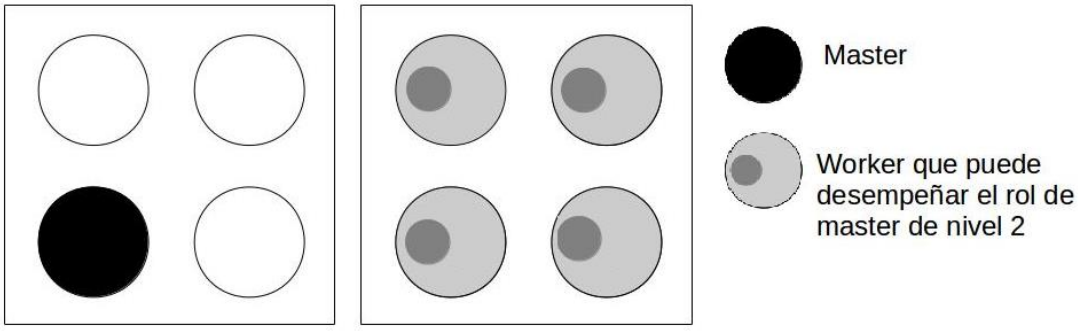


Figura 3.16

Se podría hacer una modificación en este esquema, de modo que la unidad que cambie su rol de master a Worker sea siempre la misma, por ejemplo el núcleo 0 de cada máquina. Pero así, cuando un Worker se encuentra con el buffer vacío, tendría que esperar que el proceso asignado termine de trabajar para que se realice pedido al master, en lugar de iniciar el pedido de forma instantánea.

Capítulo 4

Evaluación de Sistemas Paralelos

El estudio de la performance de sistemas paralelos se realiza con el objetivo de determinar cuál es el mejor algoritmo para resolver un problema dado, evaluar las plataformas de hardware y los beneficios de la paralelización. Para poder llevar acabo el análisis existen varias métricas basadas en distintas características del sistema paralelo. [Ana03]

Teóricamente, tanto en programación paralela como en programación secuencial, el tiempo y la memoria utilizados son métricas dominantes para estimar la performance. En la práctica, si se tuviera que elegir entre dos métodos que utilizan diferentes cantidades de memoria, y se tuviese suficiente memoria para ejecutar ambos, se elegiría el más rápido. Por lo que se tiene que el tiempo se convierte en el factor más importante. [Sar95]

Además, una de las motivaciones más importantes para el uso de sistemas paralelos, es la reducción del tiempo de ejecución de un programa el cual depende de muchos factores, tales como la arquitectura que se utiliza, el compilador, el sistema operativo, entre otros. Es decir que el tiempo de ejecución depende tanto del ambiente, como del modelo utilizado para desarrollar el algoritmo. [Rau10]

En este capítulo se analizan diferentes métricas para cuantificar el rendimiento de los sistemas paralelos y a partir de esto poder realizar la evaluación de los mismos.

4.1 Fuentes de overhead

Al aumentar los recursos de hardware en un sistema paralelo, se espera que el programa reduzca su tiempo de ejecución en forma proporcional a dicha mejora. Sin embargo, esto raramente se logra, dado a la variedad de overheads asociados al paralelismo.

Es importante analizar las fuentes de overhead porque tienen una influencia directa en el rendimiento de los sistemas paralelos; por lo que su estudio puede llevar a optimizar los algoritmos desarrollados.

Además del tiempo que un programa paralelo tarda en ejecutar el procesamiento esencial para resolver el problema (esto es, el procesamiento que realizaría un programa secuencial para resolver la misma instancia del problema), también puede utilizar tiempo para la comunicación entre procesos, para cómputo extra asociado a la paralelización, o permanecer cierto tiempo osioso. [Ana03]

En la figura 4.1 [Ana03] se puede ver el tiempo de ejecución de un programa paralelo hipotético ejecutándose en ocho elementos de procesamiento. En la figura se puede ver el tiempo que es usado para ejecutar el programa así como el tiempo de comunicación y el tiempo ocioso de cada proceso.

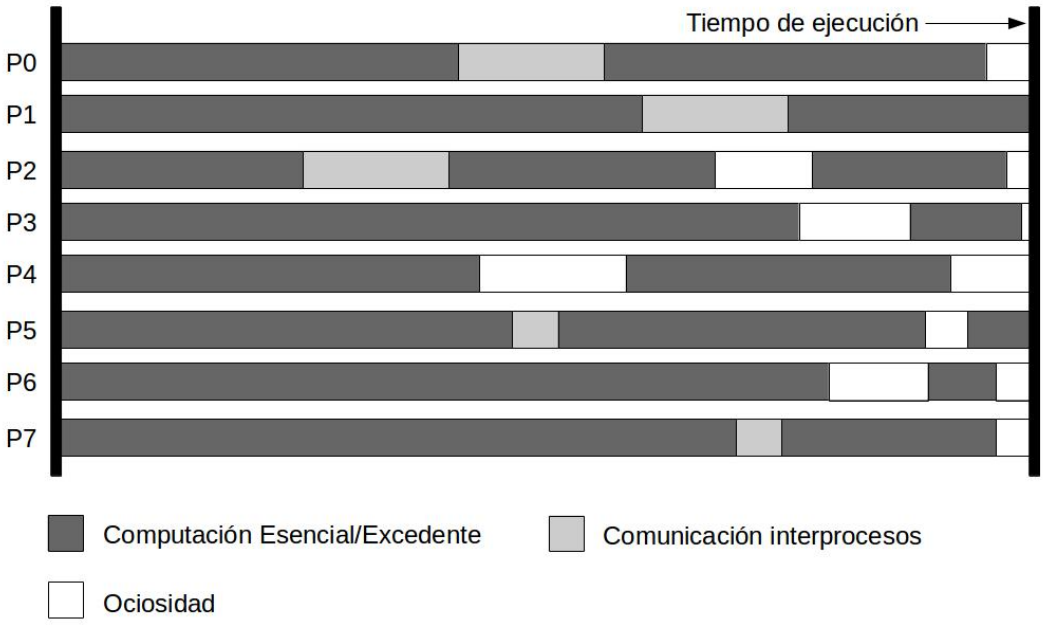


Figura 4.1

4.1.1 Interacción entre procesos

Todo sistema paralelo no trivial requiere que sus elementos de procesamiento interactúen e intercambien datos (por ejemplo, resultados parciales). Usualmente, el tiempo utilizado en comunicar datos entre los elementos de procesamiento, es la fuente de overhead más importante en los sistemas paralelos. [Ana03]

Supóngase que se tiene un esquema Master-Worker con un proceso master, y diez procesos Worker. En este caso la interacción entre procesos a simple vista estaría dada por el momento en que cada Worker le pide más trabajo al Master y el momento en que cada Worker le envía los resultados parciales de su procesamiento al Master.

4.1.2 Ocio de los procesadores

En un sistema paralelo las unidades de procesamiento pueden estar momentáneamente en un estado ocioso por diversas razones:

- Desbalance de carga. Si los diferentes elementos de procesamiento tienen distintas cargas de trabajo, algunos elementos podrían estar ociosos mientras otros terminan de procesar su trabajo. En muchas aplicaciones paralelas (por ejemplo, en las que la generación de tareas se realiza de forma dinámica) es difícil predecir el tamaño de cada sub-tarea asignada a los elementos de procesamiento.
- Sincronización. En algunos programas paralelos, los elementos de procesamiento deben sincronizarse en ciertos puntos durante la ejecución, si los elementos no están listos todos al mismo tiempo, los que sí lo estén, permanecerán ociosos hasta que el resto lo esté.
- Código que debe ser ejecutado de forma secuencial. Puede que para un problema dado algunas partes del algoritmo no puedan ser paralelizadas, permitiendo que sólo un elemento de procesamiento trabaje en esas partes, y mientras lo hace, el resto debe esperar ocioso. [Ana03]

4.1.3 Cómputo extra asociado a la paralelización

En algunas ocasiones el algoritmo secuencial óptimo para un problema, puede ser difícil o imposible de paralelizar, forzando a usar un algoritmo paralelo basado en la paralelización de un algoritmo que no es el mejor, pero que es más fácil de paralelizar (esto es, que tenga un mayor grado de concurrencia). En este caso, el overhead de procesamiento, es la diferencia entre el mejor programa secuencial, y el programa paralelo.

Un algoritmo paralelo basado en el mejor algoritmo secuencial podría incluso ejecutar mayor cantidad de cómputo agregado que el algoritmo secuencial. Un ejemplo es el Algoritmo rápido de la "Transformada de Fourier", el cual en su versión secuencial se podría reusar soluciones para ciertas computaciones, sin embargo, en la versión paralela esos resultados no pueden ser reusados dado que las distintas computaciones son ejecutadas por distintas unidades de procesamiento por lo que deben ser recalculadas desde el principio.

Además, algunas veces el algoritmo paralelo debe realizar pasos extras, como por ejemplo repartir el trabajo entre los diferentes procesos. [Ana03]

4.2 Métricas

Cuando se utiliza un algoritmo paralelo para la resolución de un problema, interesa saber cuál es la ganancia en la performance obtenida y resulta importante estudiar el rendimiento de los programas paralelos con el fin de determinar el mejor algoritmo, evaluar las plataformas de hardware, y examinar los beneficios del paralelismo.

A continuación se explican algunas métricas que pueden ser utilizadas para estudiar los sistemas paralelos.

4.2.1 Tiempo de ejecución

Una de las métricas más elementales es el tiempo de ejecución de un programa. El tiempo de ejecución de un programa secuencial T_s, es el tiempo que transcurre entre el inicio y el fin de la ejecución en un sólo procesador.

El tiempo de ejecución de un programa paralelo T_p, es el tiempo que transcurre desde que el procesamiento paralelo inicia en el primero de los procesos, hasta el momento en que el último termina su ejecución. [Ana03]

El tiempo de ejecución no siempre es la mejor métrica para evaluar la performance de un sistema paralelo, dado que normalmente varía de acuerdo al tamaño del problema. El tiempo de ejecución debería normalizarse para comparar la performance de un algoritmo con diferentes tamaños del problema. [Fos95]

4.2.2 Desbalance de carga

Como se mencionó anteriormente, una de las principales fuentes de overhead es el ocio de los procesadores, y este puede ser causado por el desbalance de carga entre los mismos. Esto ocurre principalmente cuando el trabajo a realizar no está distribuido equitativamente, esto no solo se refiere cantidad de trabajo, sino a la complejidad computacional del mismo.

Una métrica que permite evaluar este aspecto de los sistemas paralelos, es el desbalance de carga. Para medir el desbalance de carga entre los procesadores que intervienen en una aplicación paralela, se calcula la diferencia de trabajo relativa obtenida a partir de la siguiente ecuación:

$$\text{Desbalance} = \frac{maximo_{i=1..N}(T_i) - mínimo_{i=1..N}(T_i)}{promedio_{i=1..N}(T_i)},$$

En la fórmula, T_i, es el tiempo que el procesador *i*, tardó en ejecutar su parte del trabajo, y *N*, es la cantidad de procesadores que se utilizaron para ejecutar el algoritmo. [DeG05]

4.2.3 Speedup

Cuando se desarrolla una solución paralela, resulta interesante saber cuál es el beneficio de la paralelización con respecto a la solución secuencial. Esto puede ser calculado con la métrica de rendimiento conocida como Speedup, la cual toma como parámetros el tiempo de ejecución de la mejor solución secuencial T_s, y el tiempo de ejecución de la solución paralela para el mismo problema T_p. El speedup se calcula como sigue:

$$Speedup(p) = \frac{T_s}{T_p}$$

En la fórmula, *p* indica la cantidad de procesadores utilizados para ejecutar el algoritmo paralelo, y como se mencionó antes T_s y T_p son los tiempos de ejecución secuencial y paralelo respectivamente. [Ana03]

Desde el punto de vista teórico, lo mejor que puede suceder es que todos los procesadores se utilicen todo el tiempo o que todos los procesadores se utilicen a su máxima capacidad de cómputo. Esto induce a asumir que la capacidad de cálculo de la computadora paralela es igual a la suma de las capacidades de cálculo de cada uno de los procesadores que son parte de la misma. Si se tiene una arquitectura paralela homogénea esto significa que utilizar un procesador más indica reducir proporcionalmente el tiempo de ejecución paralelo. Es decir que si se utilizan *p* procesadores, el mejor tiempo paralelo estaría dado por el tiempo de ejecución del mejor algoritmo secuencial T_s dividido la cantidad de procesadores *p*. De hecho, no es más que asumir que la potencia de cálculo de la máquina paralela con *p* procesadores es *p* veces mejor que la potencia de cálculo que la máquina secuencial, es decir, un procesador.

Puesto de otra forma, el speedup consiste en identificar la relación entre la potencia de una máquina con un procesador y una máquina paralela con *p* unidades de procesamiento. De esta manera se llega a que el speedup óptimo en las computadoras paralelas homogéneas es igual a la cantidad de procesadores que se utilizan. Esto es equivalente a definir la potencia de cálculo relativa de la máquina paralela con respecto a un procesador, o directamente el valor del speedup óptimo como sigue:

$$\text{Speedup}_{óptimo} = \sum_{i=0}^{p-1} prc(proc_i)$$

En la fórmula $proc_0$, $proc_1$,..., $proc_{p-1}$, son los *p* procesadores de la máquina paralela, y *prc*($proc_i$) es la potencia relativa de cómputo del procesador $proc_i$ con respecto a los demás procesadores y se calcula como sigue:

$prc(proc_i)=\frac{potencia\ (proc_i)}{potencia(proc_{mejor})}$.

En una arquitectura heterogénea se calcularía el speedup óptimo reemplazando los valores en las fórmulas anteriormente mencionadas, pero asumiendo que todos los procesadores son iguales se cumpliría que $prc(proc_i)=1$; *para todo i=0…p-1* y por lo tanto $\text{Speedup}_{óptimo}$=p. [Tin04]

Como se mencionó anteriormente algunas partes del cómputo de un programa paralelo no pueden ser ejecutadas en paralelo. Asumiendo que un programa paralelo tendrá partes donde sólo una unidad de procesamiento estará ocupada (partes secuenciales) y que durante el resto de las partes todas las unidades de procesamiento se encuentran ocupadas sin incurrir en overhead y que se llama *f* a la fracción del programa que no puede ser ejecutada en paralelo, se tiene que el tiempo requerido por el programa paralelo utilizando *p* unidades de procesamiento está dado por $f.T_s + (1-f) T_s / p$, de aquí se obtiene que el speedup estará dado por la siguiente ecuación:

$$\text{Speedup(p)}= \frac{T_s}{f.T_s+(1-f).T_s/p} = \frac{p}{1+(p-1).f}$$

La ecuación anterior es conocida como ley de Amdahl y dice que la mejora obtenida por un modo de ejecución más rápido está limitada por el tiempo en que este modo se ejecuta; aún con un número infinito de unidades de procesamiento, el speedup máximo está limitado a *1/f*:

$$\text{Speedup(p)} \lim_{p\to\infty} = \frac{1}{f}.$$

Solo un sistema paralelo ideal con *p* procesadores puede lograr un speedup igual a *p*. En la práctica, este comportamiento ideal, no se logra debido a diferentes cuestiones inherentes a los algoritmos paralelos, por ejemplo el tiempo de comunicación entre los procesos, computación extra que no aparece en la versión secuencial del algoritmo, o periodos en los cuales no todos los procesadores están ejecutando trabajo útil y están ociosos.

Puede suceder que el resultado del speedup, dé un número mayor a *p*, este fenómeno es conocido como “Speedup Superlineal”. Usualmente un resultado como este puede ser provocado por utilizar como parámetro el tiempo de ejecución de un algoritmo secuencial que no es el óptimo; por una característica de la arquitectura del sistema que ponga en desventaja el algoritmo secuencial y favorezca al paralelo, o por una característica propia de la naturaleza del algoritmo. [Bar05]

4.2.4 Eficiencia

Resulta útil tener una métrica que indique qué fracción del tiempo total usado por un procesador, está destinado efectivamente a cómputo esencial del problema. Esta métrica de rendimiento es conocida como eficiencia, y se calcula de la siguiente manera:

$$Eficiencia= \frac{S(p)}{p}$$

En la fórmula anterior *S(p)* hace referencia al speedup, y *p* a la cantidad de procesadores utilizados en la solución paralela. [Ana03]

Cuando la arquitectura utilizada en el sistema paralelo es heterogénea, la relación entre el speedup y la cantidad de elementos de procesamiento no aporta una medida relevante del rendimiento del algoritmo. Esto se debe a que el máximo rendimiento de la arquitectura heterogénea no está dado por la cantidad de elementos de procesamiento sino con la potencia total de cómputo de la misma.

Para calcular la eficiencia tanto en arquitecturas paralelas homogéneas como heterogéneas, se puede generalizar la fórmula. La nueva función entonces se define como la relación entre el speedup logrado por el sistema paralelo y la potencia de cálculo total (o lo que es lo mismo el speedup óptimo) de la arquitectura utilizada:

$$Eficiencia = \frac{S(p)}{S(p)_{óptimo}}$$

En un sistema paralelo ideal el speedup logrado y el speedup óptimo son iguales, por lo que la eficiencia es igual a 1. En la práctica, el Speedup es menor que el óptimo, y la Eficiencia toma un valor entre 0 y 1 dependiendo de la efectividad con la que cada elemento de procesamiento es utilizado. Aquellos casos en los que se obtiene un speedup superlineal, son los únicos en los que el valor de la eficiencia supera el valor 1. [Ana03]

4.2.5 Escalabilidad

A menudo, los programas son diseñados y testeados para tamaños de problema pequeños y con pocas unidades de procesamiento, pero estos programas suelen ser desarrollados con la finalidad de resolver problemas de tamaños muchos mayores, y sobre una mayor cantidad de unidades de procesamiento. Entonces, mientras que desarrollar y testear los programas para instancias pequeñas del problema con una instancia reducida de la arquitectura es más simple, predecir la performance del programa, se torna más difícil; esto se debe a que un algoritmo que posee una buena performance para un problema seleccionado o sobre un número determinado de procesadores en un máquina dada, puede funcionar pobremente si alguno de los parámetros cambia. [Ana03] [Lau99]

Se puede ver entonces para un algoritmo, cómo afecta la variación de distintos parámetros, para un primer caso varía la cantidad de procesadores, dejando fijo el tamaño del problema, y para el siguiente caso, lo contrario, o sea queda fija la cantidad de procesadores, pero varía el tamaño del problema. En las figuras 4.2 y 4.3 se pueden ver el comportamiento ante estas variaciones:

- Para un tamaño de problema dado, mientras la cantidad de procesadores aumenta, la eficiencia del sistema paralelo decrece. Este fenómeno es común a todos los sistemas paralelos.

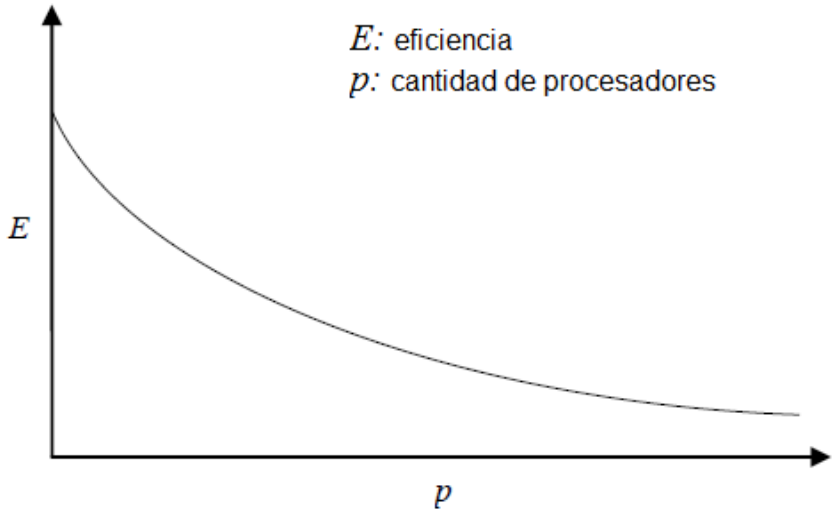


Figura 4.2

- En muchos casos, la eficiencia del sistema paralelo aumenta si el tamaño del problema se incrementa, mientras la cantidad de procesadores se mantiene constante.

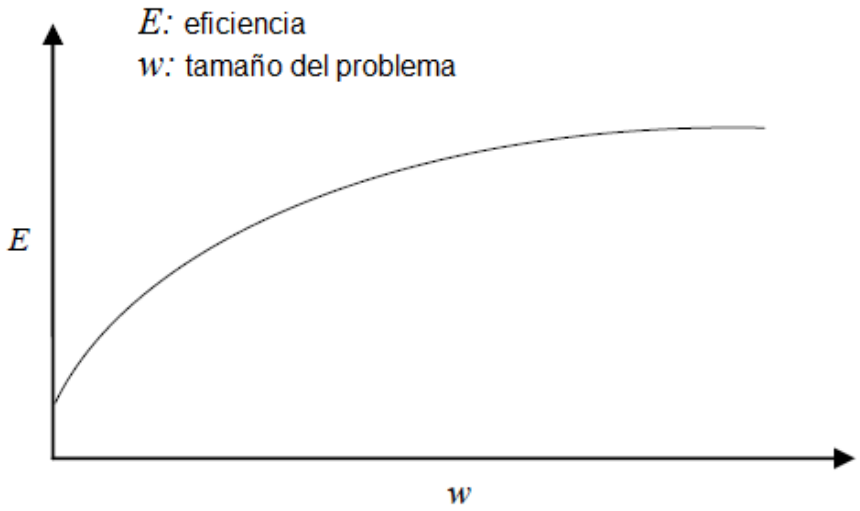


Figura 4.3

En base a esto, se puede definir un sistema paralelo escalable a aquel en el cual la eficiencia puede permanecer constante si se incrementa la cantidad de procesadores, así como el tamaño del problema. Pero debería determinarse la medida en que debe crecer el tamaño del problema, con respecto al número de procesadores, para mantener la eficiencia constante. Esta medida varía para los distintos sistemas paralelos, y es la que determina el grado de escalabilidad de los mismos; cuanto menor es, mayor eficiencia se tiene. [Ana03]

Vale aclarar, que cuando se habla del tamaño del problema, no se hace referencia necesariamente al tamaño de la entrada, sino a la cantidad de pasos computacionales que se requieren, se puede decir entonces, que el tamaño del problema se define en función del tamaño de la entrada, y esta función es distinta para cada problema. [Bar05]

Capítulo 5

Problema "N-reinas"

En este capítulo se explica en qué consiste el problema de las "N-reinas" y algunas de sus posibles aplicaciones en el mundo real. Además, se hace un repaso de su historia y el estado actual del mismo, para finalmente exponer la mejor solución secuencial conocida y la forma en que se paralelizó la misma considerando que es una clase de aplicación en la que el trabajo depende de los datos, donde no puede conocerse de antemano y el tiempo de procesamiento local es mucho mayor al tiempo de comunicación entre procesos.

5.1 Origen del problema

No está muy claro en la literatura académica cuál fue el primer lugar en donde se describió el problema de N-reinas, pero la mayoría de las referencias parecen apuntar al artículo publicado por Max Bezzel en 1848 cuyo título (traducido del alemán) es "Propuesta del problema de las 8 reinas". Sin embargo, la publicación más antigua conocida fue realizada por Nauck en 1850. Ese mismo año, Gauss postuló la existencia de 72 soluciones para tableros de 8x8. Posteriormente, en el año 1874, Glaisher probó la existencia de 92 soluciones. [Ana05]

5.2 Descripción del problema

Actualmente N-reinas es un problema muy conocido en combinatoria y consiste en encontrar todas las formas posibles de ubicar *N* reinas en un tablero de ajedrez sin que se ataquen entre ellas y es una generalización del conocido problema de las "8-reinas", el cual consiste en ubicar 8 reinas en un tablero de ajedrez de 8 casillas por 8 casillas sin que se ataquen entre ellas. Las reinas atacan a otra pieza moviéndose en cualquier posición legal (vertical, horizontal o diagonal) tantos casilleros como deseen, el único movimiento que no pueden realizar es el del caballo en ajedrez.

El problema entonces se puede formular para una cantidad *N* de reinas, al que comúnmente se hace referencia como el problema de las "N-reinas" y en su caso más general se puede enunciar de la siguiente manera: consiste en determinar todas las posibles combinaciones de *N* reinas en un tablero de ajedrez de *NxN posiciones* donde ninguna de ellas ataque a las demás. [Bru75]

En la figura 5.1 [Fra13] se pueden ver dos ejemplos para tableros de *N = 5*, donde uno es válido (a) y el otro inválido (b).

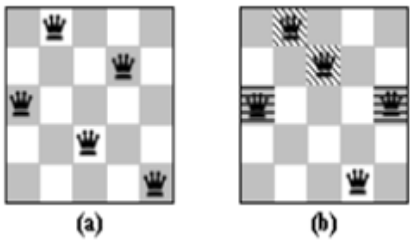


Figura 5.1

El problema de las N-reinas es conocido como un problema puramente teórico apropiado para probar algoritmos nuevos. Sin embargo tiene diversas aplicaciones ya que

se lo considera como un modelo de máxima cobertura. Una solución al problema de las N-reinas garantiza que cada objeto puede ser accedido desde cualquiera de sus ocho casillas adyacentes (dos verticales, dos horizontales y cuatro diagonales) sin que tenga conflicto con otros objetos. Entre sus aplicaciones se encuentran: [Ana05]

- Control de Tráfico Aéreo.
- Sistemas de Comunicaciones.
- Programación de Tareas Computacionales.
- Procesamiento Paralelo Óptico.
- Compresión de Datos.
- Balance de Carga en un Computador multiprocesador.
- Ruteo de mensajes o datos en un Computador multiprocesador.

Se puede hacer una primera aproximación a una solución mediante un algoritmo secuencial básico que utilice la fuerza bruta, el cual prueba todas las combinaciones posibles para ubicar las reinas en el tablero, descartando todas las posiciones inválidas y dejando solamente las válidas. Esta solución tiene una mejora que consiste en abortar la búsqueda para una combinación dada cuando es posible detectar que esa combinación no genera posiciones válidas. Además, se puede obtener una mejora adicional considerando que una combinación valida puede generar hasta 8 soluciones diferentes, como consecuencia de obtener rotaciones y simetrías de la misma, reduciendo así el espacio de búsqueda de soluciones. [Fra13] [QUE14]

En las figuras 5.2 [QUE14] y 5.3 [Fra13] se pueden ver dos ejemplos de simetrías y rotaciones donde una combinación produce 8 soluciones diferentes. En la imagen 5.2 se pueden ver las rotaciones y simetrías genéricas.

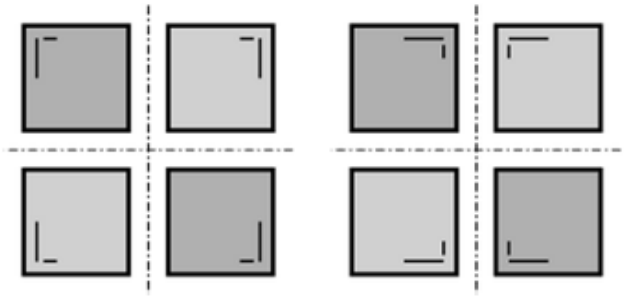


Figura 5.2

En la figura 5.3 se pueden ver rotaciones y simetrías en un tablero de 5x5.

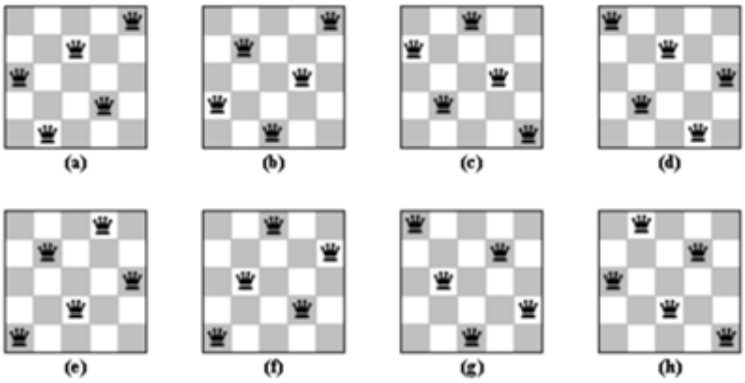


Figura 5.2. (a) y (b) combinación original y su simétrica, (c) y (d) rotación 90° y su simétrica, (e) y (f) rotación 180° y su simétrica, (g) y (h) rotación 270° y su simétrica.

Figura 5.3

El problema de las “N-reinas” pertenece a la clase de problemas NP-completos y su costo en tiempo se incrementa de forma exponencial a medida que aumenta el tamaño del tablero (*N*), lo que hace complejo trabajar con tableros donde *N* tiende a valores altos. [Fra13] [Cra92]

Actualmente, el número máximo de tablero que se ha conseguido resolver es de 26 casilleros y fue conseguido por la “Universidad Técnica de Dresden” (“Technische Universität Dresden”) mediante un proyecto que combinó el uso de múltiples FPGAs en paralelo, que comenzó el 14 de Octubre de 2008 a las 10:10 am y culminó el 11 de Julio de 2009 a las 4:28 pm CEST, dando como resultado 22317699616364044 soluciones.

Resulta claro que por sus características el problema de las N-reinas es de interés para analizar la performance de una arquitectura distribuida: es escalable, requiere balanceo en función de los datos, tiene diferentes soluciones secuenciales/paralelas, y pueden diferenciarse claramente cómputo y comunicaciones. [Fra13] [Don03]

5.3 Algoritmo secuencial

A continuación se realiza una breve descripción del mejor algoritmo secuencial para resolver el problema en un tablero de NxN.

El algoritmo secuencial realiza *N/2* iteraciones, en cada una de ellas ubica la reina correspondiente a la primera fila en una posición diferente. Las *N/2* posiciones restantes no son analizadas debido a que son soluciones simétricas de la rotación a 90° grados de los casos evaluados.

Luego, a raíz de la ubicación de la primera reina, se determina el vector de posiciones posibles para la reina en la segunda fila, de modo que no se ataquen entre ellas, y para cada una de ellas, se calcula la cantidad de soluciones posibles que las mismas generan.

Para obtener la cantidad de soluciones a partir de la fila *i*, tal que toda fila anterior a *i* tiene ubicada su reina, se establece nuevamente el vector de posiciones válidas pero esta vez para *i+1*, donde para cada una de ellas se vuelve a repetir este paso. Esto continúa hasta llegar a ubicar una reina en la última fila, o cuando no haya más posiciones válidas en una cierta fila. Al llegar a ubicar una reina en la última fila, se calcula la cantidad de soluciones diferentes que genera dicha combinación y su simétrica al ser rotadas a 90°, 180° y 270°.

A continuación se presenta un pseudo-código para la solución secuencial:

```
ubicarReina(f, pos, tab): Ubica la reina en la posición pos de la fila f del tablero tab.
determinarPosVálida(c, tab, f): determina el conjunto c de posiciones válidas para la fila f en el tablero tab.
posiciónVálida(c): saca y retorna la primer posición válida del conjunto, o retorna N si el conjunto está vacio.
rotar(g, tab): rota el tablero tab a g grados
simétrico(tab): calcula el tablero simétrico de tab

cantidadSoluciones(tablero){
        if (rotar(90, tablero) ==tablero) or (rotar(90, tablero)==simétrico(tablero)){
                return 2; }
        else{
                if (rotar(180, tablero) ==tablero) or (rotar(180, tablero)==simétrico(tablero)){
                        return 4; }
                else{
                        return 8; }
        }
}

calcularSoluciones( fila, conjPosVálida, tablero){
pos=posiciónVálida(conjPosVálida);
if (fila=N) and (pos<=N){ //se pudo ubicar la reina en una posición válida en la fila N (encontré una solución)
        ubicarReina(fila, pos, tablero);
        return cantidadSoluciones(tablero);
        }
else{
        total=0;
        while(pos<=N){
        //repito el proceso de buscar posiciones válidas en la fila siguiente, para cada pos válida
                ubicarReina(fila, pos, tablero);
                detPosVálida(nuevoConjPosVálida, tablero, fila+1);
                total+= calcularSoluciones(fila+1, nueoConjPosVálida, tablero);
                pos=posiciónVálida(conjPosVálida);
                }
        return total;
        }
}

main(){
cantSoluciones=0;
for(pos=1; pos<=N/2; pos++){
        ubicarReina(1, pos, tablero);
        determinarPosVálida(conjPosVálida, tablero, 2);
        cantSoluciones+= calcularSoluciones(2, posVálida, tablero)
        }
}//se ubica a la reina de la fila 1 en las primeras N/2 posiciones en base a ello se calculan las posiciones válidas para la reina de la segunda fila y para esas dos reinas ubicadas se terminan la cantidad de soluciones posibles teniendo en cuenta las rotaciones.
```

5.4 Algoritmo paralelo

Para la solución paralela, se selecciona una cantidad de filas que seran utilizada para obtener todas las combinaciones posibles ubicando a las reinas en cada una de las filas en cada una de las columnas. Luego cada procesador es el encargado de resolver un subconjunto de las combinaciones iniciales (cada una será una tarea en la descomposición del problema). Esto trae aparejado dos aspectos, el primero es cómo

construir las combinaciones en base a la cantidad de filas seleccionadas, y el otro es cómo repartir estas combinaciones entre los procesadores.

Para lograr distribuir las tareas en forma equilibrada, de modo de lograr un balance de carga entre los procesadores, es conveniente utilizar una solución de grano fino, esto es, muchas combinaciones de poco cómputo cada una con el objetivo de poder nivelar el trabajo realizado por cada máquina resolviendo varias de estas. Esto se logra utilizando más de una fila para formar cada una de las combinaciones a resolver; por ejemplo, usando las primeras *f* filas se obtienen $W=N^f/2$ combinaciones diferentes para distribuir entre todos los procesadores siendo *N* el tamaño del tablero.

Como se mencionó anteriormente, hay una mejora que se puede hacer teniendo en cuenta las simetrías y rotaciones del tablero. Esta cosiste en ubicar a la reina de la primer fila, solo en la primera mitad de las columnas (*N/2*, parte entera), así, el cálculo de la cantidad de combinaciones se modifica levemente $W=\left(\frac{N}{2}\right) * (N^{f-1})$.

En la figura 5.4 [Fra13] se puede ver un ejemplo en un tablero de 5x5, usando tres filas para formar todas las combinaciones iniciales (incluso las inválidas). En este caso $W=\left(\frac{5}{2}\right) * (5^{3-1})= 50$.

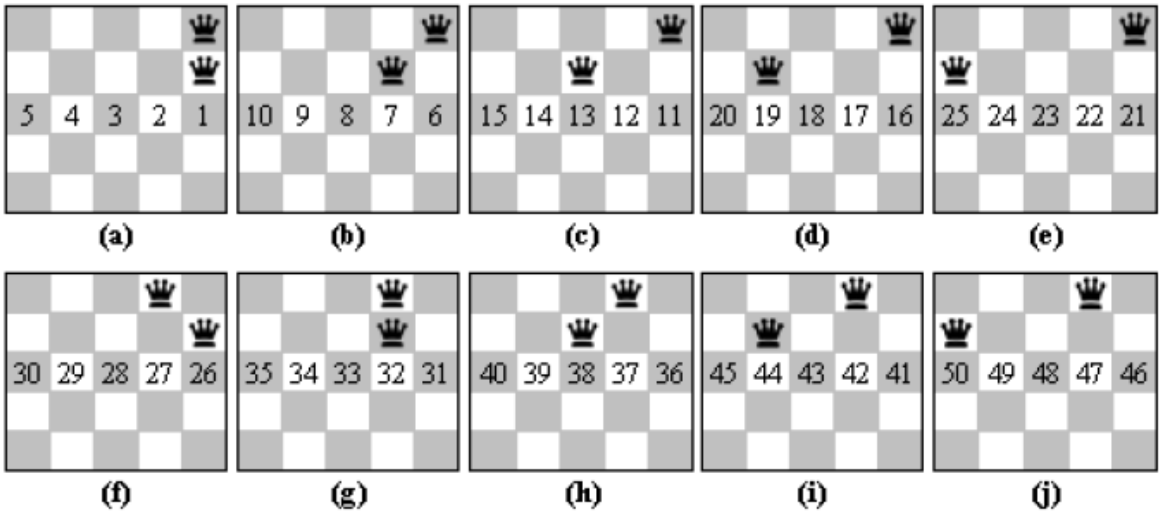


Figura 5.4

5.4.1 Solución con pasaje de mensajes

5.4.1.1 Un nivel de master

Para esta solución se utiliza el modelo Master/Worker clásico, explicado en la sección 3.2.1.1 de esta Tesina. El cual consta de un master principal que reparte tareas entre los workers, a medida que estos resuelven el trabajo asignado.

Para esta solución se utiliza un proceso por cada worker y uno más para el master. Entre ellos se comunican con el envío de mensajes. Para esto se utiliza la librería MPI.

El Master es encargado de calcular la cantidad de combinaciones de acuerdo al número de filas elegido para calcularlas (se recibe como parámetro), y después de esto empieza a recibir pedidos de los workers, respondiéndoles con el envío de una cierta cantidad de combinaciones. Cuando no tiene más combinaciones que repartir, recibe un pedido más

de cada Worker, pero esta vez en lugar de enviar trabajo envía una señalización de fin. Una vez hecho esto, recibe de cada proceso los resultados parciales y los totaliza para dar el resultado final. Esta operación en la práctica fue realizada con la función Reduce provista por MPI, combinando los resultados por medio de una suma.

Cada Worker es encargado de resolver trabajo que ayude a la resolución del problema, en este caso eso sería resolver combinaciones del tablero viendo si son inválidas o no, y en caso de que no lo sean debe calcular cuántas soluciones se desprenden de esa combinación en base a las rotaciones y simetrías. Para hacer esto le pide trabajo al master, y lo resuelve, repitiendo este comportamiento hasta que llegue la señalización de fin, que es cuando cada Worker enviará sus resultados parciales por medio de la operación colectiva reduce.

A continuación se muestra un pseudocódigo que ayuda a entender el algoritmo:

```
Master
main (N, filas){
    cantSoluciones:=0
    calcularCombinaciones (filas, N)

     while (hay combinaciones){
         receive  (Worker);
         calcularTrabajo(combinaciones);
         send (Worker,combinaciones);
      }

     for (i= 1; i<cantWorkers; i++){
          receive(worker)
           send(worker, FIN)
           }
   Reduce (master, cantSoluciones, SUMA);
}
```

```
Worker
main (N, filas){
    cantSoluciones=0

     send  (master);
     receive (master,combinaciones);

     while (NOT FIN){
        analizarComb(cantSolParciales, combinaciones);
        cantSoluciones+=cantSolParciales;
        send  (master);
        receive (master, trabajo);
      }

   Reduce (master, cantSoluciones, SUMA);
}
```

5.4.1.2 Dos niveles de master

Para esta solución se utiliza el modelo Master/Worker con dos niveles, explicado en la sección 3.2.1.2 de esta Tesina.

Para esta solución se utiliza un proceso para el master, otro por cada uno de los master de nivel dos, y otro por cada worker. Entre ellos se comunican con el envío de mensajes. Para esto se utiliza la librería MPI.

El master de nivel uno realiza las mismas funciones que en caso anterior, calcula la cantidad de combinaciones en base al tamaño del tablero y las filas utilizadas, luego reparte las combinaciones a medida que recibe pedidos, cuando no tiene más combinaciones para repartir envía la señalización de fin y finalmente totaliza los resultados. La diferencia es que en este caso, los pedidos los recibe de los masters de nivel dos y no directamente de los workers. Siendo menor la cantidad de procesos que interactúan con él.

Cada master de nivel dos se encarga de recibir pedidos de los workers que tiene asociados mientras tenga trabajo y asignarles combinaciones, cuando se queda sin combinaciones para repartir, envía un pedido al master principal o de nivel uno. Cuando este les envíe la señal de fin, los master de nivel dos replicarán este mensaje a sus workers asociados.

Los Workers desempeñan el mismo papel que en el modelo anterior, sólo que esta vez en lugar de enviar pedidos de trabajo al master de nivel uno, se los envían al correspondiente master de nivel dos.

La relación entre los workers y los master de nivel dos, está dada por los procesos que residen en un mismo multiprocesador, es decir, cada Worker se comunicará con el master de nivel dos que resida en la misma máquina.

A continuación se muestra un pseudocódigo que ayuda a entender la solución:

```
Master Nivel 1
main (N, filas){
    cantSoluciones:=0
    calcularCombinaciones (filas, N)

     while (hay combinaciones){
         receive  (master_n2);
         calcularTrabajo(combinaciones);
         send (master_n2,combinaciones);
      }

     for (i= 1; i<cantMaster_n2; i++){
          receive(master_n2)
           send(mastern2, FIN)
           }
   Reduce (master, cantSoluciones, SUMA);
}
```

```
Worker
main (){
    cantSoluciones=0

     send  (master_n2);
     receive (master_n2,combinaciones);

     while (NOT FIN){
        analizarComb(cantSolParciales, combinaciones);
        cantSoluciones+=cantSolParciales;
        send  (master_n2);
        receive (master_n2, trabajo);
      }

   Reduce (master, cantSoluciones, SUMA);
}
```

```
Master Nivel 2
main (){
  cantSoluciones=0;
    send (master);
    receive  (master);
   while(NOT FIN){
        while (hay combinaciones){
            receive  (worker);
            calcularTrabajo(comb);
            send (worker,comb);
        }
       send (master);
       receive  (master, combiaciones);
  }
     for (i= 1; i<cantWorker; i++){
          receive(worker);
           send(worker, FIN)
           }
   Reduce (master, cantSoluciones, SUMA);
}
```

5.4.2 Solución Hibrida

Para esta solución se utiliza el modelo Master/Worker híbrido, explicado en la sección 3.2.1.3 de esta Tesina. Este consta de un master principal y varios workers, los cuales eventualmente pueden cambiar su rol a master de nivel dos.

Al igual que en las soluciones anteriores el master es un proceso MPI, encargado de calcular la cantidad de combinaciones, luego recibe pedidos respondiendo a estos con el envío de una cierta cantidad de combinaciones. Cuando no tiene más combinaciones que

repartir, envía una señalización de fin. Una vez hecho esto, recibe de cada proceso los resultados parciales y los totaliza para dar el resultado final.

Los workers esta vez son hilos, englobados en distintos procesos. El proceso en sí se encarga de la creación y la administración de los hilos. Los workers esta vez toman combinaciones de un buffer compartido, cuando un Worker descubre el buffer vacío, cambia su rol momentáneamente para pedirle trabajo al master, una vez recibido, almacena el trabajo en el buffer compartido, y vuelve a tomar el rol de Worker, para procesar combinaciones. Una vez recibida la señalización de fin, uno de los hilos de cada proceso totaliza los resultados calculados. Luego se realiza la operación colectiva reduce con este resultado.

Es importante aclarar que los workers pertenecientes al mismo proceso tienen que estar sincronizados para no acceder al mismo tiempo al buffer donde se almacenan las combinaciones y correr el riesgo de dejarlo en estado inconsistente.

El pseudocódigo a continuación muestra cómo deberían comportarse tanto el master como los workers.

```
Master Nivel 1
main (N, filas){
    cantSoluciones=0;
    calcularCombinaciones (filas, N);

     while (hay combinaciones){
         receive  (master_n2);
         calcularTrabajo(combinaciones);
         send (master_n2,combinaciones);
      }

     for (i= 1; i<cantMaster_n2; i++){
          receive(master_n2)
           send(mastern2, FIN)
           }
   Reduce (master, cantSoluciones, SUMA);
}
```

```
Procesos {
        //crear hilos
        //esperar hilos
        Redulce(master, cantSoluciones, SUMA);
}
```

```
main(){ //Hilos
  cantSoluciones:=0
  lock(sección crítica);
  if (soy primero ){
       send(master);
       receive(master, combinaciones);
       move(buffer, combinaciones);
       }
  unlock(sección crítica);
  lock(sección crítica);
  while (NOT FIN){
    if (NO hay trabajo comb. en el buffer){
         send(master);
         receive(master, combinaciones);
         move(buffer, combinaciones);
         unlock(sección crítica);
         }
      else{
         getComb(buffer, comb);
         unlock(sección crítica);
        analizarComb(cantSolParciales, combinaciones);
         cantSoluciones+=cantSolParciales;
          }
       lock(sección crítica);
   }
  unlock(sección crítica);
  lock(sección crítica);
   //totalizar resultados con los de los otros hilos
   unlock(sección crítica);
}
```

Capítulo 6

Problema "Búsqueda de similitud máxima en secuencias de ADN"

6.1 Bioinformática

6.1.1 ¿Qué es?

En los últimos años la biología molecular ha sufrido una revolución debido al desarrollo de técnicas rápidas de "Secuenciamiento de ADN" (DNA Sequencing) y el progreso correspondiente en tecnologías informáticas, lo que permite lidiar con muchísima información de forma cada vez más eficiente. El término "bioinformática" fue acuñado a mediados de la década del 80 para abarcar los desarrollos informáticos aplicados a las ciencias biológicas. Su significado ha sido adaptado por diferentes disciplinas. En un sentido amplio, dentro del término "bioinformática" se consideran todos los desarrollos informáticos aplicados al manejo y análisis de datos biológicos, lo cual tiene implicaciones en áreas muy diversas, que van desde la inteligencia artificial y robótica, hasta el análisis de genomas. En el contexto de iniciativas vinculadas a los genomas, el término se aplicó originalmente a la manipulación y análisis computacional de "secuencias" de datos (ADN o proteínas). [Att99]

6.1.2 ¿Porque es importante?

El desafío central de la bioinformática es la racionalización y análisis de grandes cantidades de secuencias de información, convirtiéndola en conocimiento bioquímico y biofísico, que permita descifrar las pistas estructurales, funcionales y evolutivas que están codificadas en el lenguaje de las secuencias biológicas. Por lo tanto no alcanza con solamente adquirir secuencias, sino que se apunta a entender su conformación interna, para lo cual se aplican métodos computacionales que permitan reconocer patrones o "firmas" dentro de una secuencia, los cuales permiten detectar similitudes entre secuencias y de allí inferir estructuras y funcionalidades relacionadas al identificar cada uno de los componentes que conforman la secuencia. [Att99]

6.2 Descripción del problema

6.2.1 Conceptos preliminares

Una de las preguntas más básicas sobre un gen o proteína es si está relacionado con otro gen o proteína respectivamente. El hecho que se encuentren relacionados al nivel de secuencias sugiere que son "homólogos" y que podrían tener funciones en común. Mediante el análisis de muchas secuencias de ADN, es posible identificar "dominios" o "motifs" (propósitos) que comparten un grupo de moléculas. Este tipo de análisis se logra mediante la alineación de secuencias. [Att99]

6.2.1.1 Secuencias Homólogas

Dos secuencias son homólogas si tienen un ancestro en común en su proceso de evolución. No existen grados de homología, las secuencias o bien son homólogas o no lo son. Así ocurre que cuando dos secuencias son homólogas, sus aminoácidos o secuencias de nucleótidos usualmente comparten una identidad significativa. Así, mientras la homología es una inferencia cualitativa (las secuencias o bien son homólogas o no lo son), la identidad y similitud son cantidades que describen las relaciones entre secuencias. Por ejemplo dos moléculas pueden ser homólogas sin compartir aminoácidos o nucleótidos de forma estadísticamente significativa. Por lo tanto, dos moléculas que son homólogas pueden haber divergido tan fuerte de forma tal que no comparten ninguna identidad reconocible en su secuencia. [Att99]

6.2.1.2 Identidad

En el párrafo anterior surge el concepto de identidad entre dos secuencias. Entonces se llama identidad a la extensión en la que dos secuencias son invariantes. A diferencia del concepto de homología, existen diferentes grados de identidad, lo que permite por ejemplo decir que dos secuencias comparten un 25% de identidad (un caso de ejemplo donde 37 de 145 residuos estén alineados). [Att99]

6.2.1.3 Similitud

Otro aspecto de la alineación de secuencias es que algunos de los residuos alineados son similares pero no idénticos; se encuentran relacionados entre sí porque comparten propiedades biológicas similares. O sea, los pares de residuos que se llaman similares están estructural o funcionalmente relacionados.

La similitud ocurre cuando algunas partes de las secuencias alineadas, son similares pero no idénticas, y estas se encuentran relacionadas entre sí porque comparten propiedades bioquímicas similares. Tal como ocurre en el caso de la identidad, el concepto de similitud admite grados, por lo tanto se podría decir que dos proteínas comparten por ejemplo un 25% de identidad y un 39% de similitud. [Att99].

Se desprende entonces que si la cantidad de identidad en la secuencia es la suficiente, entonces dos secuencias son probablemente homólogas. Sin embargo, nunca es correcto decir que dos secuencias tienen cierto porcentaje de homología, porque como se explicó o bien son homólogas o no lo son. La similitud no es apropiada para describir dos secuencias como “altamente homólogas”; aunque uno sí puede decir que comparten un alto grado de similitud. [Att99]

6.2.2 Alineación de secuencias

La alineación de secuencias (Pairwise Alignment) es el proceso de alinear dos secuencias para alcanzar los niveles máximos de identidad. El propósito de la alineación de pares de secuencias es evaluar el grado de similitud y la posibilidad de que dos moléculas sean homólogas. [Att99]

6.2.2.1 Gaps

La alineación de pares de secuencias es una forma útil de identificar mutaciones que puedan haber ocurrido durante su evolución y haber causado divergencias en las secuencias de dos proteínas, aminoácidos o ADN que se estudian. Las mutaciones más

comunes son "sustituciones", "inserciones" y "borrados". Las inserciones o borrados ocurren cuando se agregan o remueven residuos y se representan típicamente con “guiones” o “líneas” que se agregan a alguna de las dos secuencias (en términos prácticos es una marca que le indica al algoritmo que se trata de uno de estos casos). Las inserciones o eliminaciones (inclusive aquellas de un carácter de longitud) reciben el nombre de gap en la alineación. Notar que uno de los efectos de agregar gaps es hacer que la longitud total de cada alineación sea exactamente la misma. El agregado de un gap ayuda a crear una alineación que modela los cambios evolutivos que puedan haber ocurrido. En un esquema típico de "scoring" hay dos tipos de penalidades de gap: una por crear un gap y otra por cada residuo adicional que extiende un gap. [Att99]

La alineación de secuencias es extremadamente difícil de lograr mediante la inspección visual. Además, si se permiten gaps en la alineación para considerar eliminaciones o inserciones que puedan haber sucedido en las dos secuencias, el número posible de alineaciones crece exponencialmente. Es por esta razón que surge la necesidad de usar un algoritmo para realizar la alineación de ADN. [Att99]

6.2.2.2 Comparando dos secuencias un caso simple

Se puede determinar la similitud entre dos secuencias, cada una elegida de un alfabeto de complejidad *N*. Por ejemplo en el caso de secuencias de ADN el alfabeto es de 4 caracteres: *A*, *T*, *G* y *C* (Adenina, Timina, Citosina y Guanina). El enfoque más simple es alinear cada una de las secuencias contra las demás e insertar caracteres adicionales para que los strings queden alineados verticalmente. [Att99]

En la figura 6.1 se ilustra del uso del carácter gap “-” para hacer que dos secuencias queden alineadas; las barras verticales denotan coincidencias idénticas (seis en el primer alineamiento, nueve en el segundo).

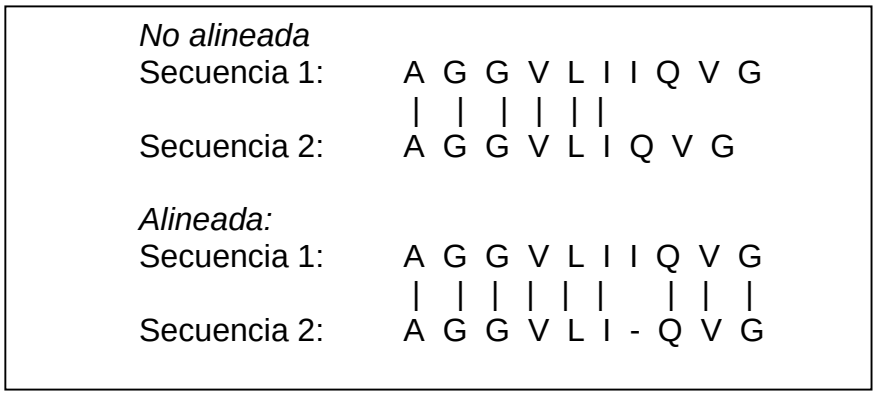


Figura 6.1

Una vez que se terminan de alinear las secuencias se puede otorgar un puntaje a la alineación contando cuántas posiciones coinciden de forma idéntica en cada posición, en el ejemplo el puntaje sin alinear es seis y una vez alineado es nueve.

Este ejemplo simple sirve para mostrar cómo el puntaje se incrementa cuando se alinean más residuos idénticos. Sin embargo, esto es simplemente ilustrativo ya que las secuencias usadas son muy cortas (la mayoría de las secuencias de proteínas tienen entre 200 y 500 residuos de longitud, inclusive mas), son casi del mismo tamaño (lo cual es muy raro en ejemplos reales) y las secuencias son casi idénticas (es imposible obtener otra alineación de residuos que alcance un puntaje mayor).

El proceso de alineación puede medirse teniendo en cuenta los gaps introducidos y la cantidad de posiciones que no coinciden en la alineación (mismatches). [Att99]

6.2.3 Subsecuencias

El ejemplo que se acaba de ver es muy simple y se encuentra alejado de lo que ocurre en la práctica. Si se tienen dos secuencias de ADN, la *secuencia A* de 400 residuos de longitud y la *B* de 650. Si la *secuencia A* es totalmente idéntica a cualquier porción de la *secuencia B*, se dice que *A* es una *subsecuencia de B*. Si este fuera el caso simplemente basta con insertar los GAPS necesarios para alinear *A* dentro de *B*, como se puede ver en la figura 6.2.

A - ————————————
B ————————————————————————————

Se alinea la subsecuencia A con la secuencia completa B. En este caso A es completamente idéntica a una parte de B, entonces la inserción de un bloque de gaps permite completar la alineación de las dos secuencias.

Figura 6.2

Supóngase ahora el caso en que *A* tiene dos regiones que muestran identidad con la *secuencia B*. En este caso es necesario identificar ambas regiones y luego insertar GAPS dentro de *A* para lograr una alineación con *B*, como se puede ver en la figura 6.3.

A - - - - - ———— - - - - - - - - - - - - - - - ————————
B ————————————————————————————

Se alinea la subsecuencia A con la secuencia completa B. En este caso A es idéntica a diferentes partes de B, de forma tal que sea necesaria la inserción de más de un bloque de gaps para completar la alineación de las dos secuencias.

Figura 6.3

Sin embargo, el alineamiento no se limita solamente a coincidir una subsecuencia, sino que involucra la comparación de secuencias completas. Una alineación exhaustiva debe tomar en cuenta todas las posiciones de todos los residuos en ambas secuencias. Lo que tendrá como consecuencia que muchos residuos tengan que ubicarse en posiciones en las cuales no son estrictamente idénticos. En este caso, colocar gaps en la alineación se vuelve más difícil de computar. Por eso se podrían insertar gaps de forma indiscriminada con tal de maximizar el puntaje obtenido. Si bien esto permitiría alcanzar un puntaje óptimo, el resultado de dicho proceso no tendría ningún sentido desde el punto de vista biológico. Es así que se introduce el concepto de "penalidad" para minimizar el número de gaps que se abren y si el gap debiera ser extendido las penalidades se extienden también. De aquí surge entonces una función de identidad entre dos secuencias y las penalidades por abrir un gap, la cual determina el puntaje total.

Para decidir el puntaje que hay que otorgarle a una coincidencia se usa una matriz de puntaje. Por una cuestión de simplicidad en el proceso que se describió anteriormente se usó una matriz "esparcida" (sparse), ya que la mayoría de sus elementos son cero y las coincidencias tienen todos los mismos puntajes, lo cual hace que todas las coincidencias tengan el mismo peso bajando así la performance de diagnóstico de la matriz. Es por eso que en casos reales se usan diferentes tipos de matrices que mejoren el puntaje de las potenciales coincidencias aunque estas sean débiles pero lo más importante es que tengan significado desde un punto de vista biológico y no desde un punto de vista meramente matemático, logrando así mejorar el puntaje sin amplificar el ruido. [Att99]

6.2.4 Similitud local y global

Los alineamientos son simplemente modelos matemáticos cuyo comportamiento puede ser influenciado por el uso de parámetros como por ejemplo la matriz de puntaje. Por esta razón es que existen diferentes modelos, los cuales fueron diseñados para encapsular una variedad de características físicas de las secuencias, como su relación funcional, estructural o evolutiva. Es en este contexto que se debe entender entonces que no existe un alineamiento correcto o incorrecto, sino diferentes modelos biológicos que reflejan diferentes perspectivas biológicas. [Att99]

En el modelo de similitud global se considera la similitud a lo largo de toda la secuencia, en cambio el segundo se enfoca en encontrar regiones de similitud en partes de la secuencia solamente. Es importante entender la diferencia entre los dos modelos, ya que si las secuencias a comparar no tienen una similitud uniforme en toda su extensión, no tiene ningún sentido hacer un alineamiento global si solamente tienen similitud local.

Entonces el objetivo de hacer una alineación local es que los sitios funcionales se localizan en regiones cortas y estos en general se encuentran relativamente conservados de mutaciones o borrados que podrían haber sufrido otras partes de la secuencia. Todo esto termina dando como resultado una búsqueda con mucho más sentido biológico y resultados más exactos que los que se podrían lograr al realizar un alineamiento global de toda la secuencia.

En 1981, Smith y Waterman describieron un algoritmo que permite otorgarle puntaje a pequeñas regiones con similitud local. El método de Smith-Waterman ha sido usado como base de otros algoritmos y se lo suele citar como el algoritmo de referencia a la hora de comparar diferentes técnicas de alineamiento. El algoritmo utiliza un enfoque matricial y se usa el retroceso para reconstruir los alineamientos que contienen gaps. Es ciertamente una técnica sensible, debe tenerse presente al usar cualquier implementación de la misma, que la función del algoritmo es hallar pequeñas regiones con similitud local. [Att99]

Se utiliza una matriz para calcular las similitudes de las secuencias. El algoritmo original calcula el puntaje de similitud entre dos secuencias y luego emplea un retroceso para crear la alineación óptima. Esta parte no es necesaria en aquellos casos en que solo se requiere el puntaje de similitud como son las búsquedas en Bases de Datos. Como no se realiza retroceso en el algoritmo, no es necesario almacenar la matriz *H* completa.

6.2.5 Algoritmo Smith-Waterman

A continuación se realiza una explicación de alto nivel sobre el funcionamiento del algoritmo.

Dadas dos secuencias *A* y *B*:

$$A = a_1a_2a_3...a_M \qquad B = b_1b_2b_3...b_N$$

Se construye la matriz *H* de (*N+1*)x(*M+1*), de tal forma que las bases de nucleótidos que forman la *secuencia A* etiquetan las filas (a partir de la 1) y los de *B* las columnas (a partir de la 1). Luego se calculan los valores de *H* que darán el puntaje de similitud entre *A* y *B*.

1) Se inicializan la fila y la columna en cero.

$$H_{i,0} = H_{0,j} = 0 \qquad \text{para } 0 \leq i \leq N \text{ y } 0 \leq j \leq M \qquad (6.1)$$

2) Se calcula el valor de H_{ij} para todo $i \in [1..N]$ y para todo $j \in [1..M]$ por medio de la siguiente ecuación. Este valor indica la máxima similitud entre dos segmentos que terminan en a_i y b_j respectivamente.

$$H_{ij} = \max \begin{cases} 0 \\ H_{i-1,j-1} + V(a_i,b_j) \\ C_{ij} \\ F_{ij} \end{cases} \qquad (6.2)$$

Donde $V(a_i, b_j)$ es la función de coincidencias que indica el puntaje dado por hacer coincidir a a_i y b_j. Esta función está basada en una tabla de valores llamada matriz de sustitución que representa la posibilidad de que la base nucléotida de la *secuencia A* en la posición *i*, pueda ser sustituida por la base de la secuencia *B* en la posición *j*. La matriz más común es la que premia con un valor positivo cuando a_i y b_j son idénticos y castiga con un valor negativo en caso contrario. Se puede ver un ejemplo de matriz en la tabla 6.1.

	A	**C**	**G**	**T**
A	3	-1	-1	-1
C	-1	3	-1	-1
G	-1	-1	3	-1
T	-1	-1	-1	3

Tabla 6.1

C_{ij} es el puntaje considerando un gap en la columna *j*, y se calcula por medio de la ecuación 6.3 que se muestra a continuación.

$$C_{ij} = \max_{1 \leq k \leq i} \{H_{i-k,j} - g(k)\} \qquad (6.3)$$

F_{ij} es el puntaje considerando un gap en la fila *i*, y se calcula por medio de la ecuación 6.4 que se encuentra a continuación.

$$F_{ij} = \max_{1 \leq p \leq j} \{H_{i,j-p} - g(p)\} \qquad (6.4)$$

$g(x)$ es la función de penalidad por un gap de longitud *x*, y se obtiene por medio de la ecuación 6.5 que se encuentra a continuación (siendo *q* la penalidad por la apertura de un gap y *r* por la prolongación del mismo).

$$g(x) = q + r\,x \; (q \geq 0; r \geq 0) \qquad (6.5)$$

3) En el siguiente paso, se obtiene el puntaje de similitud como se indica en la ecuación 6.6, la cual se puede encontrar a continuación.

$$G = \max_{(0 \leq i \leq N)(0 \leq j \leq M)} \{H_{ij}\} \qquad (6.6)$$

4) En caso que se quiera obtener el par de segmentos con máxima similitud es necesario realizar un proceso de retroceso a partir de la posición donde se encontró el

valor *G* (que representa el final del alineamiento de puntuación más elevada entre las dos secuencias) en la matriz *H*, hasta que se llega a una posición cuyo valor es 0, siendo este punto el inicio del segmento. Es importante destacar que este paso es necesario solamente para obtener los segmentos con máxima similitud, no es necesario si solamente se quiere obtener el puntaje de similitud máximo.

En la figura 6.3 se puede ver la aplicación del algoritmo Smith-Waterman a las *secuencias A* y *B*.

En este caso el puntaje de similitud es 3,33 y el par de segmentos con máxima similitud es:

Secuencia 1	G	C	C	-	U	C	G
Secuencia 2	G	C	C	A	U	U	G

	S2	C	A	G	C	C	U	C	G	C	U	U	A	G
S1	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
A	0,00	0,00	1,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	0,00
A	0,00	0,00	1,00	0.66	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	0.66
U	0,00	0,00	0,00	0.66	0.33	0,00	1,00	0,00	0,00	0,00	1,00	1,00	0,00	0.66
G	0,00	0,00	0,00	1,00	0.33	0,00	0,00	0.66	1,00	0,00	0,00	0.66	0.66	1,00
C	0,00	1,00	0,00	0,00	2,00	1.33	0.33	1,00	0.33	2,00	0.66	0.33	0.33	0.33
C	0,00	1,00	0.66	0,00	1,00	3,00	1.66	1.33	1,00	1.33	1.66	0.33	0,00	0,00
A	0,00	0,00	2,00	0.66	0.33	1.66	2.66	1.33	1,00	0.66	1,00	1.33	1.33	0,00
U	0,00	0,00	0.66	1.66	0.33	1.33	2.66	2.33	1,00	0.33	1.66	2,00	1,00	1,00
U	0,00	0,00	0.33	0.33	1.33	1,00	2.33	2.33	2,00	0.66	1.66	2.66	1.66	1,00
G	0,00	0,00	0,00	1.33	0,00	1,00	1,00	2,00	3.33	2,00	1.66	1.33	2.33	2.66
A	0,00	0,00	1,00	0,00	1,00	0.33	0.66	0.66	2,00	3,00	1.66	1.33	2.33	2,00
C	0,00	1,00	0,00	0.66	1,00	2,00	0.66	1.66	1.66	3,00	2.66	1.33	1,00	2,00
G	0,00	0,00	0.66	1,00	0.33	0.66	1.66	0.33	2,66	1.66	2.66	2.33	1,00	2,00
G	0,00	0,00	0,00	1.66	0.66	0.33	0.33	1.33	1.33	2.33	1.33	2.33	2,00	2,00

Figura 6.3

6.3 Algoritmo secuencial

El objetivo del problema "Búsqueda de máxima similitud en Bases de Datos de secuencias de ADN" utilizado en esta tesina es el siguiente: dada una secuencia de ADN que será llamada test, y una base de secuencias de ADN, encontrar la secuencia de la base que tiene mayor similitud con la secuencia test.

Para lograr esto se utiliza el algoritmo Smith-Waterman con el objetivo de determinar el puntaje de similitud entre dos secuencias de ADN, lo que significa que no se tiene en cuenta el proceso de retroceso para obtener el segmento que representa la alineación óptima (no se realiza el punto 4 del algoritmo explicado en la sección 6.2.5). Este algoritmo se explica en primera instancia ya que se utiliza como base para resolver el problema de Búsqueda de máxima similitud en Bases de Datos de secuencias de ADN que se trata en este capítulo.

Existe dependencia de datos para calcular los valores de la matriz. Para obtener $H_{i,j}$ se requiere el resultado de $H_{i-1,\ j-1}$ (H_d en la figura 6.4) y se necesita saber el puntaje al considerar un gap en la fila *i* y otro en la columna *j*. Esta relación lleva a calcular los valores de *H* de arriba hacia abajo y de izquierda a derecha (H_{11}, H_{12}, H_{13}, … H_{21}, H_{22}, H_{23}, …).

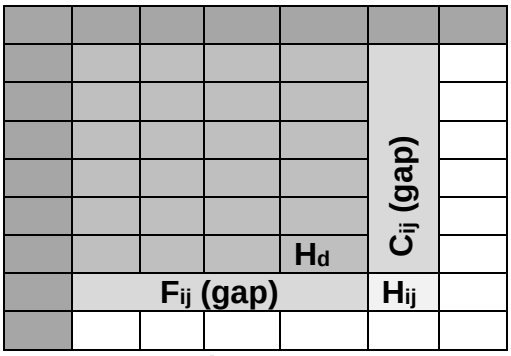


Figura 6.4

Como no se realiza el paso 4 del algoritmo explicado en la sección anterior, no es necesario almacenar la matriz *H* completa, en su lugar se necesita:

- Un vector *h* de longitud *M+1* que mantiene en cada posición el valor obtenido en la última fila procesada sobre esa columna. En la ecuación 6.7 que se encuentra a continuación se indican los valores de h en el ejemplo de la figura anterior.

$$h_k = \begin{cases} H_{i,k} & k < j-1 \\ H_{i-1,k} & k \geq j-1 \end{cases} \qquad (6.7)$$

- Un elemento *e* para guardar en forma temporal el último valor calculado en la fila que se está procesando. En la figura anterior, $e = H_{i,j-1}$.

- Un vector *c* de longitud *M+1* que mantiene en cada posición el máximo puntaje considerando un gap en esa columna. En la ecuación 6.8, que se encuentra a continuación, se indican los valores de *c* en el ejemplo de la figura anterior.

$$c_k = \begin{cases} C_{i,k} & k < j \\ C_{i-1,k} & k \geq j \end{cases} \qquad (6.8)$$

- Un elemento *f* que mantiene el máximo puntaje considerando un gap en la fila que se está procesando. En el ejemplo de la figura anterior, $f = F_{i,j-1}$.

Como se mencionó al principio de la sección el problema principal a abordar es el problema de búsqueda de máxima similitud en “Bases de Datos de secuencias de ADN”. Actualmente se cuenta con numerosas Bases de Datos con información relacionada con las secuencias de ADN. Cada una de estas bases contiene miles de registros de secuencias de ADN y se encuentra en constante crecimiento.

El propósito de las bases es registrar propiedades específicas de las secuencias de ADN para ser utilizadas en diferentes aplicaciones. Con esta información se pueden predecir características de una nueva secuencia que se vaya a comparar contra todas las almacenadas en la base de datos, al realizar una búsqueda de aquellas secuencias que cumplen con un cierto criterio de similitud con la nueva secuencia.

Este problema puede considerarse como una extensión del alineamiento entre pares de secuencias descripto anteriormente, donde una nueva secuencia se compara frente a

una base de datos de muchas miles de secuencias. Llevar a cabo esta búsqueda en una base de datos de secuencias de ADN de forma eficaz no es un problema trivial ya que a medida que los conjuntos de datos crecen, más esfuerzo se dedica a mejorar la eficacia. En el caso de base de datos grandes y numerosas secuencias, efectuar un alineamiento de Smith-Waterman resulta prohibitivo por el tiempo consumido.

La velocidad de ejecución es un aspecto importante a tener en cuenta en este problema y en el algoritmo que se describe, la misma depende en gran parte de la longitud de la nueva secuencia a comparar y del tamaño (con respecto a cantidad y longitud de secuencias) de la base de datos frente a la que se la compara. [ATT02]

Las bases de datos utilizadas están compuestas por secuencias de diferentes tamaños, se representan por arreglos de caracteres y su alfabeto está constituido por las 4 bases de nucleótidos (*A*, *T*, *C*, *G*), donde la Adenina solo puede estar unida a la Timina y la Citosina solamente a la Guanina. Se tiene así una base de datos formada por *K* secuencias almacenadas en un archivo de texto plano y en otro archivo separado que contiene una única secuencia se encuentra la nueva secuencia (*test*) a comparar frente al archivo de base de dato.

Para este trabajo la base de datos es generada artificialmente de forma aleatoria donde se puede especificar la cantidad de secuencias (*K*), a generar; el tamaño base de cada secuencia; el porcentaje de variación en la longitud de las secuencias y finalmente permite especificar una semilla para generar mayor aleatoriedad en el archivo.

El algoritmo secuencial carga la secuencia test almacenada en un archivo, en un arreglo; luego comienza a iterar sobre el archivo de base de datos de secuencias de ADN y para cada secuencia S_i la carga en un arreglo y obtiene el *Puntaje de Similitud* (G_i) entre test y S_i por medio del algoritmo Smith-Waterman explicado anteriormente (sin la realización del punto 4 que comprende el retroceso para obtener el segmento de mayor similitud, ya que solamente resulta de interes el puntaje y no el segmento), se guarda en G_{max} el Puntaje de Similitud máximo y en S_{max} el número de secuencia con el que se obtuvo ese valor y a medida que se avanza en el archivo de base de datos y se encuentra un nuevo máximo se actualizan ambos valores. A continuación se presenta un pseudocódigo que ayuda a entender el algoritmo utilizado:

```
main (){
        similitudMaxima=-1;
        secuenciaMáxima=0;
        leerTest(test, archivo_test);
        for(sec=0;sec<cantSec; sec++){
                leerSecuencia(secuencia, archivo_secuencias);
                compararSecuencias(similitud, test, secuencia);
                if(similitudeMáxima<similitud){
                        similitudMáxima=similitud;
                        secuenciaMáxima=secuencia;}
                }
}
```

6.4 Algoritmo paralelo

Las secuencias de ADN pueden estar formadas por un gran número de bases nucleótidos (en el orden de 10^9). Es por esto que al momento de realizar una alineación de dos secuencias el tiempo y el espacio requerido son excesivos cuando se realiza de forma secuencial, surgiendo así la necesidad de paralelizar el algoritmo.

Como se mencionó anteriormente, cada secuencia puede tener un tamaño distinto, dependiendo de los parámetros utilizados para la generación de la base. Es por eso que cada tarea puede variar en el tiempo de ejecución en la longitud de la secuencia utilizada.

Al analizar el problema a resolver y la infraestructura con la que se cuenta se decide resolver el problema con el paradigma master-worker usando tres enfoques diferentes. Paralelizando la cantidad de secuencias de la base que son comparadas contra la secuencia test al mismo tiempo, y no la comparación de dos secuencias específicas, como se podría hacer con el algoritmo Smith-Waterman. En todos los casos analizados, el master se encarga de repartir entre los workers, las diferentes secuencias de la base, considerando cada una como una tarea diferente relativamente independiente, y generando interacción al momento de calcular la máxima similitud.

6.4.1 Solución con pasaje de mensajes

6.4.1.1 Un nivel de master

Para esta solución se utiliza el modelo uno, explicado en la sección 3.2.1.1 de esta Tesina. El cual consta de un master principal que reparte tareas entre los workers, a medida que estos resuelven el trabajo asignado.

En esta solución se utiliza la librería MPI, teniendo un proceso para el master, y otro por cada uno de los workers. La comunicación entre el master y cada uno de los workers, se realiza mediante el pasaje de mensajes.

En una primera instancia el master realiza una operación colectiva para distribuir la secuencia test a cada uno de los workers, esta operación es conocida como Broadcast, y es provista por la libraría MPI. Además, se tiene la opción de hacer una carga de trabajo inicial a cada Worker, que puede ser un porcentaje pequeño de la base de datos; dicho porcentaje se envía como parámetro cuando se ejecuta el programa, junto con otros, tales como: el nombre del archivo que contiene la secuencia test, el nombre del archivo que contiene la base de datos y la cantidad de secuencias que posee la base.

Una vez hecho esto, el master permanece en una iteración, en la cual recibe un pedido de trabajo y envía una cantidad determinada de secuencias al Worker que le envió el pedido, así hasta que no tenga más secuencias para repartir. Después que llega al final del archivo de secuencias, recibe un pedido más de trabajo de cada Worker, pero esta vez no envía secuencias, sino que envía una señalización de fin. Luego recibe los resultados parciales de cada Worker, y los combina para obtener la secuencia que tuvo la similitud máxima.

Por su lado, cada Worker, permanece en una iteración hasta que el master le envíe la señal de fin. En cada iteración le pide secuencias al master, y para cada una realiza la comparación con la secuencia test mediante el algoritmo Smith-Waterman, calculando si el resultado obtenido es el máximo local o no. Una vez que recibe la señalización de fin, el Worker envía su máximo local al master y finaliza su ejecución.

A continuación se presenta pseudocódigo que ayuda a entender este algoritmo:

```
//Master
main (archivo_test, archivo_secuencias, cantSec){

        similitudMaxima=-1;
        secuenciaMáxima=0;
        leerTest(test, archivo_test);
        cant=x //x es el número de secuencias por pedido
        broadcast (test);
        sec=0;
        while(sec<cantSec){
                receive(worker); //recibe pedido
                leerSecuencias(archivo_secuencias, cant);
               send(worker, secuencias); //envía trabajo
               sec+=cant;
                       }
       for(i=0; i<cantWorkers; i++){
               receive(worker);
               send(worker, señal_fin)
               }
       for(i=0; i<cantWorkers; i++){
               receive(worker, sec_max_parcial, sim_max_parcial);
               if (sim_max_parcial>similitudMáxima){
                       similitudMáxima=sim_max_parcial;
                       secuenciaMáxima=sec_max_parcial;
                                }
               }
       Print(similitudMáxima, secuenciaMáxima);
}
```

```
//Worker
Main(){
        sim_max_parcial=-1;
        Broadcast(test);
        Send(master);
        Receive(master, secuencias);
        While(  secuencias != Fin){
           While(secuancias<>NULL){
                tomarUna(secuencias, secuencia);
                CompararSecuencias(test, secuencia, similitud, secuencia);
                If(similitud>sim_max_parcial){
                      //actualiza máximos
                      }
                Send(master);
                Receive(master, trabajo);
                }
           }
           Send(master, sec_max_parcial, sim_max_parcial);
}
```

6.4.1.2 Dos niveles de master

Para esta solución se utiliza el modelo dos, explicado en la sección 3.2.1.2 de esta Tesina. El cual consta de un master principal que reparte tareas entre los masters de segundo nivel, los cuales a su vez reparten el trabajo entre los workers asociados a medida que estos resuelven el trabajo asignado. Como se mencionó anteriormente, por

cada multiprocesador se tiene un proceso master de nivel dos en un núcleo, y un proceso Worker en cada uno de los núcleos restantes, excepto en uno de los multiprocesadores que sólo se tiene un proceso que hará el rol de master principal.

En esta solución se usa la libraría MPI, teniendo un proceso para el master, y otro por cada uno de los masters de nivel dos y los workers.

En una primera instancia el master de nivel uno realiza una operación colectiva (broadcast) para distribuir la secuencia test a cada uno de los workers. Además, se tiene la opción de hacer una carga de trabajo inicial a cada master de nivel dos, que puede ser un porcentaje pequeño de la base de datos; dicho porcentaje se envía como parámetro cuando se ejecuta el programa, junto con otros, tales como: el nombre del archivo que contiene la secuencia test, el nombre del archivo que contiene la base y la cantidad de secuencias que posee la base.

Una vez hecho esto el master de nivel uno permanece en una iteración, en la cual recibe un pedido de trabajo y envía una cantidad determinada de secuencias al master de nivel dos que le envió el pedido hasta que no tenga más secuencias que repartir. Después que llega al final del archivo de secuencias, recibe un pedido más de trabajo de cada master de segundo nivel, pero esta vez no envía secuencias, sino que envía una señalización de fin. Luego recibe los resultados parciales de cada master de nivel dos, y los combina para obtener la secuencia que tuvo la similitud máxima.

Cada master de segundo nivel permanece en una iteración hasta que el master le envíe la señal de fin. En cada iteración realiza dos importantes tareas, la primera es recibir pedidos de los workers asociados y enviarles trabajo. Además cuando no tiene más trabajo para repartir, envía un pedido al master de nivel uno, y vuelve a repetir la iteración. Una vez que recibe la señalización de fin, termina su procesamiento.

Por su lado, cada Worker, permanece en una iteración hasta que el master asociado le envíe la señal de fin. En cada iteración le pide secuencias al master de nivel dos correspondiente, y para cada una realiza la comparación con la secuencia test mediante el algoritmo Smith-Waterman, calculando si el resultado obtenido es el máximo local o no. Una vez que recibe la señalización de fin, el Worker envía su máximo local al master y finaliza su ejecución.

```
//Master
main (archivo_test, archivo_secuencias, cantSec){

        similitudMaxima=-1;
        secuenciaMáxima=0;
        leerTest(test, archivo_test);
        cant=x //x es el número de secuencias por pedido
        broadcast (test);
        sec=0;
        while(sec<cantSec){
                receive(master_n2); //recibe pedido
                leerSecuencias(archivo_secuencias, cant);
               send(master_n2, secuencias); //envía trabajo
               sec+=cant;
                     }
       for(i=0; i< cantMaster_n2; i++){
               receive(master_n2);
               send(master_n2, señal_fin)
               }
       //Recibir resultados y calcular el máximo global
       Print(similitudMáxima, secuenciaMáxima);
}
```

```
//Master nivel 2
main (){
        broadcast (test);
        Send(master);
        Receive(master, secuencias);
        while(secuencias!=Fin){
               //reparte el trabajo entre todos los workers
                while(hay_trabajo){
                      receive(worker); //recibe pedido
                        send(Worker, trabajo); //envía trabajo}
                //pide más trabajo al master de nivel uno
                 Send(master);
                Receive(master, secuencias);}
       for(i=0; i< cantWorkersAsociacod; i++){
               receive(worker);
               send(worker, señal_fin)}
}
```

```
//Worker
Main(){
        sim_max_parcial=-1;
        Broadcast(test);
        Send(master_n2);
        Receive(master_n2, secuencias);
        While(  secuencias != Fin){
           While(secuancias<>NULL){
                tomarUna(secuencias, secuencia);
                CompararSecuencias(test, secuencia, similitud, secuencia);
                If(similitud>sim_max_parcial){
                     //actualiza máximos
                     }
                Send(master_n2);
                Receive(master_n2, trabajo);
                }
           }
           //enviar máximo parcial
}
```

6.4.2 Solución Hibrida

Para esta solución se utiliza el modelo uno, explicado en la sección 3.2.1.3 de esta Tesina. El cual consta de un master principal y varios workers, que eventualmente pueden asumir el rol de master de nivel dos.

En este caso hay un proceso MPI para el master principal, y luego por cada multiprocesador se tienen tantos hilos como núcleos haya; englobados en un proceso MPI que no tiene más funcionalidad que el manejo de dichos threads.

En una primera instancia el master realiza una operación colectiva para distribuir la secuencia test, esta vez un hilo por cada proceso MPI se hará cargo de recibir esta secuencia y alojarla en un buffer compartido de modo que el resto de los hilos de ese proceso puedan accederla. Luego permanece en una iteración hasta que no tenga más secuencias que repartir; en cada iteración recibe un pedido de trabajo y envía una cantidad determinada de secuencias al proceso que le envío el pedido.

Cada Worker (hilo) toma una secuencia del buffer compartido, hace el cálculo de similitud correspondiente, y calcula su máximo local. Cuando un Worker quiere sacar una secuencia del buffer compartido y se encuentra con que este está vacío, su rol se ve alterado haciendo la tarea de un master de segundo nivel al enviarle un pedido al master principal y recibir trabajo de este, alojándolo en el buffer para luego volver a tomar su rol como Worker para ejecutar trabajo. Cuando el buffer está vacío, y en lugar de trabajo se recibe la señalización de fin, uno de los hilos calcula el máximo parcial entre todos los workers pertenecientes al mismo proceso MPI.

A continuación un pseudocódigo de la solución:

```
//Master
main (archivo_test, archivo_secuencias, cantSec){

        similitudMaxima=-1;
        secuenciaMáxima=0;
        leerTest(test, archivo_test);
        cant=x //x es el número de secuencias por pedido
        broadcast (test);
        sec=0;
        while(sec<cantSec){
                receive(master_n2); //recibe pedido
                leerSecuencias(archivo_secuencias, cant);
               send(master_n2, secuencias); //envía trabajo
               sec+=cant;
                      }
       for(i=0; i< cantMaster_n2; i++){
               receive(master_n2);
               send(master_n2, señal_fin)
               }
       //Recibir resultados y calcular el máximo global
        Print(similitudMáxima, secuenciaMáxima);
}
```

```
//Worker
Hilo(){
        sim_max_parcial=-1;
       lock(seccion_critica);
        If (primero){
            Broadcast(test);
            Send(master);
            Receive(master, secuencias);
            Buffer=secuencias;
            }
       lock(seccion_critica);

        lock(seccion_critica);
        sacar_sec(buffer, secuencia);
        While(  NOT Fin){
                If( isEmpty(buffer)){
                      send(master);
                      receive(master, secuencias);
                     buffer=secuencias;
                     unlock(seccion_critica);
                }
                else{
                      getSecuencia(buffer);
                      unlock(sección_crítica);
                      CompararSecuencias(test, secuencia, similitud, secuencia);
                       If(similitud>sim_max_parcial){
                              //actualiza máximos
                       }
                 }
                 lock(sección crítica);
            }
            unlock(seccion_critica);

            lock(seccion_critica);
                     //comparar similitud máxima contra el resto
                    //enviar el resultado al master
            unlock(seccion_critica);
}
```

```
Procesos
Main(){
       //crear hilos
       //esperar hilos
      //enviar resultados al master
}
```

Capítulo 7

Experimentación

En esta sección se describen las pruebas y resultados obtenidos para analizar las diferentes soluciones implementadas para ambos problemas, así como el entorno que se utilizó para la resolución. En primera instancia se comparan las tres soluciones para el problema de N-reinas y luego se hace lo propio para el problema de "Búsqueda de similitud máxima en secuencias de base de datos de ADN".

7.1 Arquitectura utilizada

La experimentación se realizó sobre una arquitectura homogénea con 16 computadoras pertenecientes al Aula de Postgrado de la Facultad de Informática de la UNLP con las siguientes características:

Las computadoras están compuestas por procesadores i5 2300 con 4 cores físicos (sin hyperthreading) y 8 GB de memoria RAM.

La comunicación entre las computadoras se realiza por medio de una red Ethernet de 1Gbit.

El lenguaje utilizado para las implementaciones es C junto con la librería MPI para manejar las comunicaciones entre procesos y la librería Pthreads en el caso de los hilos.

7.2 N-reinas

En la experimentación, se utilizaron tableros de tamaño *N*=17, 18, 19, 20 y 21. En la tabla 7.1 se puede ver el tiempo secuencial para cada tamaño de tablero, junto con la cantidad de soluciones posibles para ese tamaño de tablero:

Tamaño del tablero	Tiempo de ejecución	Cantidad de soluciones
17	28,930	95815104
18	211,179	666090624
19	1620,609	4968057848
20	13114,126	39029188884
21	111467,299	314666222712

Tabla 7.1

Cada uno de los tiempos de ejecución presentados en la tabla 7.1, es resultado del promedio de los tiempos de cinco ejecuciones diferentes dada la misma entrada.

A continuación se hará un análisis desde dos puntos de vista distintos. El primero consiste en evaluar la eficiencia de las distintas soluciones para distintos tamaños de tablero a modo de poder realizar una comparación entre ellos; y por otro lado analizando el comportamiento cuando el tamaño del problema se mantiene, pero se aumentan las unidades de procesamiento que se utilizan para resolverlo. Los resultados expuestos fueron elegidos dado que se los considera los más representativos.

7.2.1 Comparación de las tres soluciones

Los tamaños de los tableros utilizados son los mencionados anteriormente. En todas las pruebas se utilizan las cuatro primeras filas del tablero para formar las combinaciones, las cuales serán las tareas a repartir entre los "workers". Además de la variación en el tamaño de tablero, para cada *N*, se probó con diferentes porcentajes de reparto inicial los cuales son 10%, 20% y 40%.

En la tabla 7.2 se pueden ver los resultados obtenidos y las métricas calculadas para las tres soluciones, utilizando 12 máquinas para cada uno de los tamaños del tablero *N*=17, 18, 19. 20, 21 si se utiliza como porcentaje de reparto inicial el 20% de las combinaciones.

La tabla 7.1 refleja la eficiencia de cada solución para los distintos tamaños del tablero y en base a la tabla se presenta un gráfico.

	Un Master	Híbrido	Dos master
17	0,78472805	0,55556188	0,54858578
18	0,92203129	0,72095893	0,64994471
19	0,94310175	0,78070922	0,66277181
20	0,95229326	0,78760363	0,66902300
21	0,96131775	0,78760363	0,67412538

Tabla 7.2

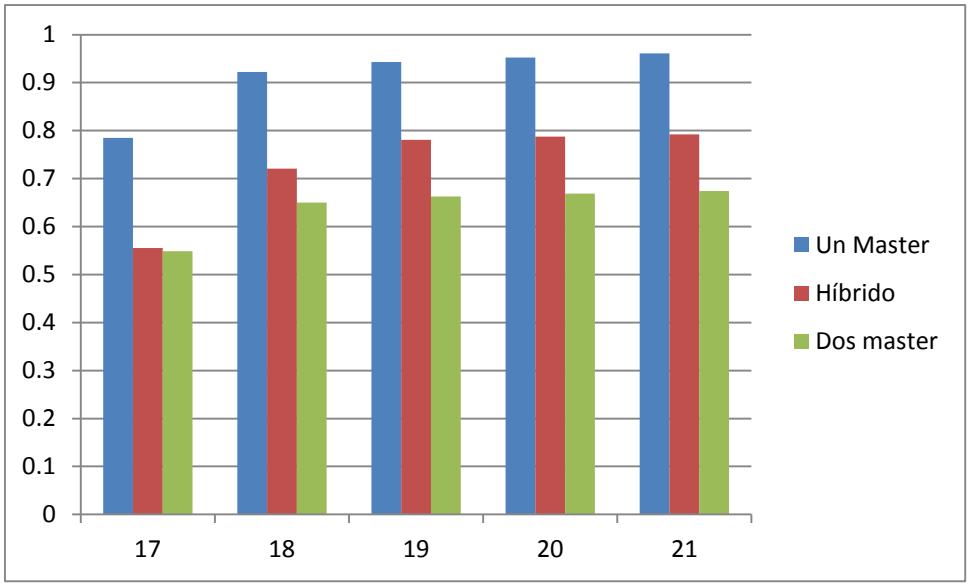


Figura 7.1

En el gráfico 7.1 se pueden apreciar dos aspectos. El primero es que a medida que el tamaño del tablero aumenta, también lo hace la eficiencia, si bien el primer salto notable se presenta al pasar de *N*=17 a *N*=18 (dado a que el tablero de 17x17 tarda muy poco tiempo) hay también un leve incremento en los *N* mayores. El segundo aspecto que se puede ver es que para todos los tamaños de tablero, la solución de Un Master (Modelo 1) aparece como la mejor, mostrando la mayor eficiencia en todos los casos, esto puede deberse al hecho de que aprovecha más cores de la arquitectura, y el cuello de botella que podría generar que todos los workers tengan que comunicarse con el mismo master no llega a generarse para la arquitectura utilizada. La siguiente

mejor solución es la Híbrida (Modelo 3), en siguiente lugar viene la solución que utiliza Dos Master con PM (Modelo 2).

7.2.2 Análisis del comportamiento cuando crece la arquirectura

Una vez determinada la eficiencia de cada solución en su comparación, se busca determinar el comportamiento de la misma para diferentes tamaños de arquitectura, analizando de esta manera la escalabilidad.

Para mostrar este punto, se tomó como referencia la ejecución de cada solución con los siguientes parámetros, tamaño del tablero NxN, tal que *N*=21, y porcentaje de reparto inicial 20% de las combinaciones. En la tabla 7.3 se puede ver la eficiencia en dicho caso:

	Un Master	Híbrido	Dos Master
4	0,927884	0,654695	0,562664
8	0,958532	0,749881	0,658056
12	0,961317	0,749881	0,674125
16	0,9714	0,81585	0,694061

Tabla 7.3

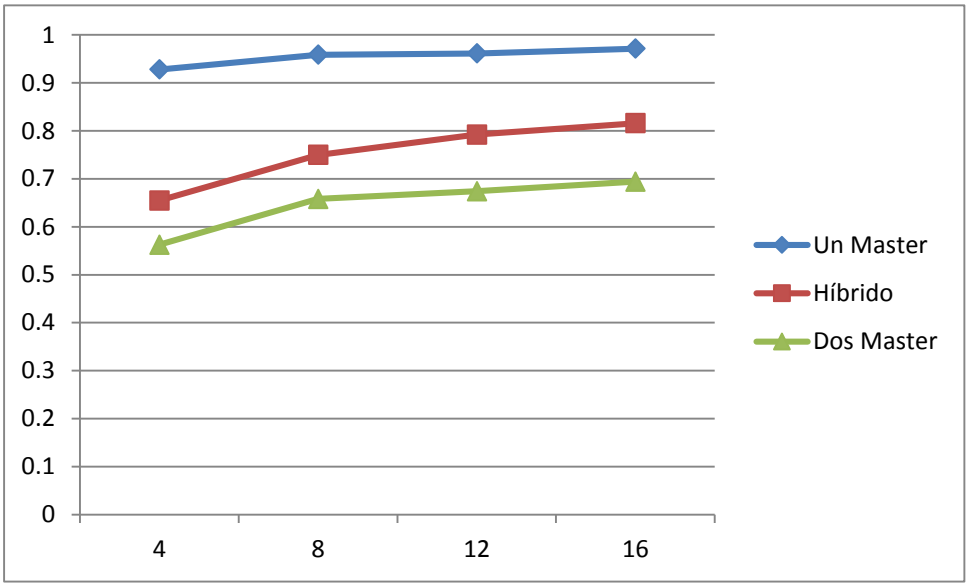


Figura 7.2

En el gráfico 7.2 se puede apreciar que en todas las soluciones a medida que el tamaño de la arquitectura utilizada aumenta, la eficiencia también lo hace. La solución de “Un Master” presenta para todas los tamaños una muy buena eficiencia y es por eso que en el gráfico no se ven grandes saltos de un tamaño a otro; es de suponer que si la arquitectura sigue creciendo, se encontrará un máximo, y hasta empezará a decaer debido al cuello de botella que se genera en el momento de la demanda de trabajo por parte de los procesos. Por otro lado tanto para la solución Híbrida como con PM con Dos niveles de Master se puede apreciar mejor el aumento de la eficiencia a medida que la arquitectura crece, probablemente este incremento se mantenga para tamaños de arquitectura más grandes donde la primera solución “Un Master” comenzaría a decaer.

7.3 Búsqueda de similitud máxima entre secuencias de ADN

Para realizar la experimentación se generaron bases de datos con distinta cantidad de secuencias. La diferencia entre dichas bases radica en el tamaño máximo (*N*) de los elementos que la componen (2500, 5000, 7500, 10000 y 20000), generando además una secuencia que se usa para realizar la comparación de similitud frente a las bases de datos. Dentro de cada Base de Datos la longitud de las secuencias varía el porcentaje del tamaño máximo. Se realizaron diferentes pruebas variando:

- Tamaño máximo de las secuencias en la Base de Datos y secuencia Test (2500, 5000, 7500, 10000 y 20000).

- Cantidad (*K*) de secuencias de la Base de Datos que se tendrán en cuenta (2500, 5000, 7500, 10000 y 15000).

Los tamaños de las bases y secuencias utilizados, son los mencionados anteriormente, pero se introduce además una variación que reside en el porcentaje de variación respecto del tamaño máximo que cada secuencia de una base puede tener. Los porcentajes que se usan en la experimentación son 10%, 25% y 50%. Es decir que si el tamaño máximo de secuencia para una base es 1000 y el porcentaje usado es 50%, podría haber secuencias con tamaño desde 500 hasta 1000. Las bases, así como la secuencia que se usa para la comparación, son generadas aleatoriamente por otro algoritmo, el cual para la generación de una base debe recibir como parámetro la longitud máxima, el porcentaje de variación y la cantidad de secuencias.

No vale la pena presentar aquí los resultados obtenidos para cada entrada, dado que si las bases utilizadas fuesen otras, los resultados serían distintos ya que la generación de las mismas se realiza de forma aleatoria.

Los resultados expuestos fueron elegidos dado que se los considera los más representativos. El conjunto total de los resultados puede encontrarse en el anexo correspondiente.

7.3.1 Comparación de las tres soluciones

Para cada una de las soluciones se probaron las combinaciones de tamaños máximos de secuencias para cada longitud de base (cantidad de secuencias) y con cada porcentaje de variabilidad con el que las bases fueron generadas.

A fin de mostrar los resultados, se eligió la ejecución para un tamaño máximo de secuencia de 20000, con un porcentaje de variabilidad del 25%. En la tabla 7.4 se puede ver la eficiencia de cada solución para las distintas cantidades de secuencias por base.

Cant. Secuencias	Un Master	Híbrido	Dos master
2500	0,9606778	0,8899377	0,6733401
5000	0,9721823	0,8991458	0,6827940
7500	0,9679597	0,8955431	0,6803071
10000	0,9562375	0,8842997	0,6718066
15000	0,9651734	0,8926673	0,6777436

Tabla 7.4

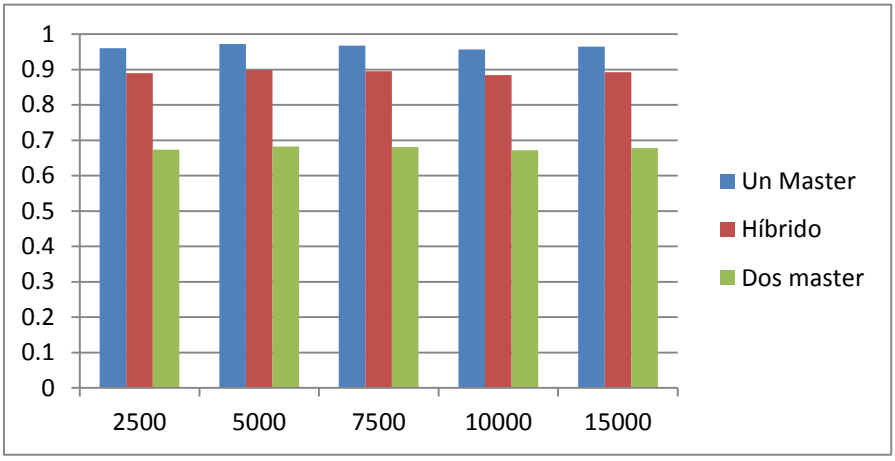


Figura 7.3

Al igual que para el problema N-reinas, resulta con mejor eficiencia la solución para un nivel de master, presentando una eficiencia similar, al igual que el de dos niveles de master. Pareciera entonces que el hecho de que el mensaje a transferir sea de mayor tamaño (tamaños de las secuencias de ADN) no afecta el rendimiento de las soluciones.

7.3.2 Análisis del comportamiento cuando crece la arquirectura

Una vez determinada la eficiencia de cada solución en su comparación, se busca determinar el comportamiento de la misma para diferentes tamaños de arquitectura, analizando de esta manera la escalabilidad.

Para mostrar este punto, se tomó como referencia la ejecución de cada solución con los siguientes parámetros longitud de las secuencias 20000, cantidad de secuencias 5000 y porcentaje de variación para el tamaño de la secuencia del 20%. En la tabla 7.5 se puede ver la eficiencia.

Cant. de máquinas	Un Master	Híbrido	Dos master
4	0,9280163	0,7587409	0,5573456
8	0,9582136	0,8793408	0,6489439
12	0,9721823	0,8991458	0,6827940
16	0,9821164	0,9252271	0,7016663

Tabla 7.5

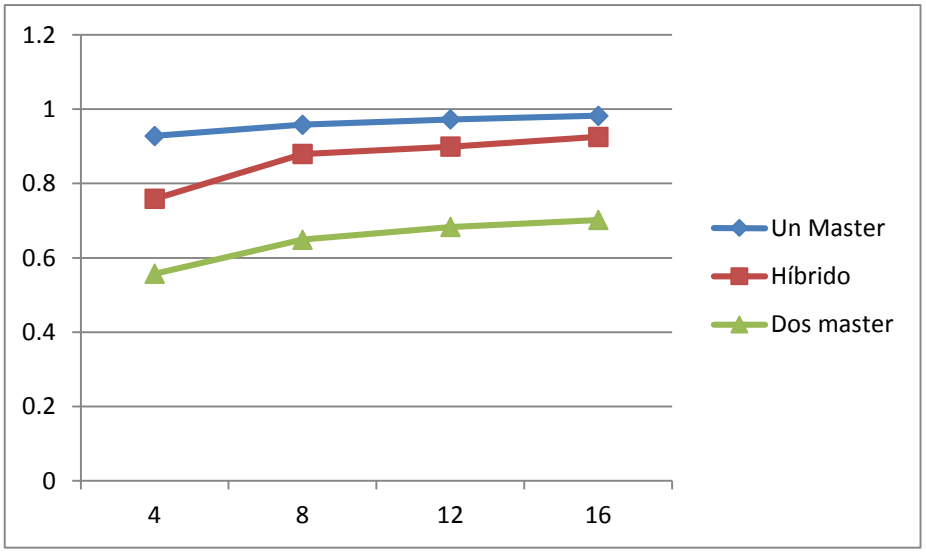


Figura 7.4

Se desprende de aquí entonces, que la eficiencia mejora a medida que se aumentan la cantidad de procesadores utilizados. El salto más grande se puede ver al pasar de utilizar 4 máquinas al doble, para los siguientes tamaños de arquitectura el porcentaje de mejora va disminuyendo, principalmente para el caso de la solución de un master, el cual además presenta muy buena eficiencia para todos los tamaños. Cuando la arquitectura utilizada es de 16 máquinas, la eficiencia de un master y la solución híbrida se empiezan a acercar más.

Capítulo 8

Conclusiones

El crecimiento del procesamiento paralelo y su utilidad en el aprovechamiento de sistemas de cómputo resulta innegable, principalmente debido a los avances de los últimos años que han puesto a disposición una diversidad muy amplia de arquitecturas multiprocesador, muy utilizadas por sus ventajas en términos de la relación costo/rendimiento, además de varias herramientas de programación que permiten explotar el paralelismo de las mismas.

Este crecimiento trae acarreado mayor incremento en la investigación y desarrollo de esta área de la Ciencia de la Computación, buscando cumplir con los objetivos del procesamiento paralelo: resolver problemas de mayor complejidad y volumen, incrementar la velocidad en la resolución de los mismos, y aprovechar de forma eficiente la arquitectura paralela.

El objetivo de esta Tesina es comparar el rendimiento del paradigma Master-Worker con distribución dinámica de trabajo utilizando uno y dos niveles de master sobre un cluster de multicores homogéneo, empleando diferentes modelos de comunicación, pasaje de mensajes y modelo híbrido.

Para realizar dicho análisis se utilizaron en particular dos aplicaciones. Por un lado la aplicación clásica de "N-reinas" donde predomina el procesamiento sobre el tamaño de los datos a comunicar. Por otro lado se utilizó el problema de "Búsqueda de máxima similitud en Bases de Datos de secuencias de ADN", donde en cambio las comunicaciones involucran gran cantidad de datos.

Se analizaron las soluciones para el problema de "N-reinas". Como resultado de este análisis se obtiene que la solución de "Un Master" resulta ser la mejor en todos los tamaños de tableros que fueron probados. Esta solución se encuentra seguida en eficiencia por la solución híbrida y apenas un poco más abajo por la solución de dos masters. Estas otras dos soluciones crecen en eficiencia como lo hace la de un master pero resultan ser claramente menos eficientes.

Se continuó con el análisis de las soluciones para el problema de "Búsqueda de máxima similitud en Bases de Datos de secuencias de ADN". Como resultado de este análisis se obtiene que la solución de "Un Master" resulta ser la mejor en todas las pruebas realizadas, seguida de cerca por la solución híbrida. Con una diferencia significativa sobre las otras dos soluciones se encuentra la solución de dos masters como la menos eficiente de todas. La eficiencia para todas las soluciones se mantiene constante y cercana a uno en el caso de un master y de la solución híbrida, a medida que el tamaño del problema crece.

Sobre estos resultados se puede concluir que la buena performance de la solución “Un master” se debe a que aprovecha al máximo la arquitectura utilizada y que dado al tamaño de esta, no se alcanza un cuello de botella que justificaría tener otro nivel de master para atacar esa contingencia. Por otro las soluciones que provee un nivel de master adicional tanto en Memoria compartida como con Pasaje de mensajes, se utilizan

menos núcleos de la arquitectura total, y esto los pone en desventaja. Sin embargo siguen mostrando buena performance.

La escalabilidad de un sistema paralelo es otro aspecto de importancia dado que permite capturar las características de un algoritmo paralelo y de la arquitectura en la que se lo implementa. Permite testear el rendimiento de un programa paralelo sobre pocos procesadores y predecir su comportamiento en un número mayor. También, da la posibilidad de caracterizar la cantidad de paralelismo inherente en un algoritmo, y puede usarse para estudiar el comportamiento de un sistema paralelo con respecto a cambios en parámetros de hardware tales como la velocidad de los procesadores y canales de comunicación.

Al analizar el comportamiento de las soluciones a medida que crecía el tamaño de la arquitectura en el problema de "N-reinas", todas mostraron una buena adaptación, en mayor o menor escala, a medida que el tamaño de la arquitectura aumenta. En el caso de un master este salto no resulta significativo entre un aumento y otro. En el caso de dos masters, se encuentra un primer salto significativo con una tendencia a estabilizarse. Finalmente, en el caso de la solución híbrida fue la que mayor porcentaje presentó con una tendencia a estabilizarse.

Al analizar el comportamiento de las soluciones a medida que crecía el tamaño de la arquitectura en el problema de "Búsqueda de máxima similitud en Bases de Datos de secuencias de ADN", todas mostraron ser un buen comportamiento a medida que el tamaño de la arquitectura aumenta. La la mejora resultó menos significativa a medida que el tamaño de la arquitectura aumenta para la solución de "Un Master", esto se debe a que esa solución ya presenta una muy buena eficiencia y está en línea a encontrar el punto óptimo. En cambio las soluciones "Híbridas" y de "Dos Niveles de Masters" mostraron una incremento de performance mayor, que disminuye entre un aumento y otro de la arquitectura. Todas ellas tendiendo a estabilizarse.

Tomando como punto de partida el trabajo de esta tesina y los resultados obtenidos se abre la puerta a un conjunto de temas para su futura investigación:

- Resulta particularmente interesante estudiar el comportamiento de ambas aplicaciones en arquitecturas de mayor tamaño, donde se pueda combinar el cluster de multicores empleado en esta tesina con clusters heterogéneos tradicionales.

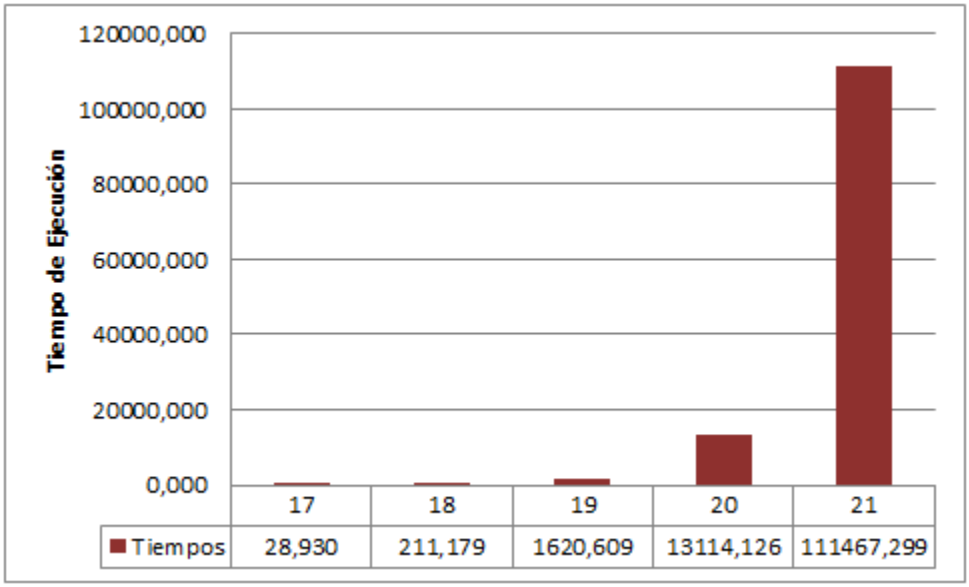
- Además, analizar la escalabilidad de los dos problemas asegurando un nivel determinado de eficiencia.

- Estudiar dónde se encuentra el cuello de botella de la solución con un master. Una vez encontrado ese punto, a partir de ahí estudiar qué sucede con la solución con dos niveles de masters, si supera en eficiencia a la solución con un master.

Apéndice A

Detalle de los resultados para "N-reinas"

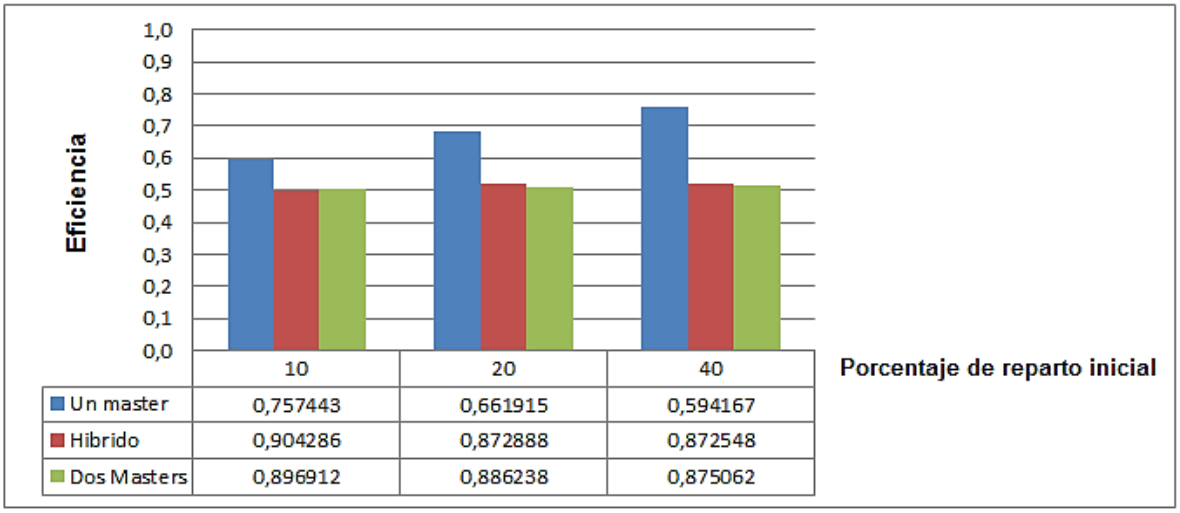
Resultados de los algoritmos secuenciales



	17	18	19	20	21
Tiempos	28,930	211,179	1620,609	13114,126	111467,299

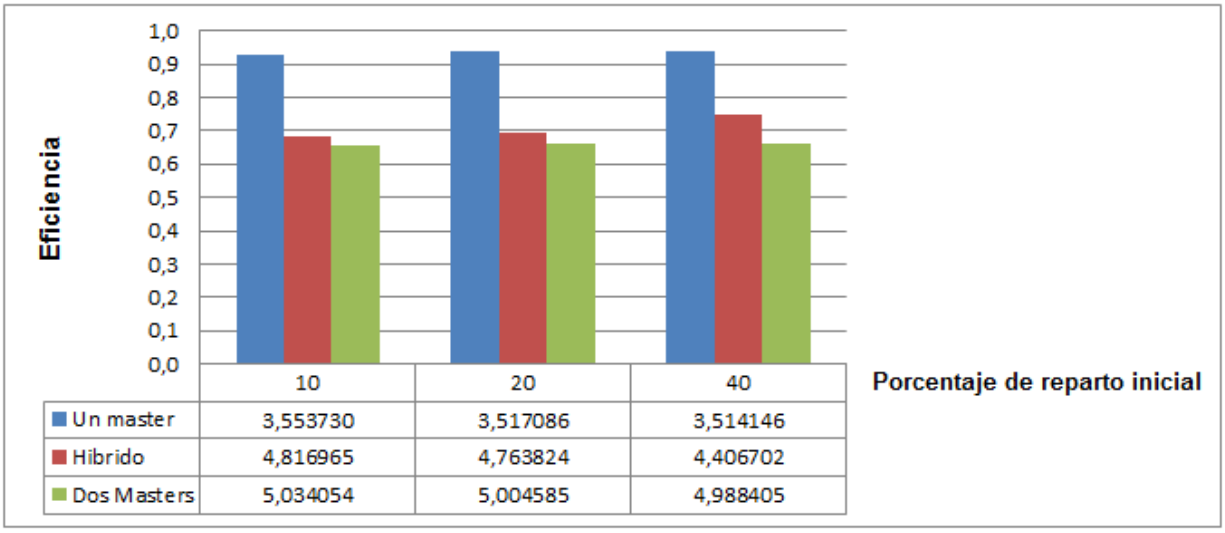
Figura A.1. En el gráfico y la tabla se pueden ver los tiempos de ejecución para los distintos N, siendo NxN el tamaño del tablero.

Resultados utilizando una arquitectura de 16 máquinas



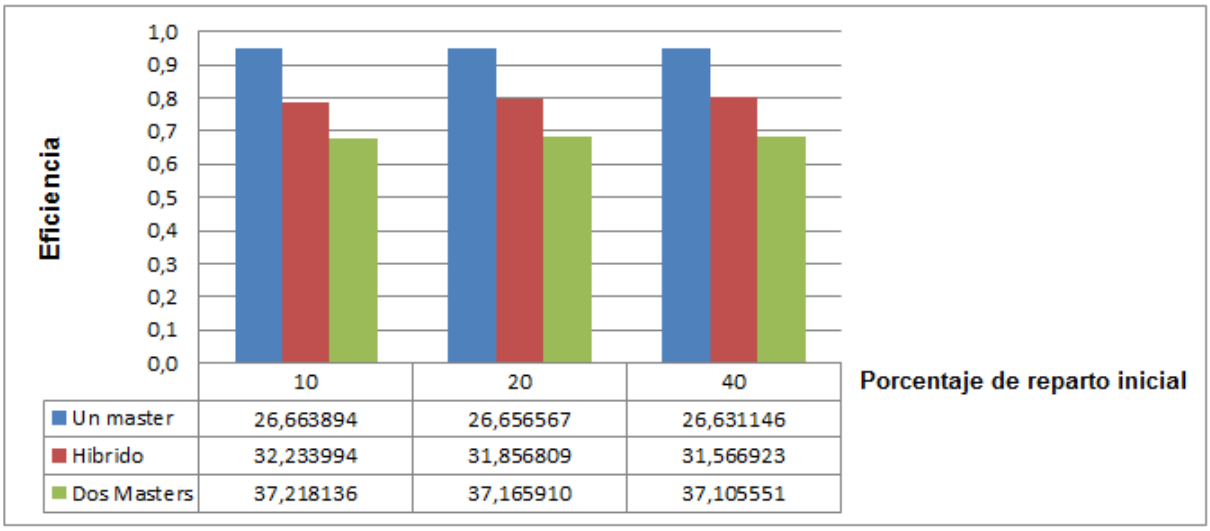
	10	20	40
Un master	0,757443	0,661915	0,594167
Hibrido	0,904286	0,872888	0,872548
Dos Masters	0,896912	0,886238	0,875062

Figura A.2. El gráfico representa la eficiencia para un tablero de tamaño 17x17, para los distintos porcentajes de reparto inicial. La tabla de abajo representa los tiempos de ejecución.



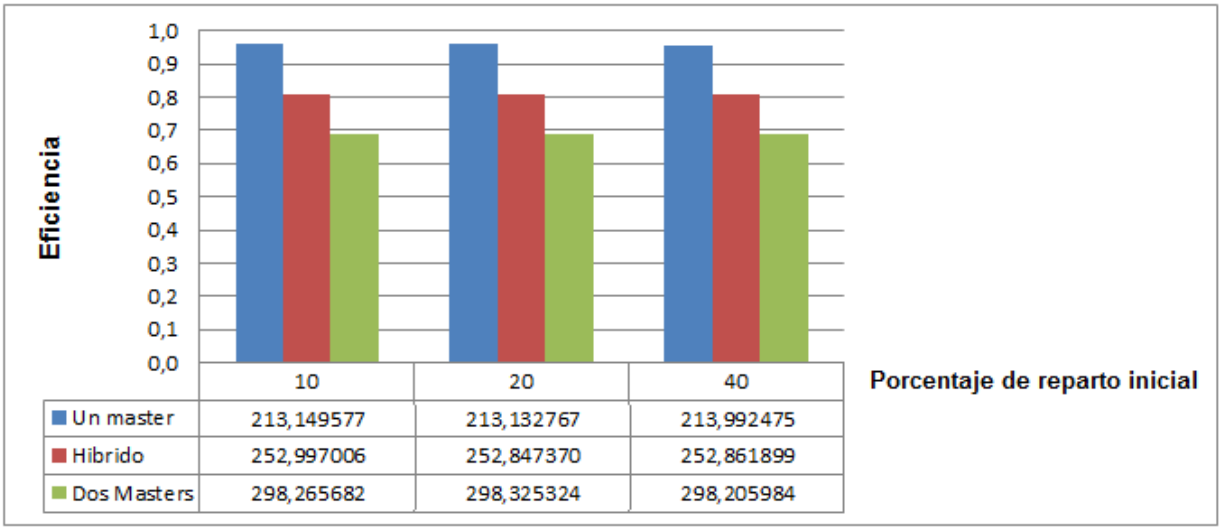
	10	20	40
Un master	3,553730	3,517086	3,514146
Hibrido	4,816965	4,763824	4,406702
Dos Masters	5,034054	5,004585	4,988405

Figura A.3. El gráfico representa la eficiencia para un tablero de tamaño 18x18, para los distintos porcentajes de reparto inicial. La tabla de abajo representa los tiempos de ejecución.



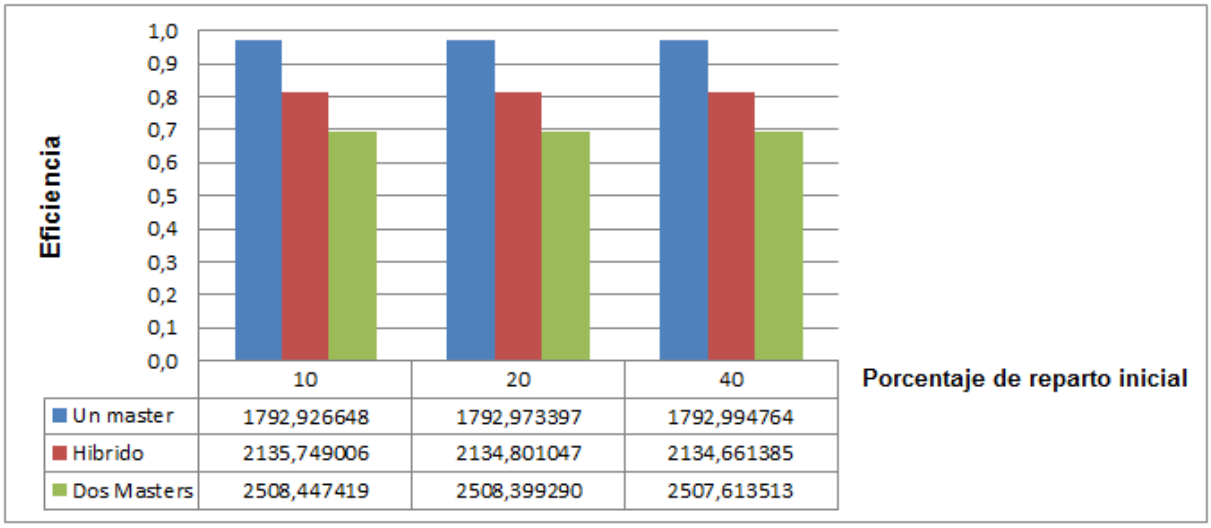
	10	20	40
Un master	26,663894	26,656567	26,631146
Hibrido	32,233994	31,856809	31,566923
Dos Masters	37,218136	37,165910	37,105551

Figura A.4. El gráfico representa la eficiencia para un tablero de tamaño 19x19, para los distintos porcentajes de reparto inicial. La tabla de abajo representa los tiempos de ejecución.



	10	20	40
Un master	213,149577	213,132767	213,992475
Hibrido	252,997006	252,847370	252,861899
Dos Masters	298,265682	298,325324	298,205984

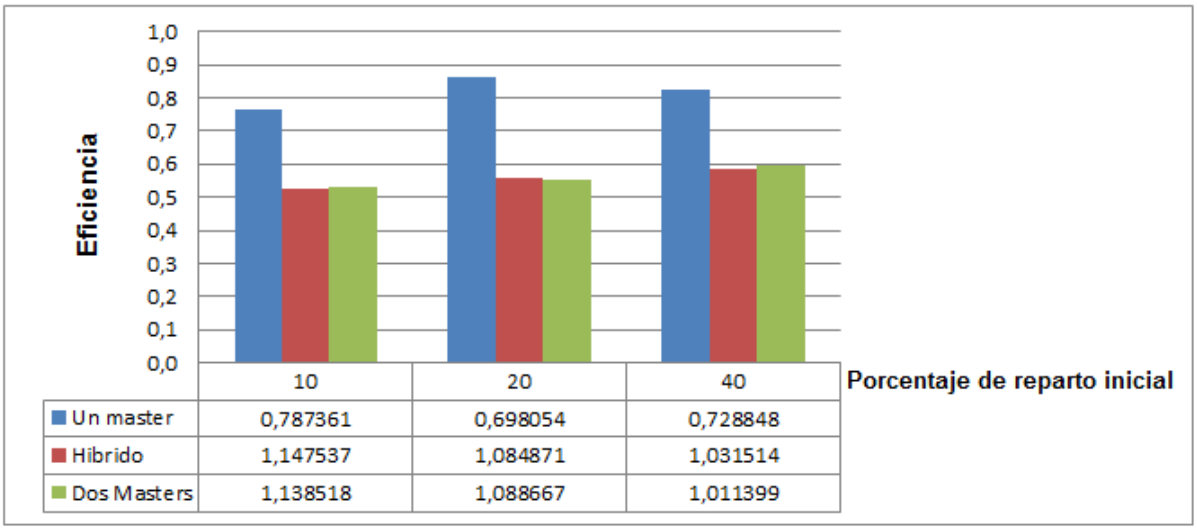
Figura A.5. El gráfico representa la eficiencia para un tablero de tamaño 20x20, para los distintos porcentajes de reparto inicial. La tabla de abajo representa los tiempos de ejecución.



	10	20	40
Un master	1792,926648	1792,973397	1792,994764
Hibrido	2135,749006	2134,801047	2134,661385
Dos Masters	2508,447419	2508,399290	2507,613513

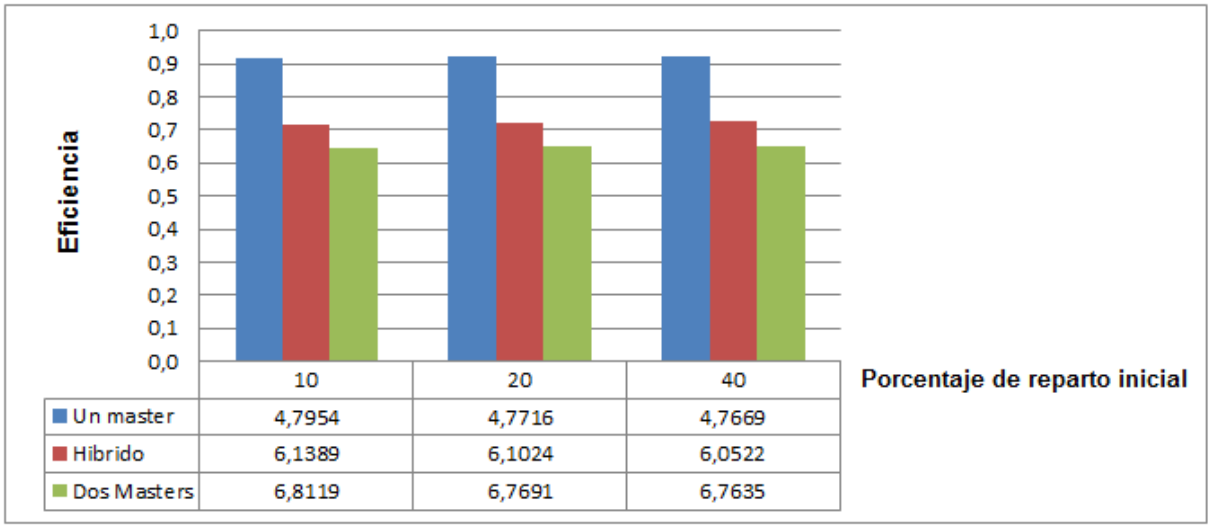
Figura A.6. El gráfico representa la eficiencia para un tablero de tamaño 21x21, para los distintos porcentajes de reparto inicial. La tabla de abajo representa los tiempos de ejecución.

Resultados utilizando una arquitectura de 12 máquinas



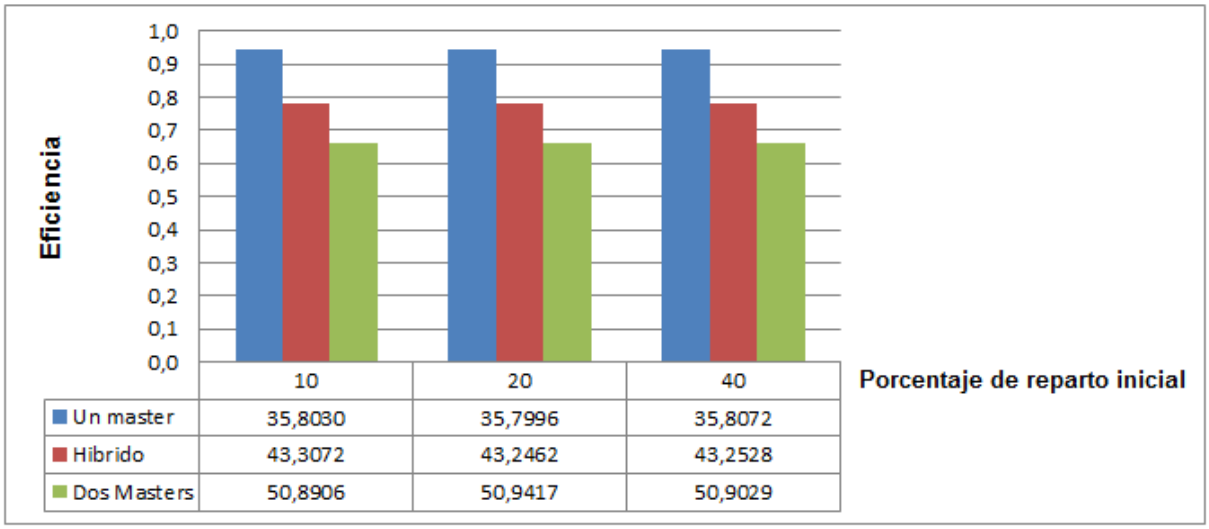
	10	20	40
Un master	0,787361	0,698054	0,728848
Hibrido	1,147537	1,084871	1,031514
Dos Masters	1,138518	1,088667	1,011399

Figura A.7. El gráfico representa la eficiencia para un tablero de tamaño 17x17 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



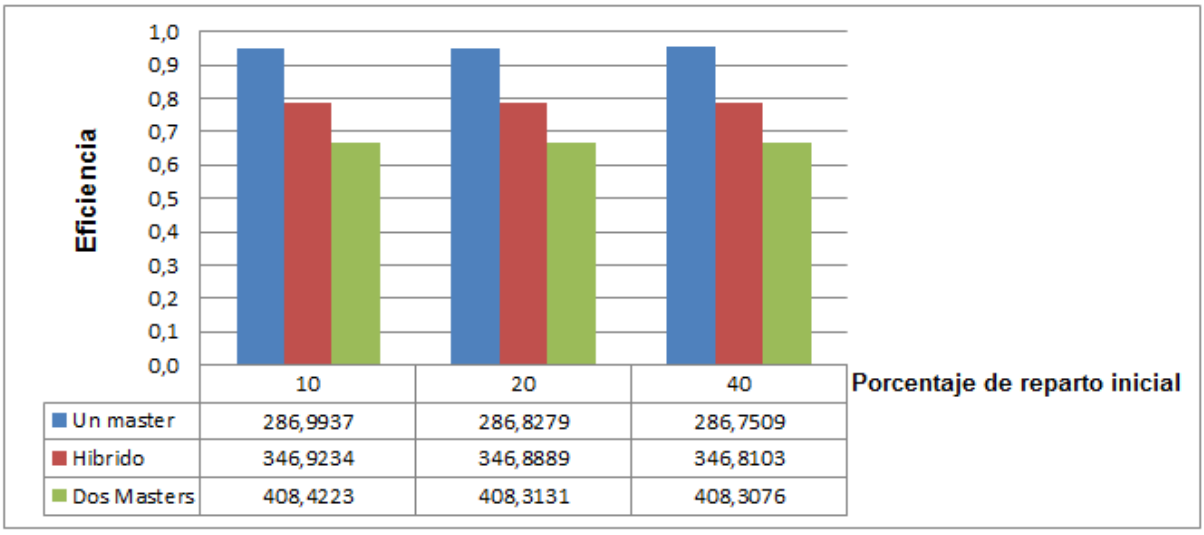
	10	20	40
Un master	4,7954	4,7716	4,7669
Hibrido	6,1389	6,1024	6,0522
Dos Masters	6,8119	6,7691	6,7635

Figura A.8. El gráfico representa la eficiencia para un tablero de tamaño 18x18 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



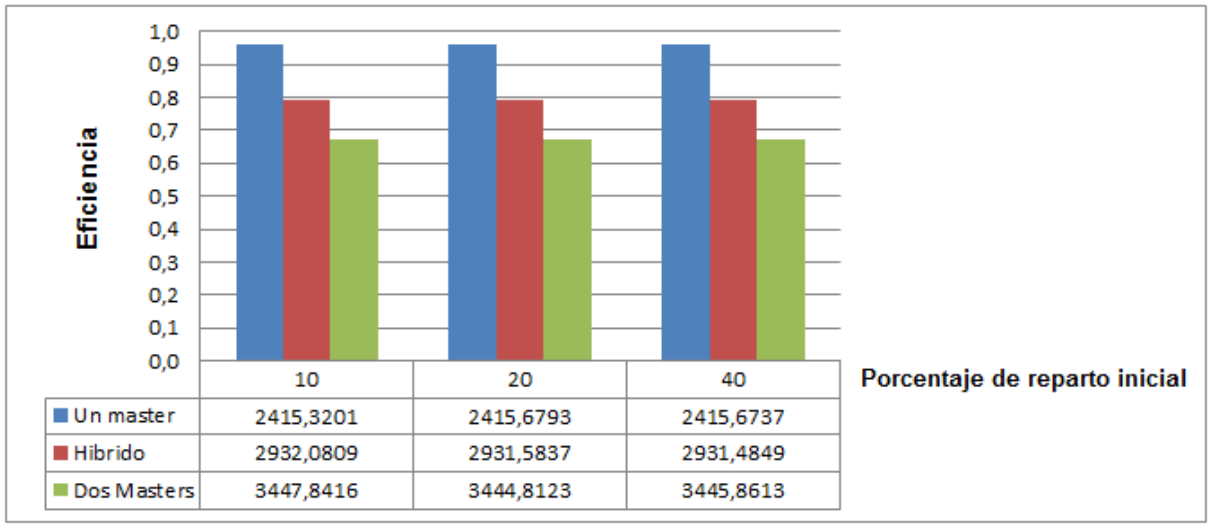
	10	20	40
Un master	35,8030	35,7996	35,8072
Hibrido	43,3072	43,2462	43,2528
Dos Masters	50,8906	50,9417	50,9029

Figura A.9. El gráfico representa la eficiencia para un tablero de tamaño 19x19 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



	10	20	40
Un master	286,9937	286,8279	286,7509
Hibrido	346,9234	346,8889	346,8103
Dos Masters	408,4223	408,3131	408,3076

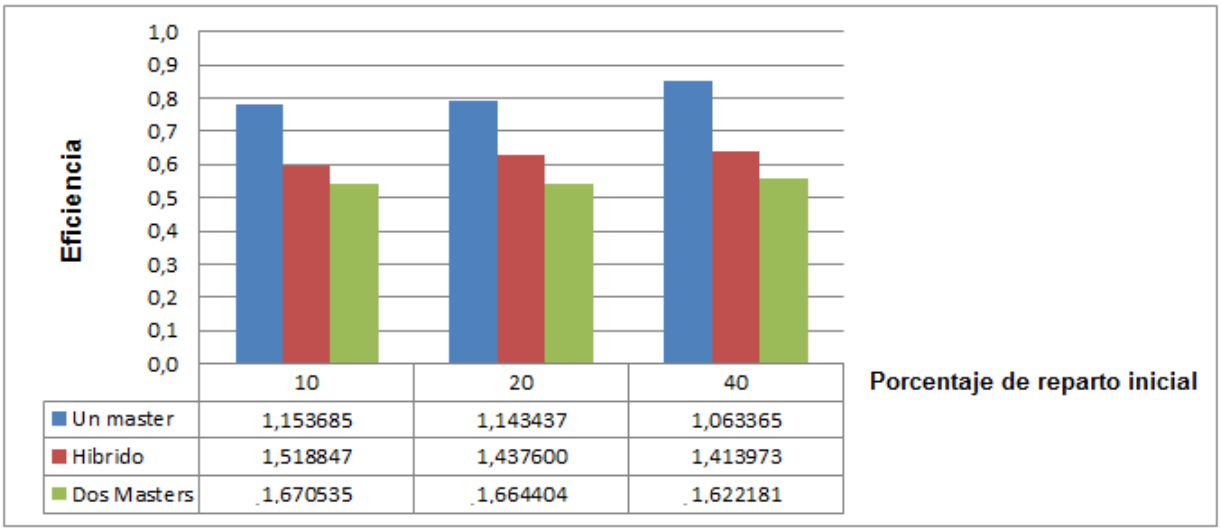
Figura A.10. El gráfico representa la eficiencia para un tablero de tamaño 20x20 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



	10	20	40
Un master	2415,3201	2415,6793	2415,6737
Hibrido	2932,0809	2931,5837	2931,4849
Dos Masters	3447,8416	3444,8123	3445,8613

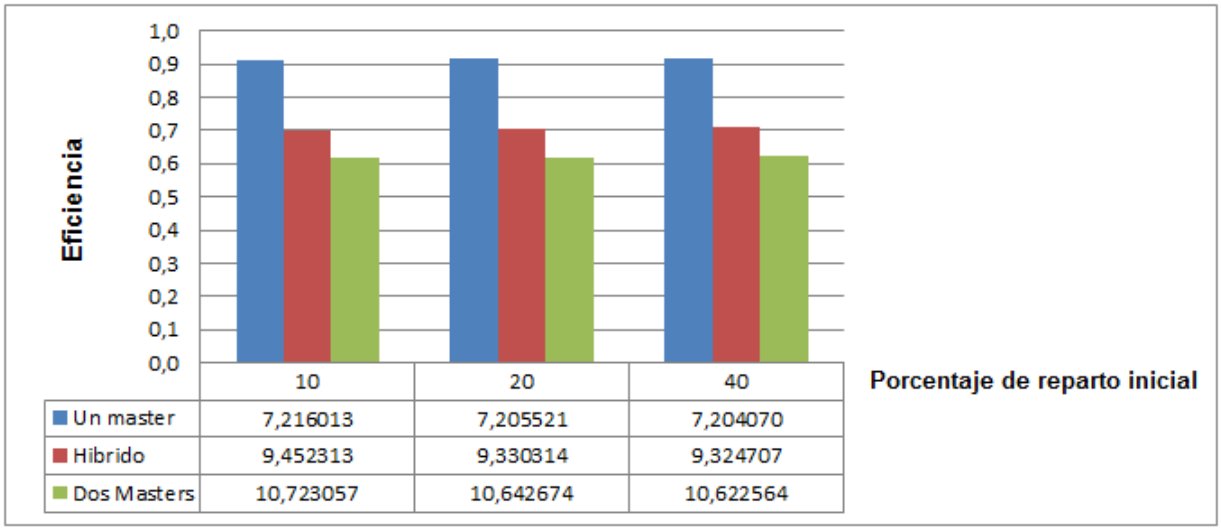
Figura A.11. El gráfico representa la eficiencia para un tablero de tamaño 21x21 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.

Resultados utilizando una arquitectura de 8 máquinas



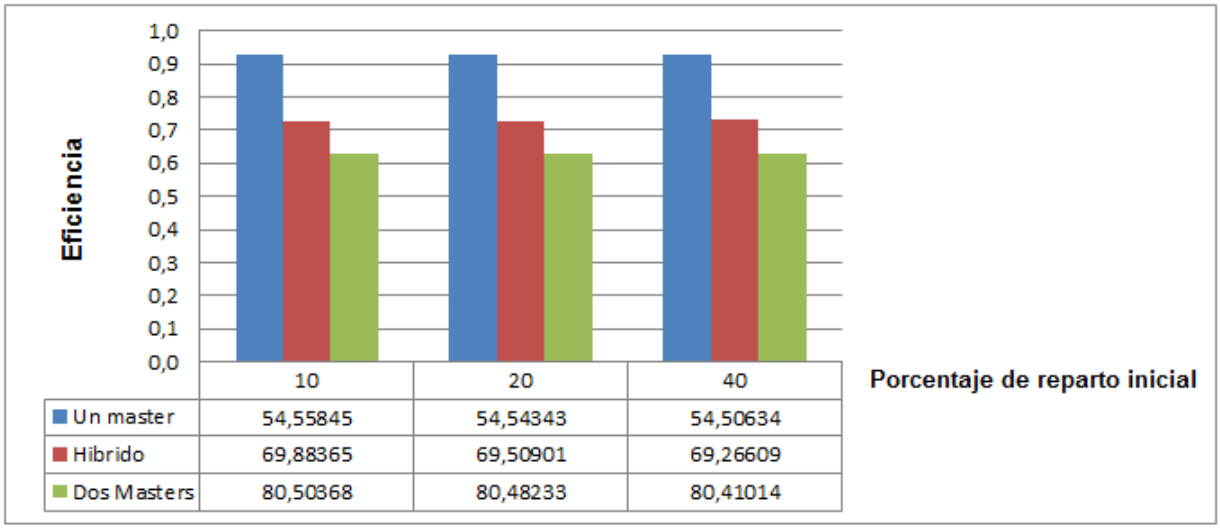
	10	20	40
Un master	1,153685	1,143437	1,063365
Hibrido	1,518847	1,437600	1,413973
Dos Masters	1,670535	1,664404	1,622181

Figura A.12. El gráfico representa la eficiencia para un tablero de tamaño 17x17 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



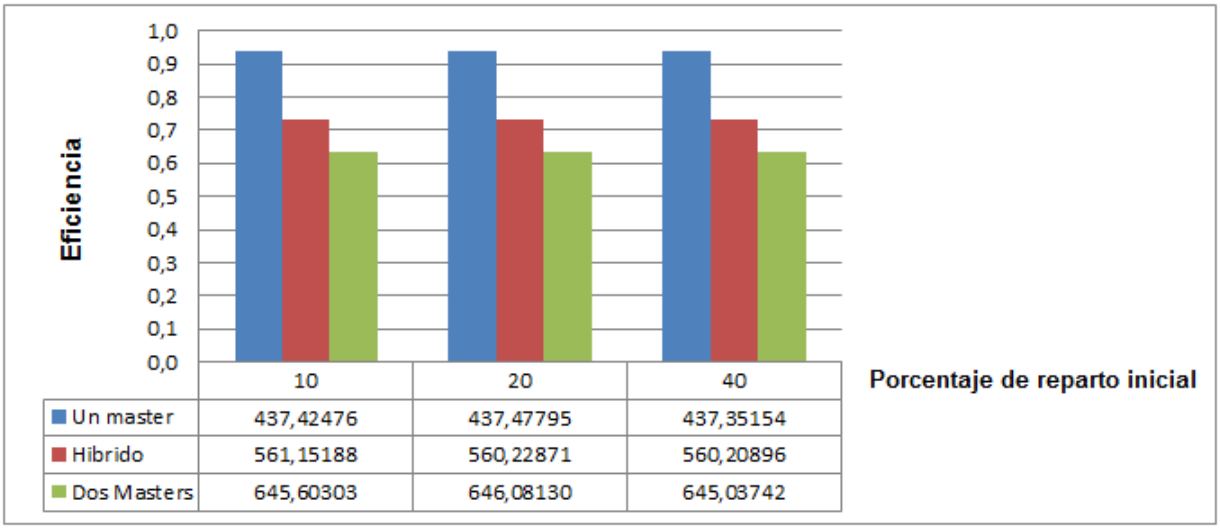
	10	20	40
Un master	7,216013	7,205521	7,204070
Hibrido	9,452313	9,330314	9,324707
Dos Masters	10,723057	10,642674	10,622564

Figura A.13. El gráfico representa la eficiencia para un tablero de tamaño 18x18 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



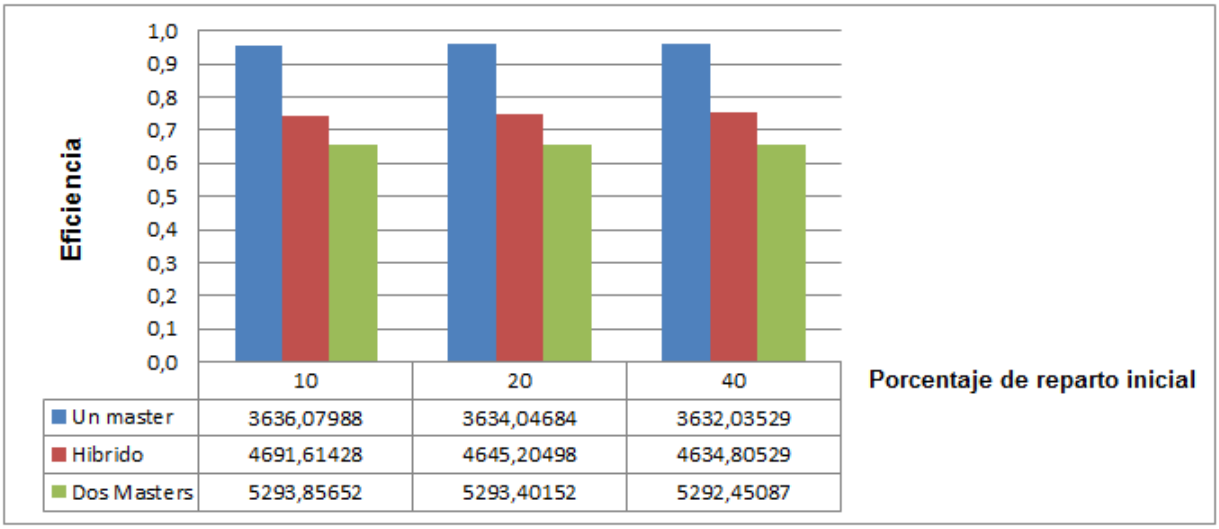
	10	20	40
Un master	54,55845	54,54343	54,50634
Hibrido	69,88365	69,50901	69,26609
Dos Masters	80,50368	80,48233	80,41014

Figura A.14. El gráfico representa la eficiencia para un tablero de tamaño 19x19 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



	10	20	40
Un master	437,42476	437,47795	437,35154
Hibrido	561,15188	560,22871	560,20896
Dos Masters	645,60303	646,08130	645,03742

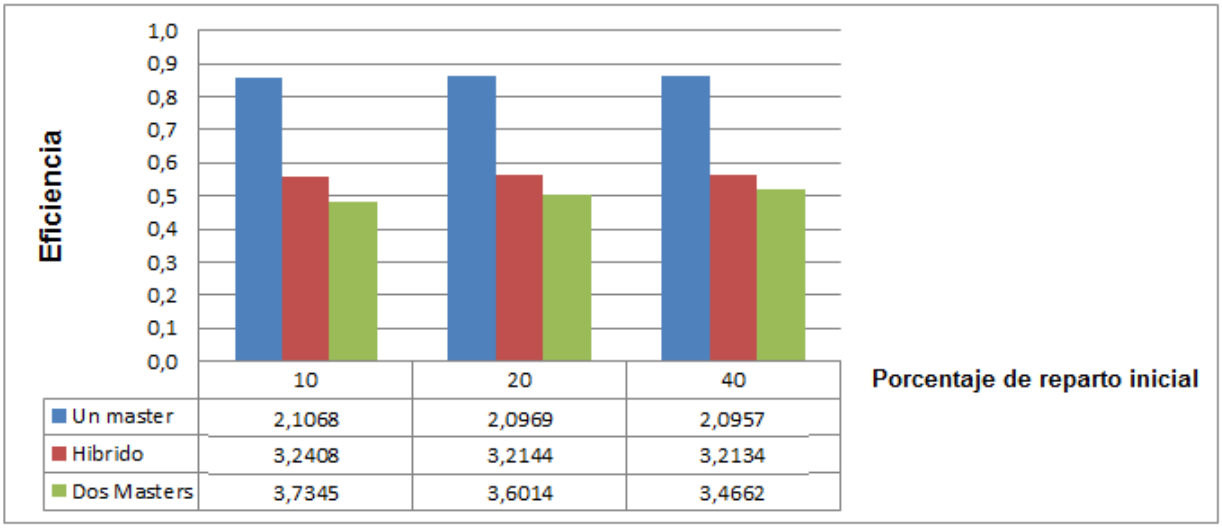
Figura A.15. El gráfico representa la eficiencia para un tablero de tamaño 20x20 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



	10	20	40
Un master	3636,07988	3634,04684	3632,03529
Hibrido	4691,61428	4645,20498	4634,80529
Dos Masters	5293,85652	5293,40152	5292,45087

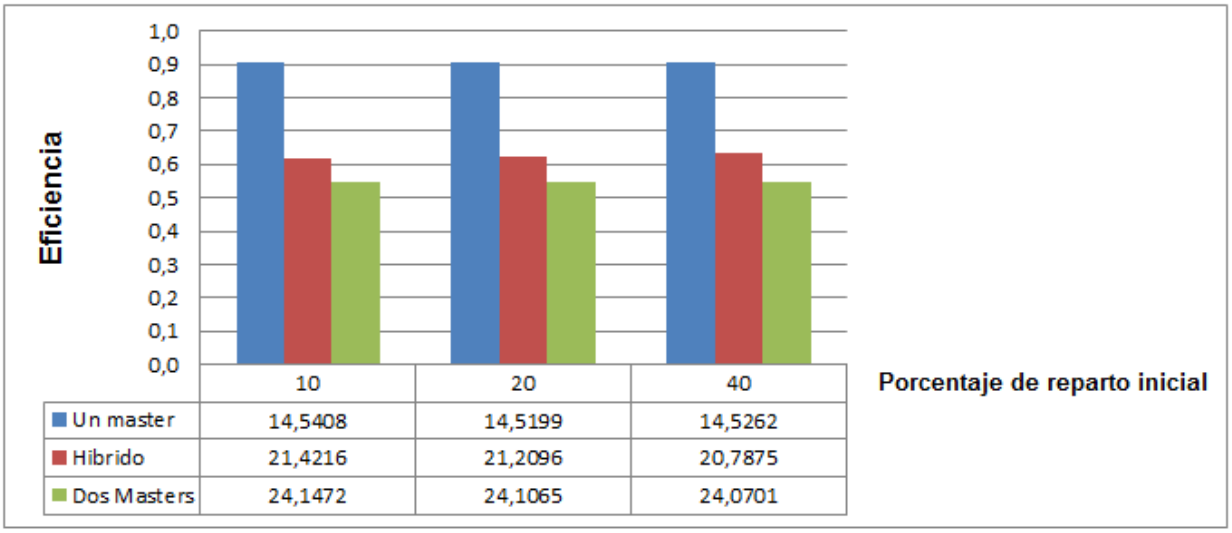
Figura A.16. El gráfico representa la eficiencia para un tablero de tamaño 21x21 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.

Resultados utilizando una arquitectura de 4 máquinas



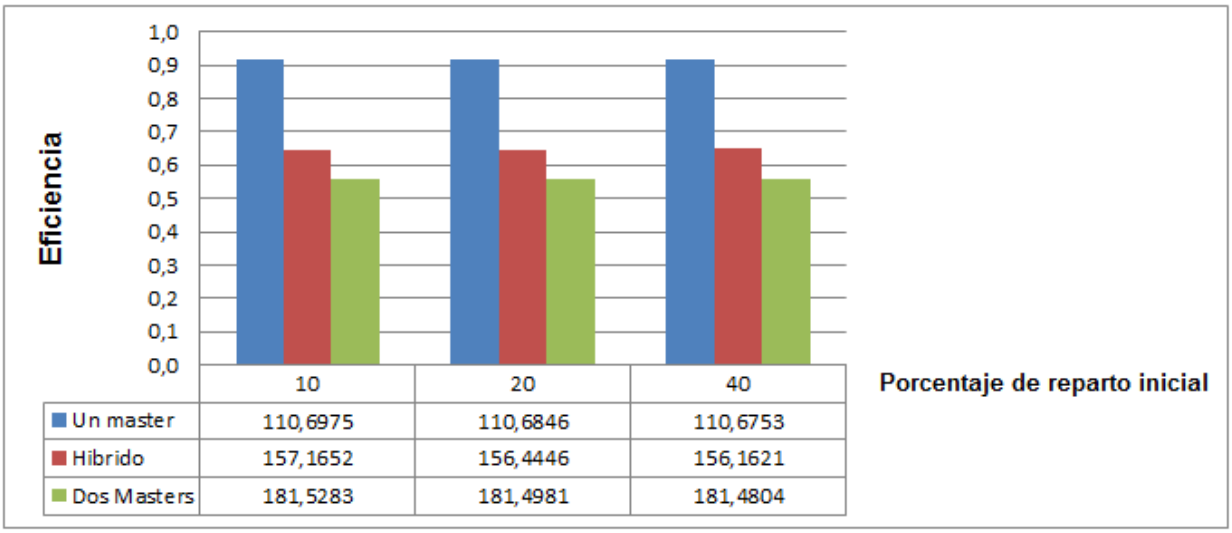
	10	20	40
Un master	2,1068	2,0969	2,0957
Hibrido	3,2408	3,2144	3,2134
Dos Masters	3,7345	3,6014	3,4662

Figura A.17. El gráfico representa la eficiencia para un tablero de tamaño 17x17 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



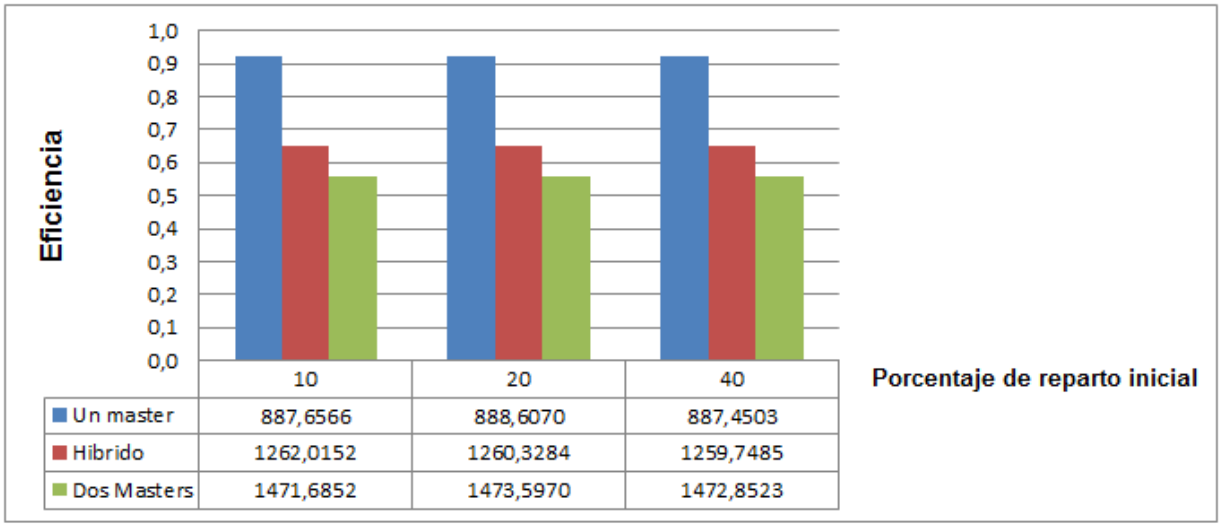
	10	20	40
Un master	14,5408	14,5199	14,5262
Hibrido	21,4216	21,2096	20,7875
Dos Masters	24,1472	24,1065	24,0701

Figura A.18. El gráfico representa la eficiencia para un tablero de tamaño 18x18 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



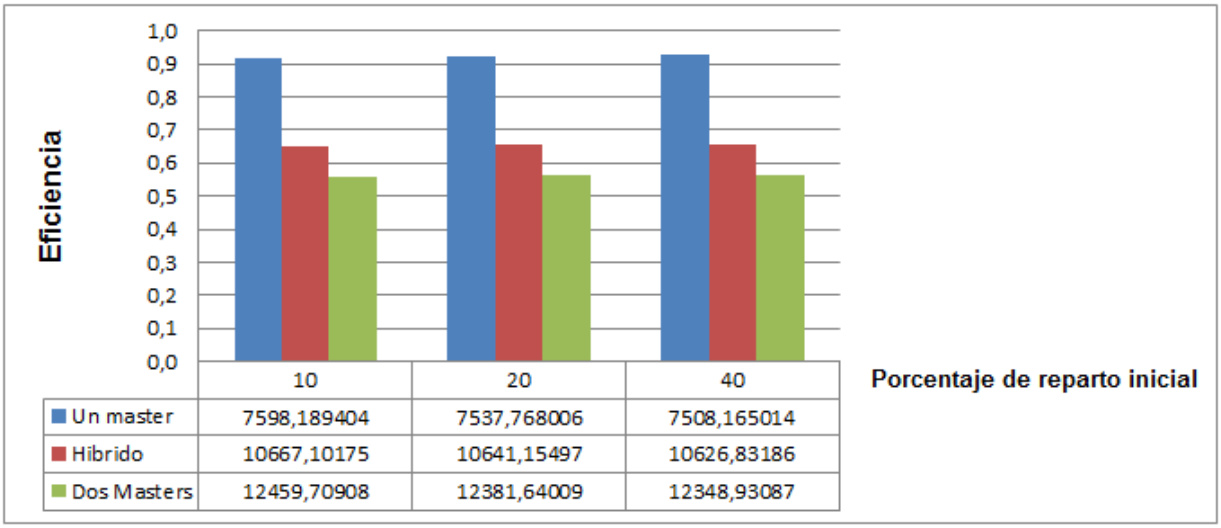
	10	20	40
Un master	110,6975	110,6846	110,6753
Hibrido	157,1652	156,4446	156,1621
Dos Masters	181,5283	181,4981	181,4804

Figura A.19. El gráfico representa la eficiencia para un tablero de tamaño 19x19 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



	10	20	40
Un master	887,6566	888,6070	887,4503
Hibrido	1262,0152	1260,3284	1259,7485
Dos Masters	1471,6852	1473,5970	1472,8523

Figura A.20. El gráfico representa la eficiencia para un tablero de tamaño 20x20 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.



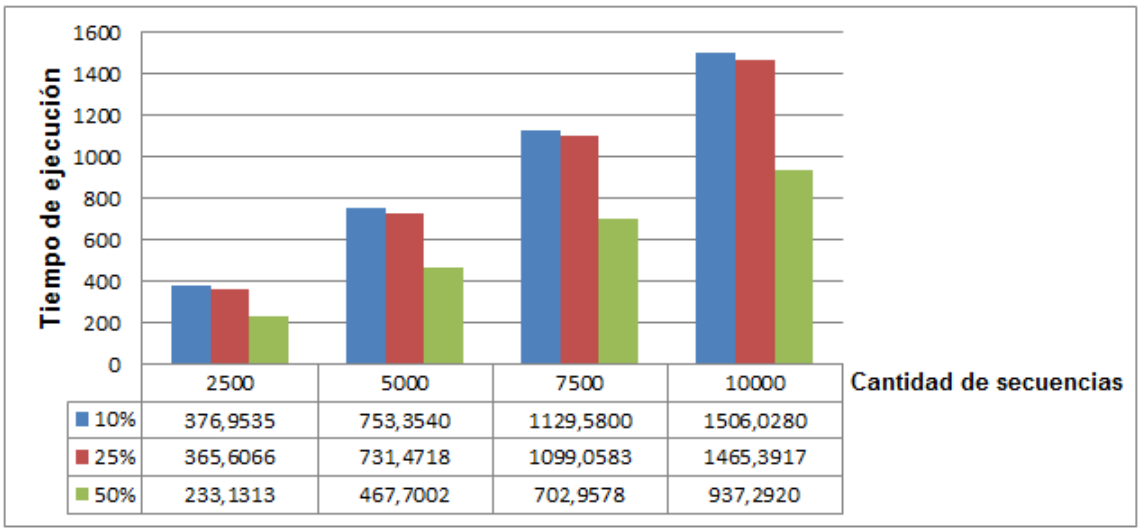
	10	20	40
Un master	7598,189404	7537,768006	7508,165014
Hibrido	10667,10175	10641,15497	10626,83186
Dos Masters	12459,70908	12381,64009	12348,93087

Figura A.21. El gráfico representa la eficiencia para un tablero de tamaño 21x21 para los distintos porcentajes de reparto inicial para las distintas soluciones. En la tabla se pueden ver los tiempos de ejecución.

Apéndice B

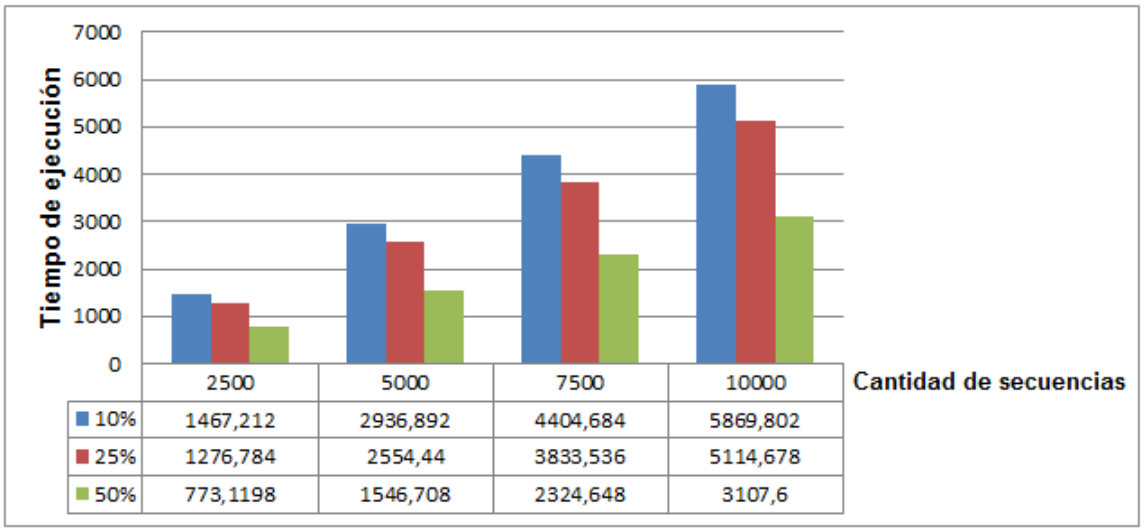
Detalle de los resultados para "Búsqueda de similitud máxima en secuencias de ADN"

Resultados de algoritmos secuenciales



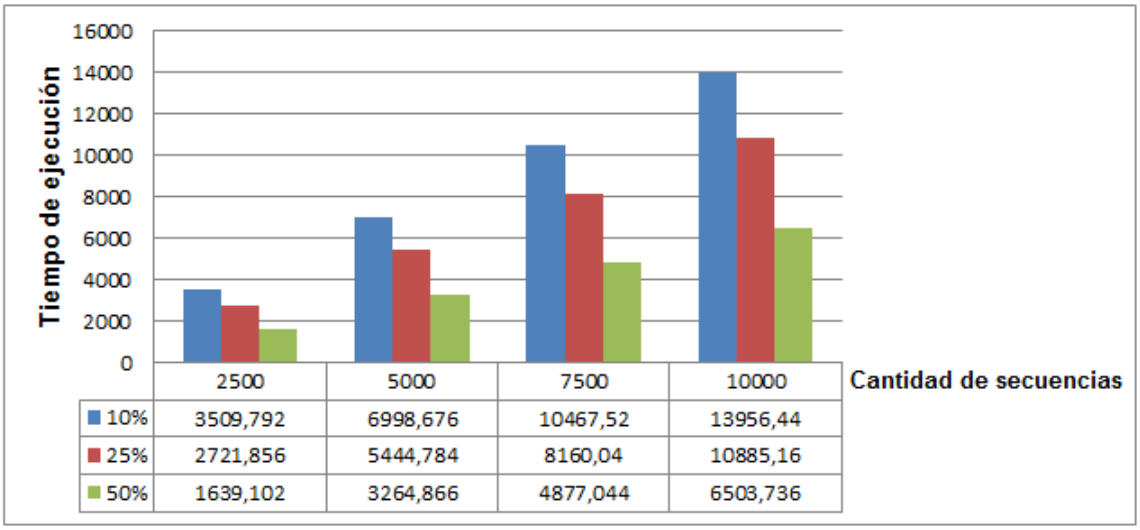
	2500	5000	7500	10000
10%	376,9535	753,3540	1129,5800	1506,0280
25%	365,6066	731,4718	1099,0583	1465,3917
50%	233,1313	467,7002	702,9578	937,2920

Figura B.1. El gráfico representa los tiempos de ejecución para usando secuencias de longitud 2500, para los distintos porcentajes de variabilidad en el tamaño y distintas cantidades de secuencias.



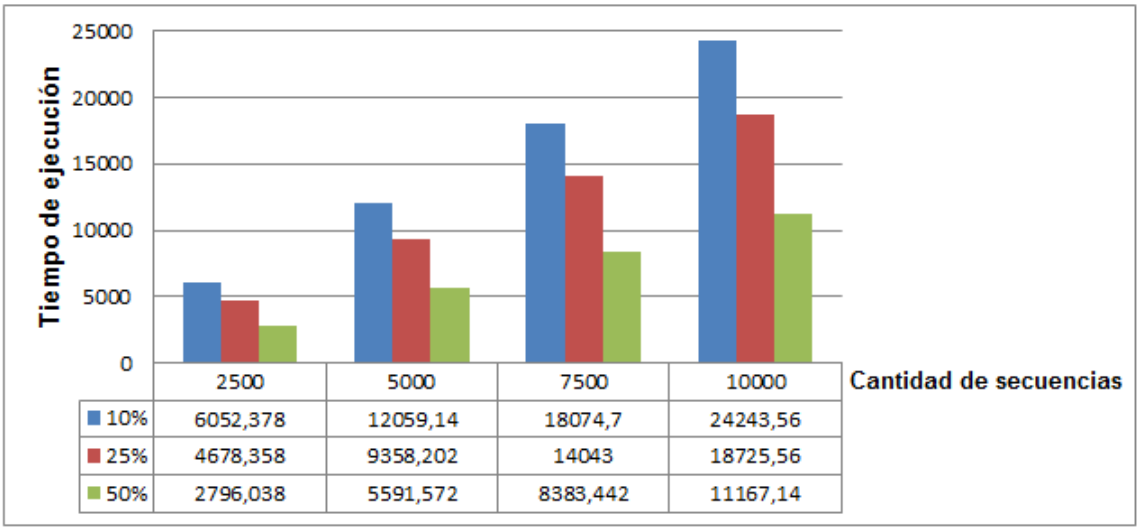
	2500	5000	7500	10000
10%	1467,212	2936,892	4404,684	5869,802
25%	1276,784	2554,44	3833,536	5114,678
50%	773,1198	1546,708	2324,648	3107,6

Figura B.2. El gráfico representa los tiempos de ejecución para usando secuencias de longitud 5000, para los distintos porcentajes de variabilidad en el tamaño y distintas cantidades de secuencias.



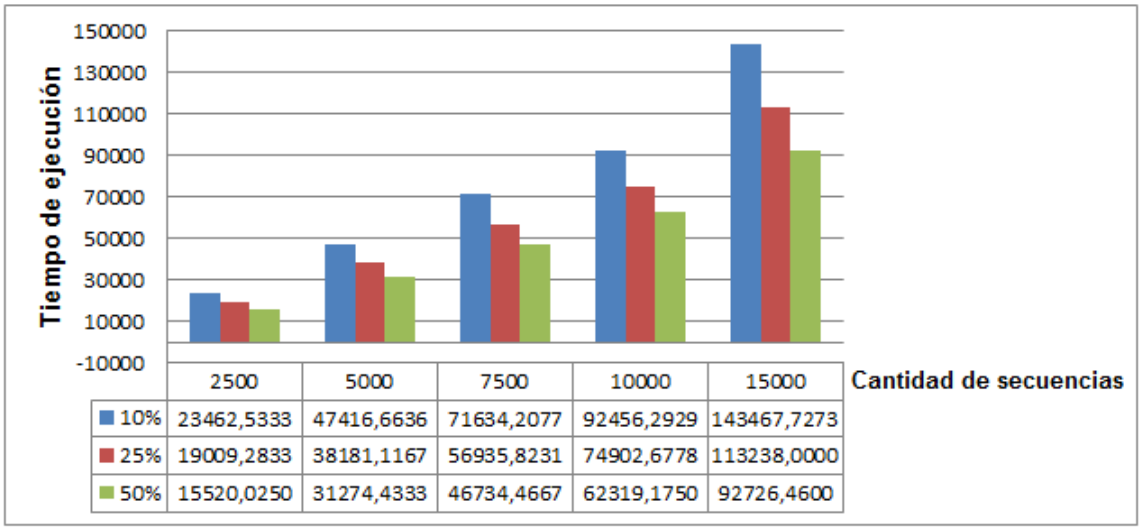
	2500	5000	7500	10000
10%	3509,792	6998,676	10467,52	13956,44
25%	2721,856	5444,784	8160,04	10885,16
50%	1639,102	3264,866	4877,044	6503,736

Figura B.3. El gráfico representa los tiempos de ejecución para usando secuencias de longitud 7500, para los distintos porcentajes de variabilidad en el tamaño y distintas cantidades de secuencias.



	2500	5000	7500	10000
10%	6052,378	12059,14	18074,7	24243,56
25%	4678,358	9358,202	14043	18725,56
50%	2796,038	5591,572	8383,442	11167,14

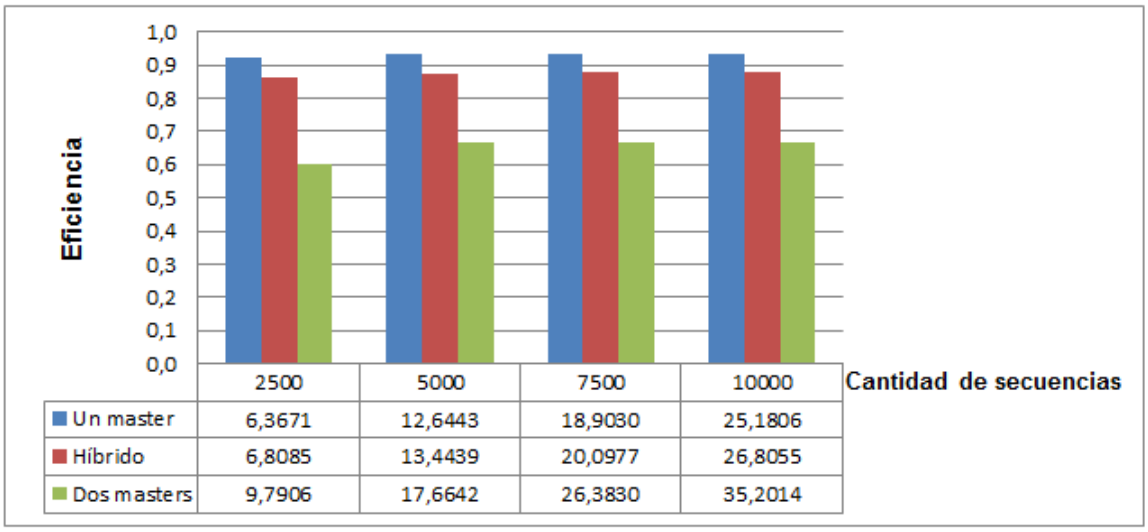
Figura B.4. El gráfico representa los tiempos de ejecución para usando secuencias de longitud 10000, para los distintos porcentajes de variabilidad en el tamaño y distintas cantidades de secuencias.



	2500	5000	7500	10000	15000
10%	23462,5333	47416,6636	71634,2077	92456,2929	143467,7273
25%	19009,2833	38181,1167	56935,8231	74902,6778	113238,0000
50%	15520,0250	31274,4333	46734,4667	62319,1750	92726,4600

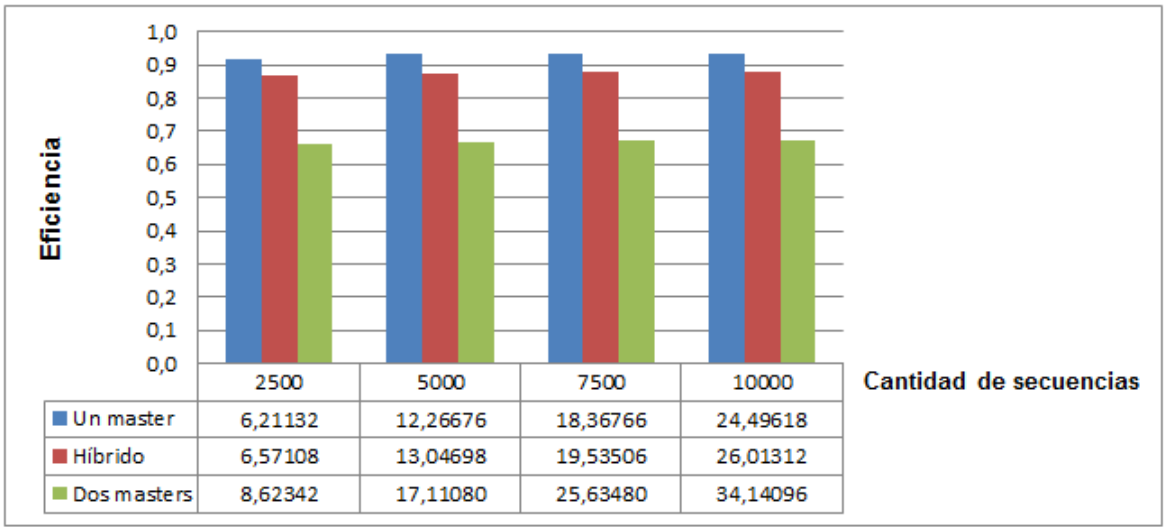
Figura B.5. El gráfico representa los tiempos de ejecución para usando secuencias de longitud 20000, para los distintos porcentajes de variabilidad en el tamaño y distintas cantidades de secuencias.

Resultados utilizando una arquitectura de 16 máquinas



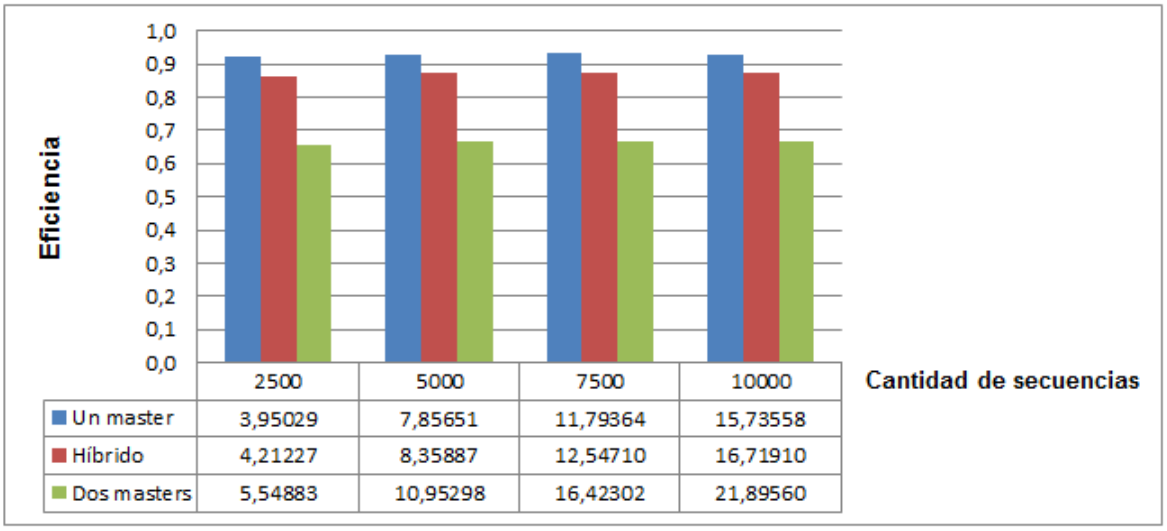
	2500	5000	7500	10000
Un master	6,3671	12,6443	18,9030	25,1806
Híbrido	6,8085	13,4439	20,0977	26,8055
Dos masters	9,7906	17,6642	26,3830	35,2014

Figura B.6. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



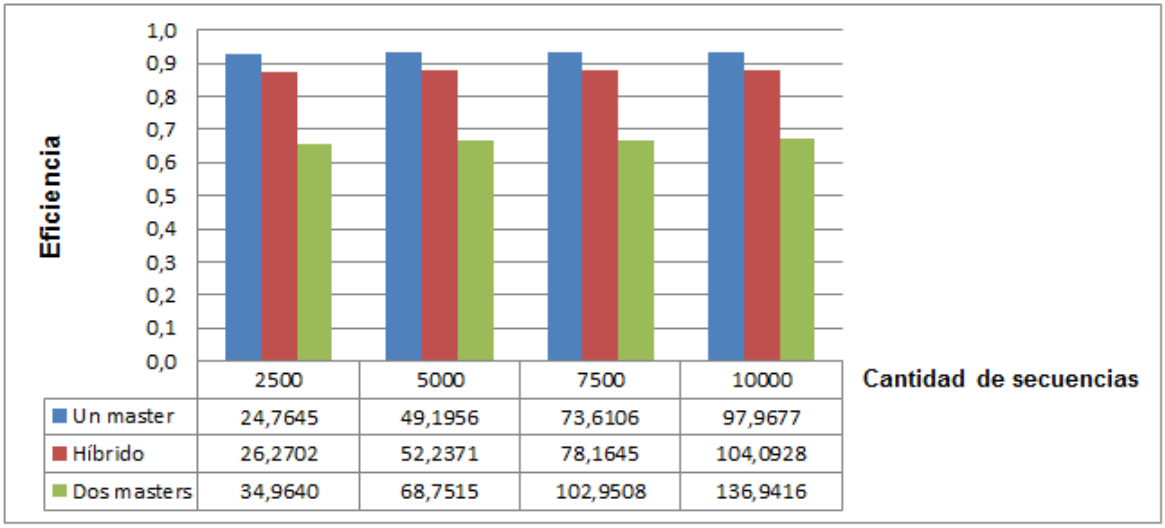
	2500	5000	7500	10000
Un master	6,21132	12,26676	18,36766	24,49618
Híbrido	6,57108	13,04698	19,53506	26,01312
Dos masters	8,62342	17,11080	25,63480	34,14096

Figura B.7. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



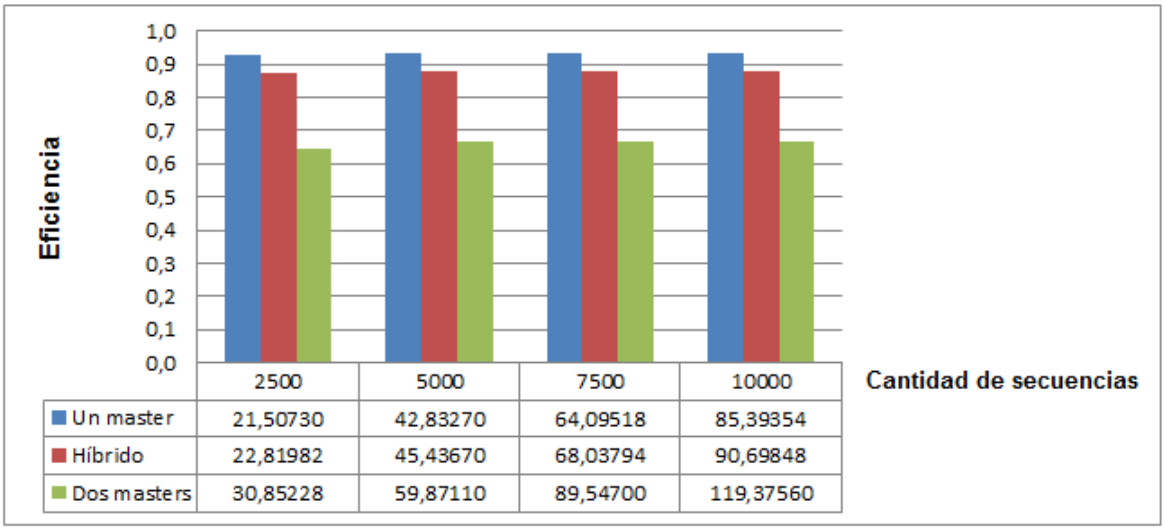
	2500	5000	7500	10000
Un master	3,95029	7,85651	11,79364	15,73558
Híbrido	4,21227	8,35887	12,54710	16,71910
Dos masters	5,54883	10,95298	16,42302	21,89560

Figura B.8. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



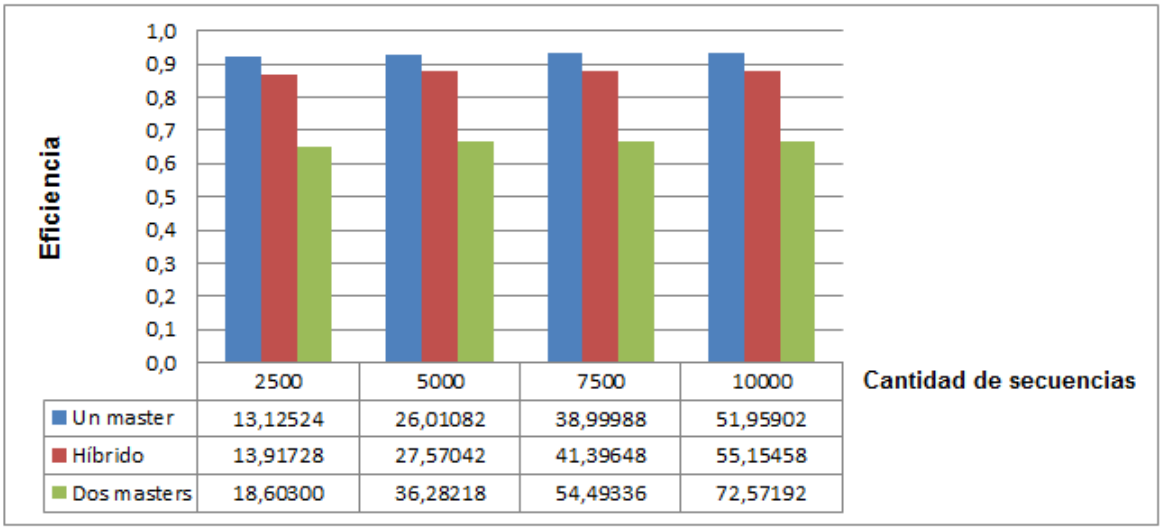
	2500	5000	7500	10000
Un master	24,7645	49,1956	73,6106	97,9677
Híbrido	26,2702	52,2371	78,1645	104,0928
Dos masters	34,9640	68,7515	102,9508	136,9416

Figura B.9. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



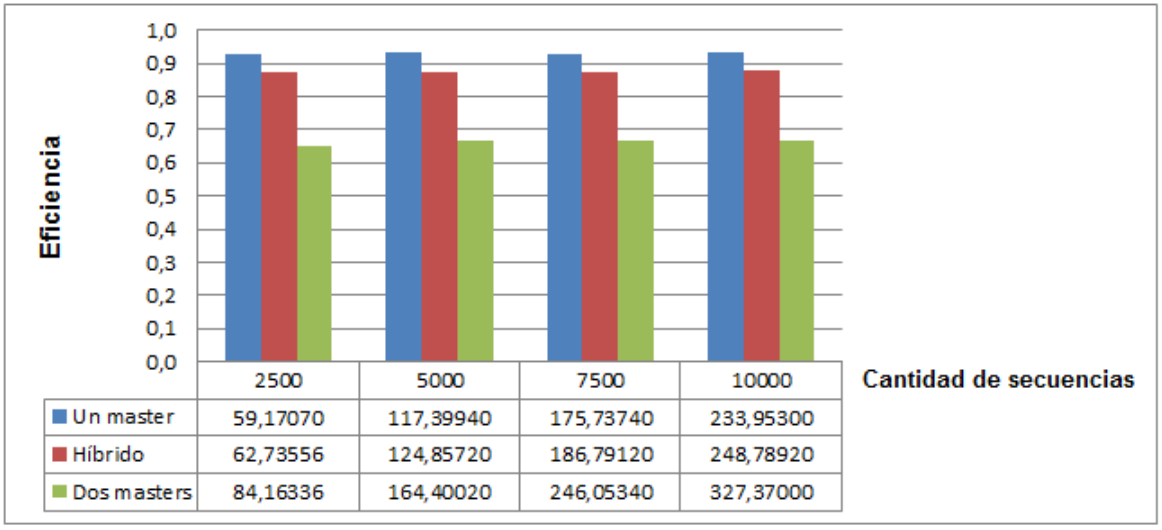
	2500	5000	7500	10000
Un master	21,50730	42,83270	64,09518	85,39354
Híbrido	22,81982	45,43670	68,03794	90,69848
Dos masters	30,85228	59,87110	89,54700	119,37560

Figura B.10. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



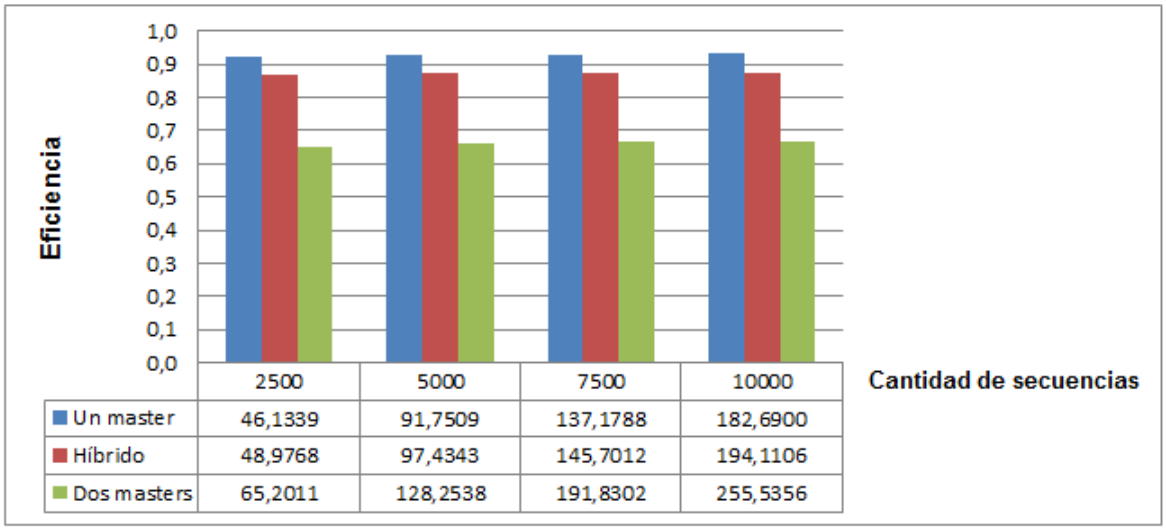
	2500	5000	7500	10000
Un master	13,12524	26,01082	38,99988	51,95902
Híbrido	13,91728	27,57042	41,39648	55,15458
Dos masters	18,60300	36,28218	54,49336	72,57192

Figura B.11. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



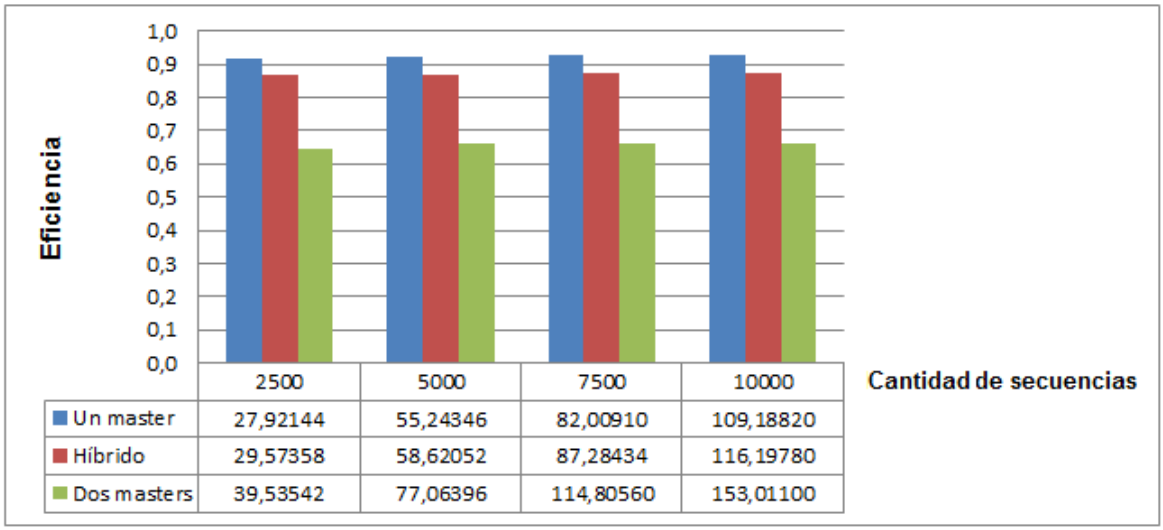
	2500	5000	7500	10000
Un master	59,17070	117,39940	175,73740	233,95300
Híbrido	62,73556	124,85720	186,79120	248,78920
Dos masters	84,16336	164,40020	246,05340	327,37000

Figura B.12. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



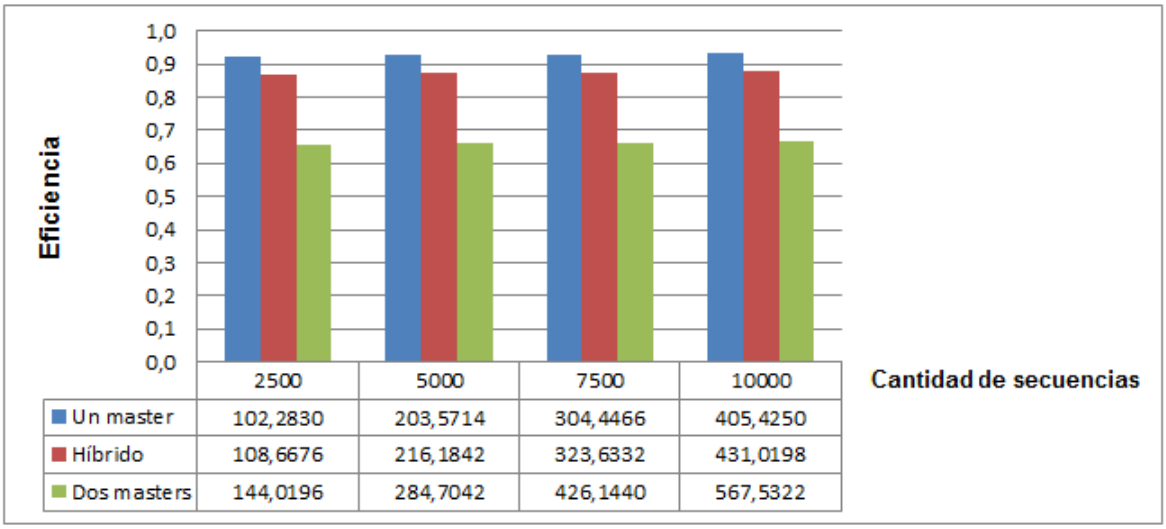
	2500	5000	7500	10000
Un master	46,1339	91,7509	137,1788	182,6900
Híbrido	48,9768	97,4343	145,7012	194,1106
Dos masters	65,2011	128,2538	191,8302	255,5356

Figura B.13. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



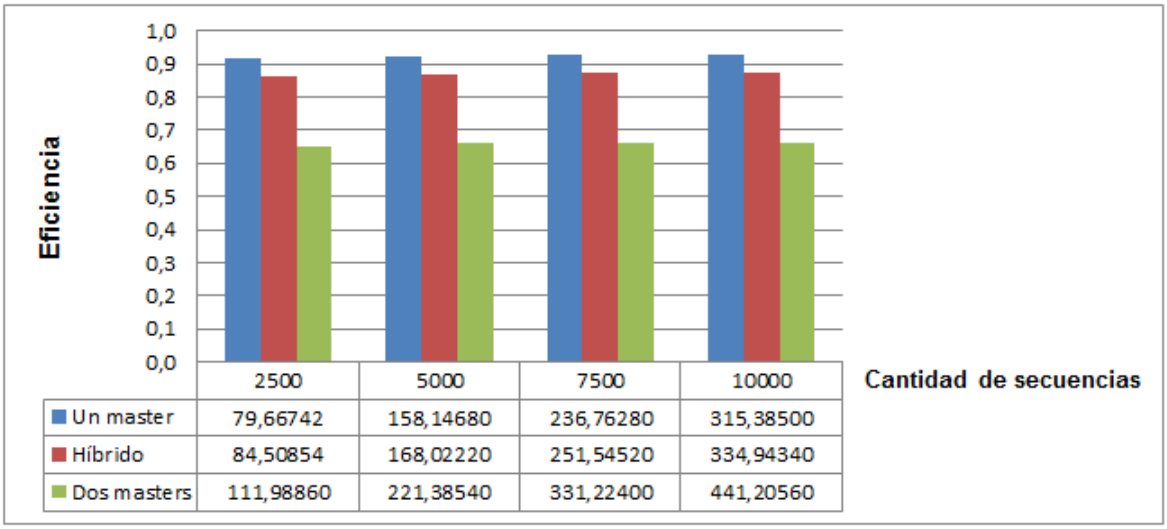
	2500	5000	7500	10000
Un master	27,92144	55,24346	82,00910	109,18820
Híbrido	29,57358	58,62052	87,28434	116,19780
Dos masters	39,53542	77,06396	114,80560	153,01100

Figura B.14. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



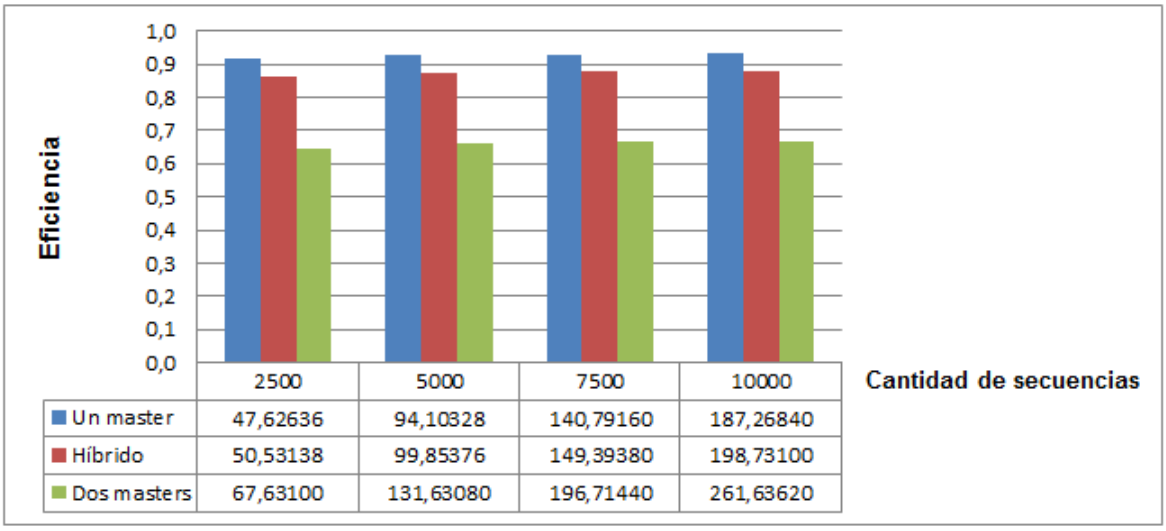
	2500	5000	7500	10000
Un master	102,2830	203,5714	304,4466	405,4250
Híbrido	108,6676	216,1842	323,6332	431,0198
Dos masters	144,0196	284,7042	426,1440	567,5322

Figura B.15. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



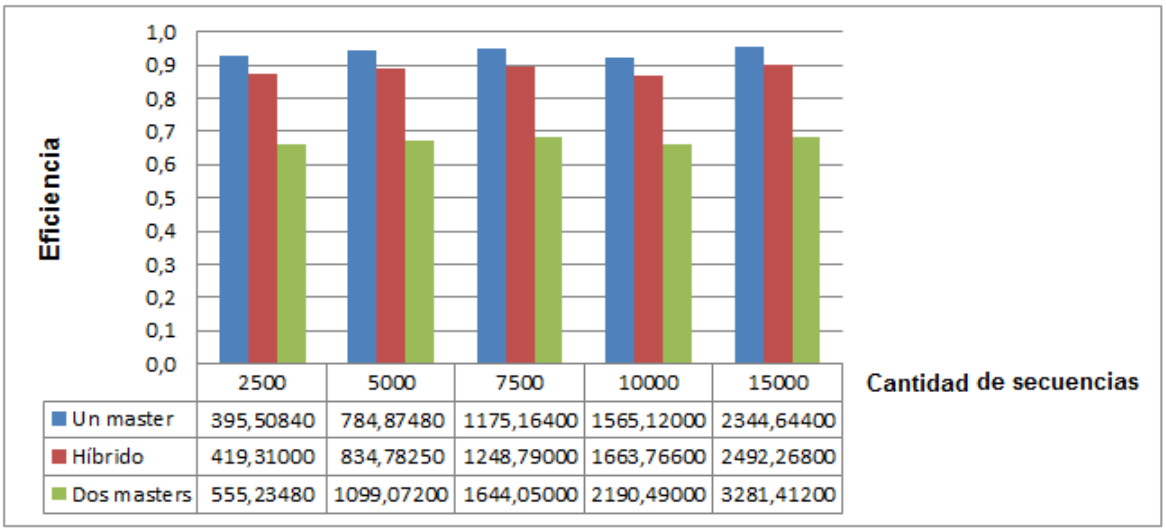
	2500	5000	7500	10000
Un master	79,66742	158,14680	236,76280	315,38500
Híbrido	84,50854	168,02220	251,54520	334,94340
Dos masters	111,98860	221,38540	331,22400	441,20560

Figura B.16. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



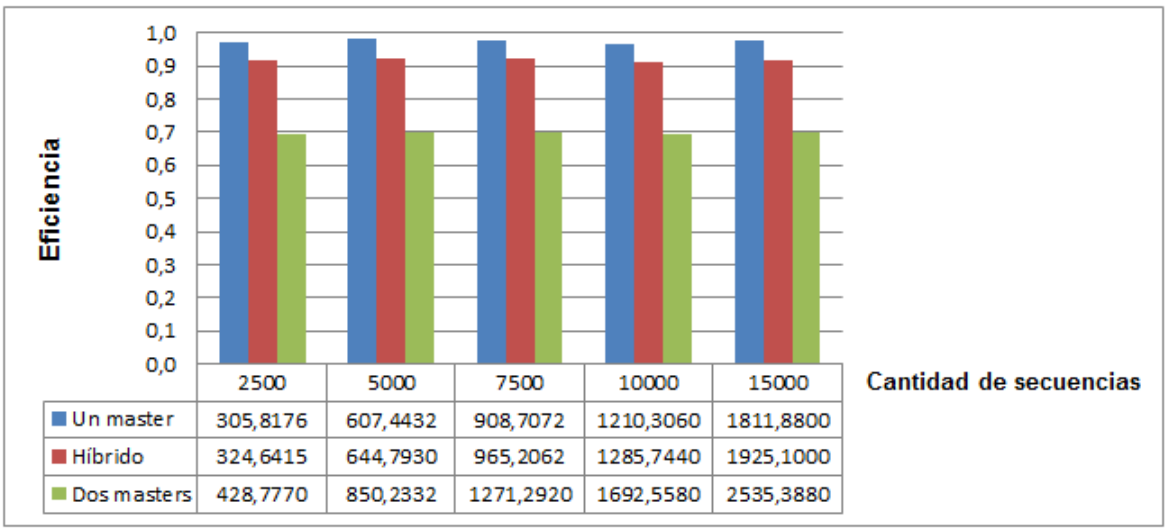
	2500	5000	7500	10000
Un master	47,62636	94,10328	140,79160	187,26840
Híbrido	50,53138	99,85376	149,39380	198,73100
Dos masters	67,63100	131,63080	196,71440	261,63620

Figura B.17. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



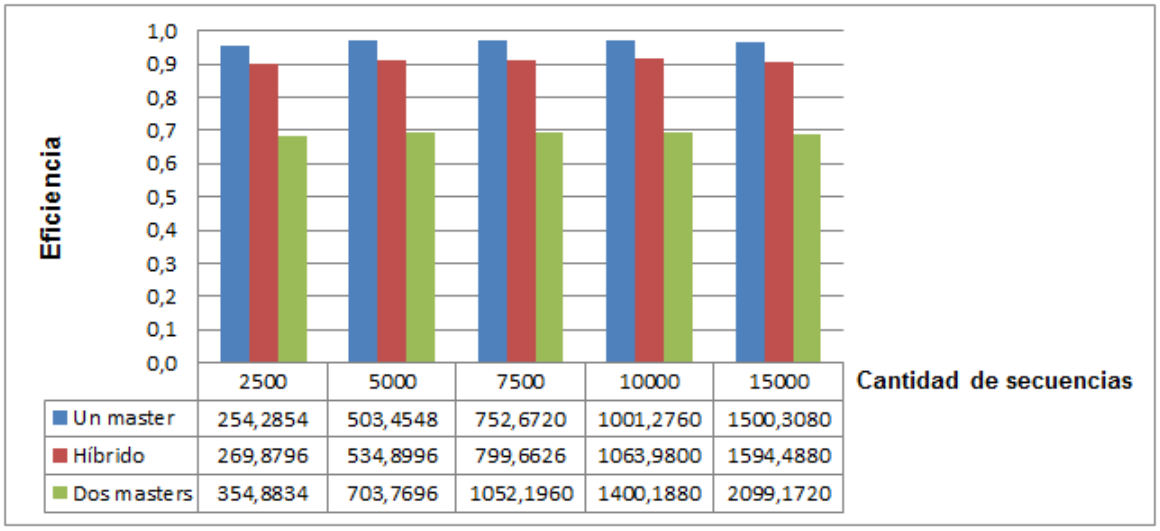
	2500	5000	7500	10000	15000
Un master	395,50840	784,87480	1175,16400	1565,12000	2344,64400
Híbrido	419,31000	834,78250	1248,79000	1663,76600	2492,26800
Dos masters	555,23480	1099,07200	1644,05000	2190,49000	3281,41200

Figura B.18. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



	2500	5000	7500	10000	15000
Un master	305,8176	607,4432	908,7072	1210,3060	1811,8800
Híbrido	324,6415	644,7930	965,2062	1285,7440	1925,1000
Dos masters	428,7770	850,2332	1271,2920	1692,5580	2535,3880

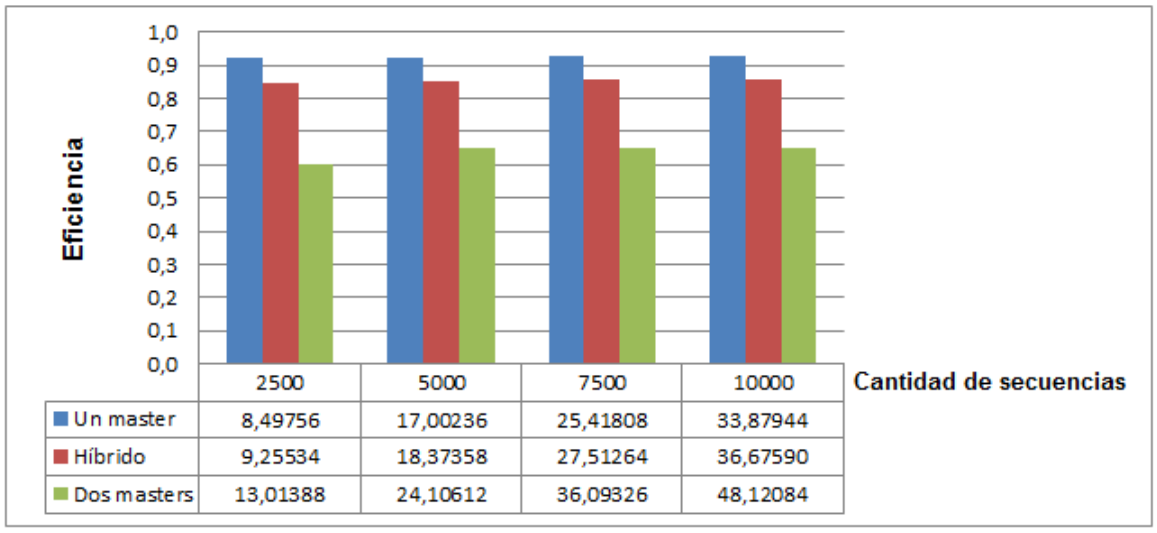
|Figura B.19. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



	2500	5000	7500	10000	15000
Un master	254,2854	503,4548	752,6720	1001,2760	1500,3080
Híbrido	269,8796	534,8996	799,6626	1063,9800	1594,4880
Dos masters	354,8834	703,7696	1052,1960	1400,1880	2099,1720

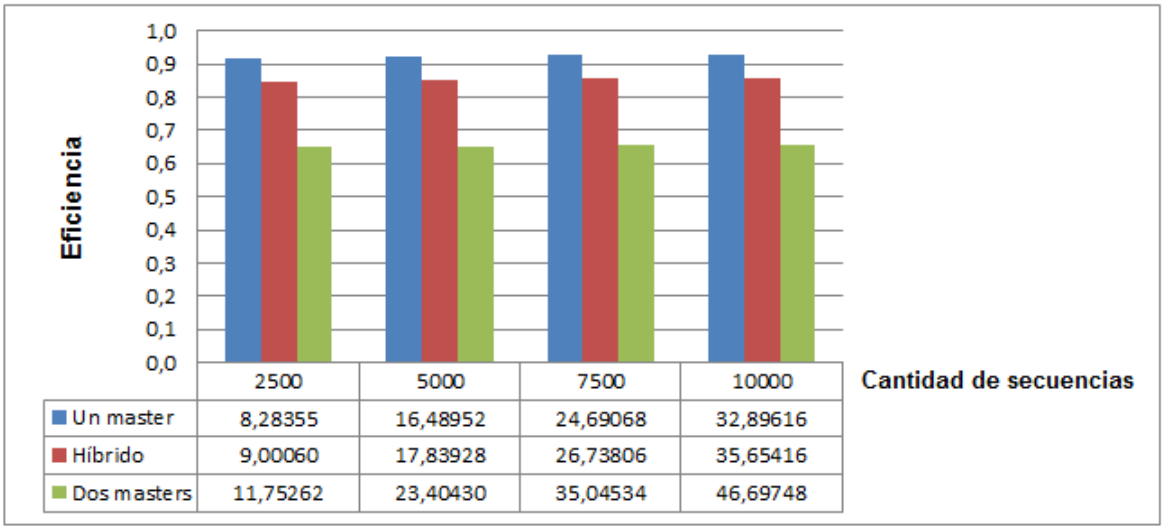
Figura B.20. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.

Resultados utilizando una arquitectura de 12 máquinas



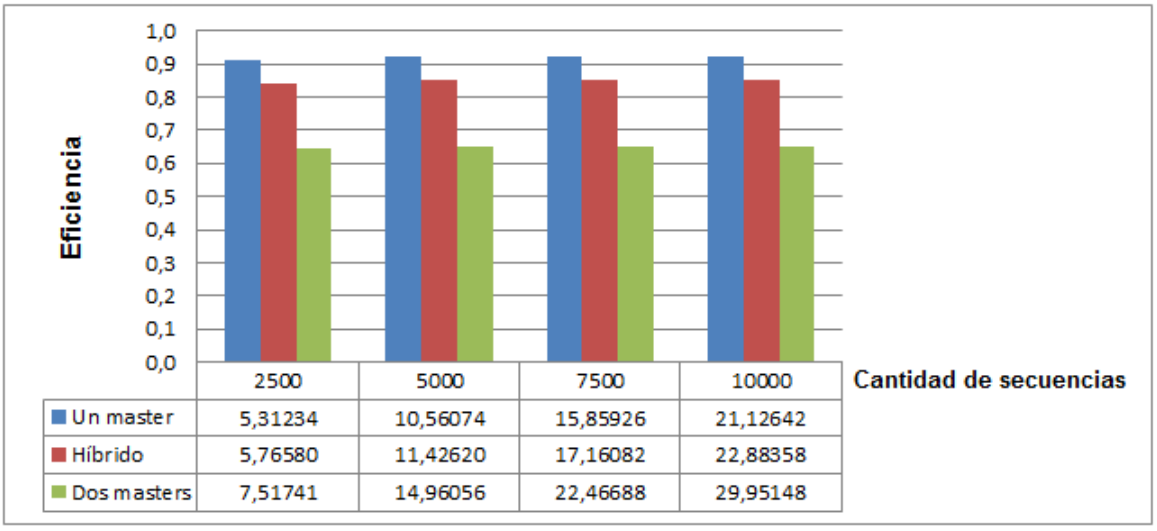
	2500	5000	7500	10000
Un master	8,49756	17,00236	25,41808	33,87944
Híbrido	9,25534	18,37358	27,51264	36,67590
Dos masters	13,01388	24,10612	36,09326	48,12084

Figura B.21. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



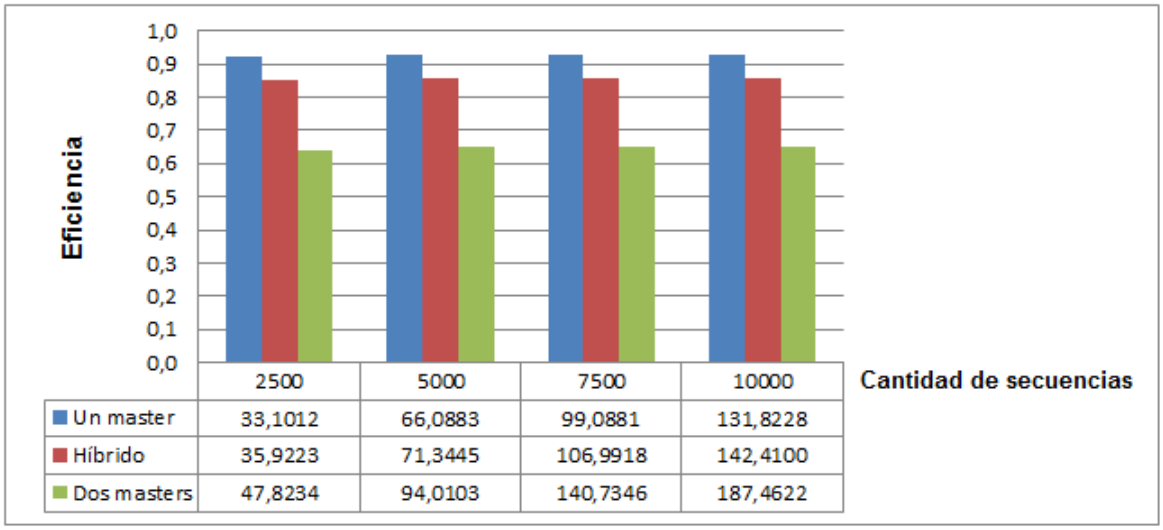
	2500	5000	7500	10000
Un master	8,28355	16,48952	24,69068	32,89616
Híbrido	9,00060	17,83928	26,73806	35,65416
Dos masters	11,75262	23,40430	35,04534	46,69748

Figura B.22. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



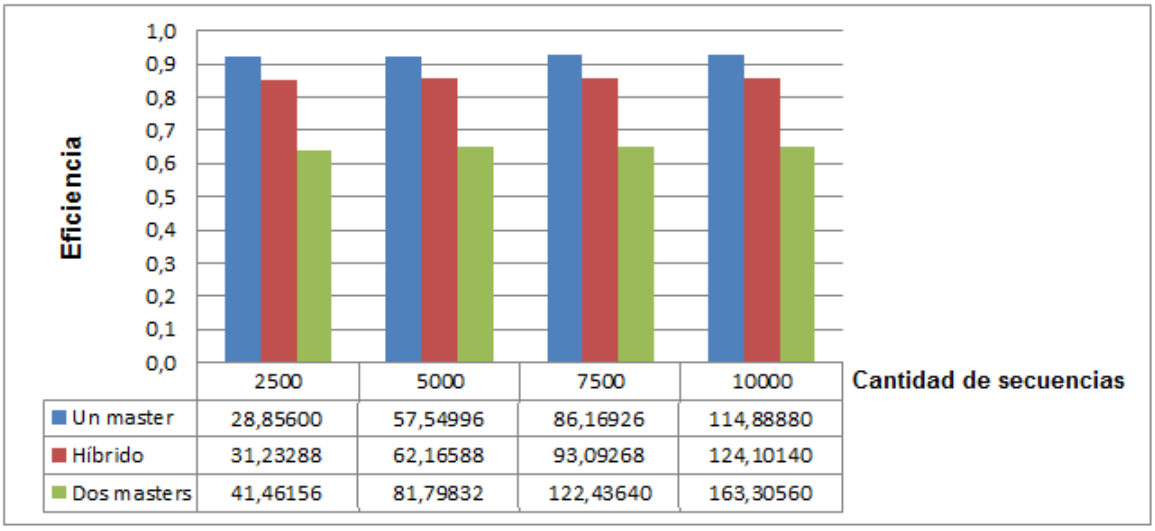
	2500	5000	7500	10000
Un master	5,31234	10,56074	15,85926	21,12642
Híbrido	5,76580	11,42620	17,16082	22,88358
Dos masters	7,51741	14,96056	22,46688	29,95148

Figura B.23. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



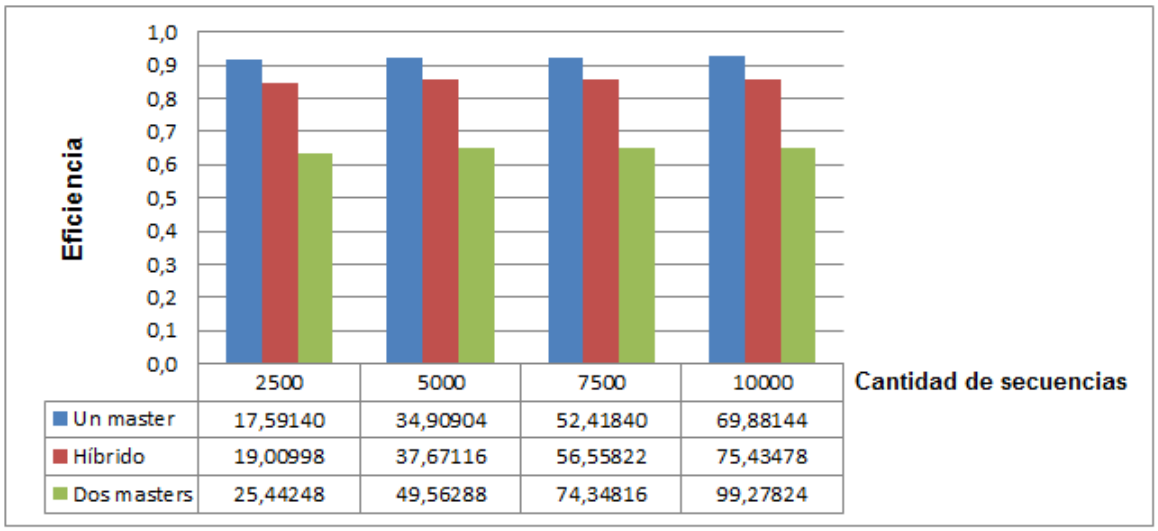
	2500	5000	7500	10000
Un master	33,1012	66,0883	99,0881	131,8228
Híbrido	35,9223	71,3445	106,9918	142,4100
Dos masters	47,8234	94,0103	140,7346	187,4622

Figura B.24. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



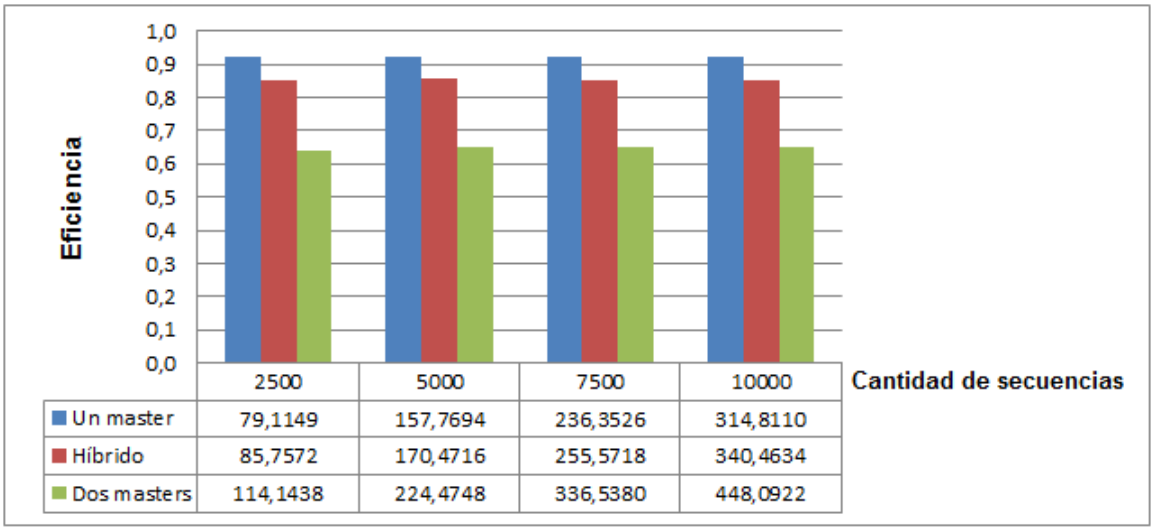
	2500	5000	7500	10000
Un master	28,85600	57,54996	86,16926	114,88880
Híbrido	31,23288	62,16588	93,09268	124,10140
Dos masters	41,46156	81,79832	122,43640	163,30560

Figura B.25. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



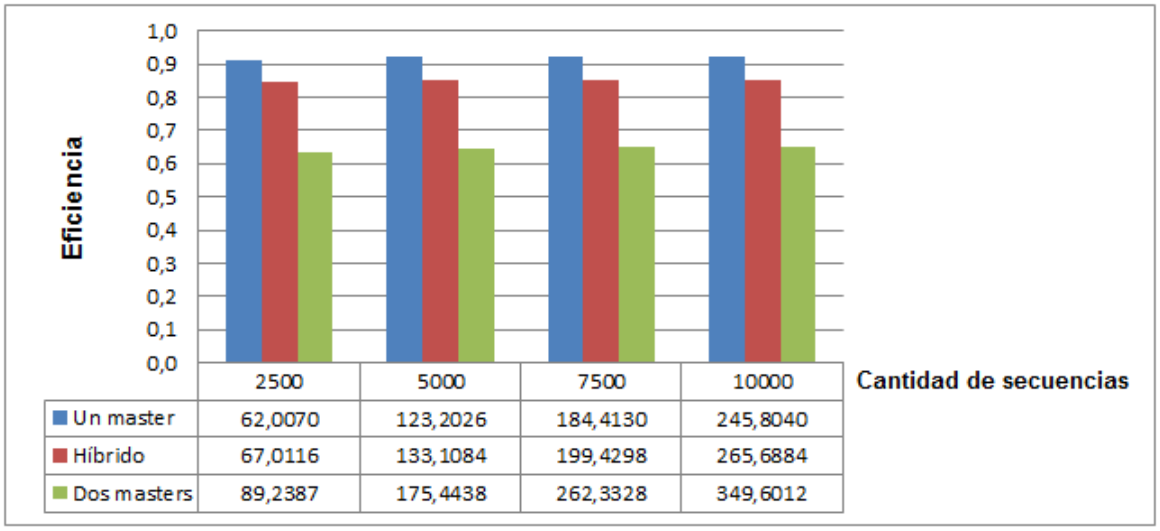
	2500	5000	7500	10000
Un master	17,59140	34,90904	52,41840	69,88144
Híbrido	19,00998	37,67116	56,55822	75,43478
Dos masters	25,44248	49,56288	74,34816	99,27824

Figura B.26. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



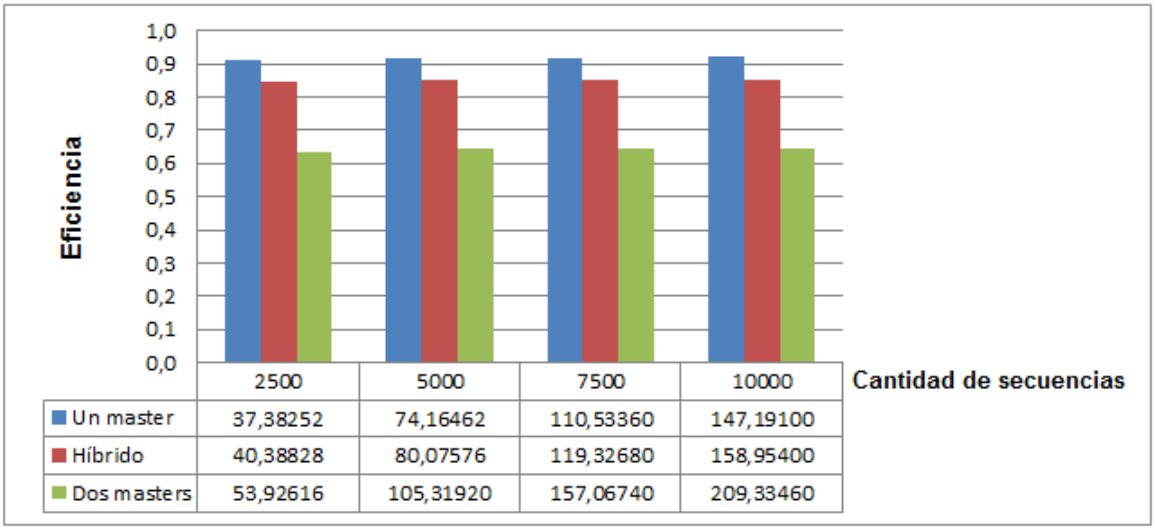
	2500	5000	7500	10000
Un master	79,1149	157,7694	236,3526	314,8110
Híbrido	85,7572	170,4716	255,5718	340,4634
Dos masters	114,1438	224,4748	336,5380	448,0922

Figura B.27. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



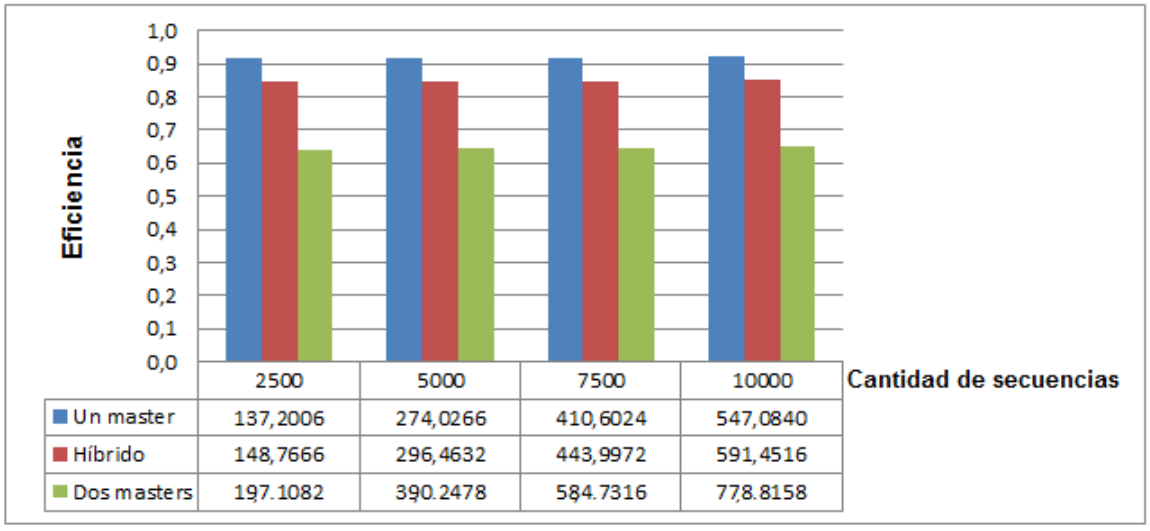
	2500	5000	7500	10000
Un master	62,0070	123,2026	184,4130	245,8040
Híbrido	67,0116	133,1084	199,4298	265,6884
Dos masters	89,2387	175,4438	262,3328	349,6012

Figura B.28. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



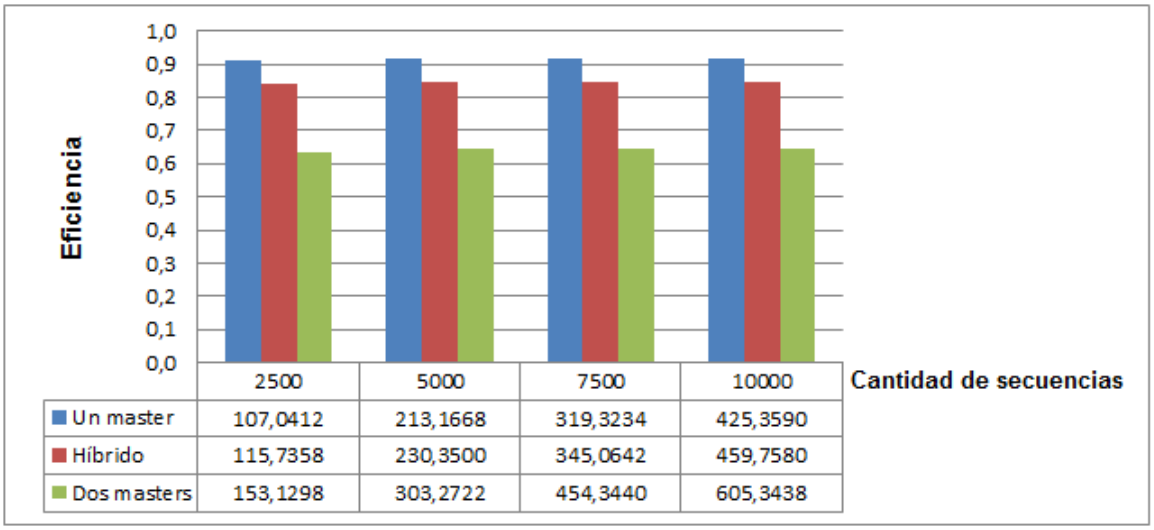
	2500	5000	7500	10000
Un master	37,38252	74,16462	110,53360	147,19100
Híbrido	40,38828	80,07576	119,32680	158,95400
Dos masters	53,92616	105,31920	157,06740	209,33460

Figura B.29. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



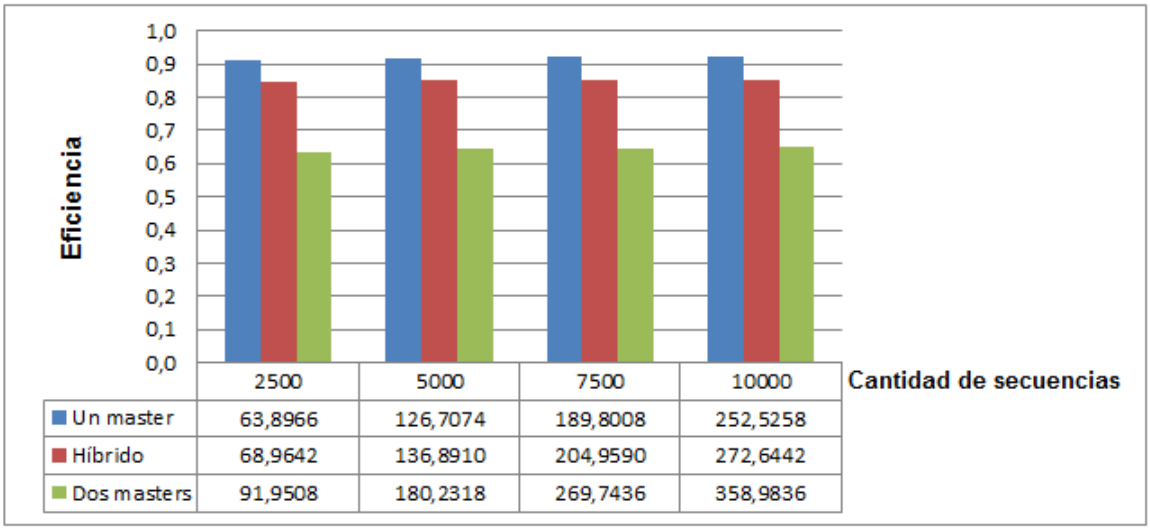
	2500	5000	7500	10000
Un master	137,2006	274,0266	410,6024	547,0840
Híbrido	148,7666	296,4632	443,9972	591,4516
Dos masters	197.1082	390.2478	584.7316	778.8158

Figura B.30. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



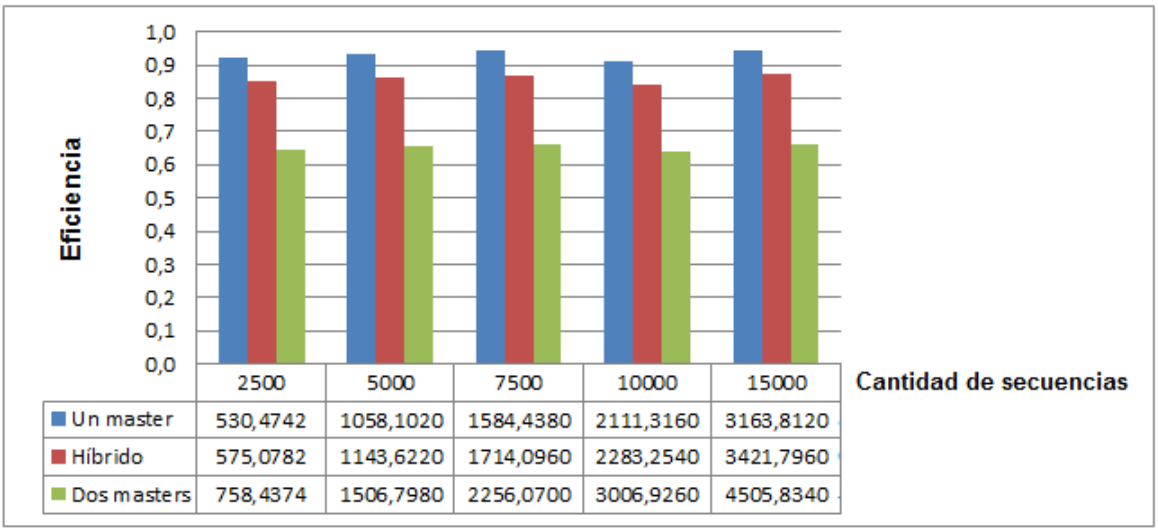
	2500	5000	7500	10000
Un master	107,0412	213,1668	319,3234	425,3590
Híbrido	115,7358	230,3500	345,0642	459,7580
Dos masters	153,1298	303,2722	454,3440	605,3438

Figura B.31. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



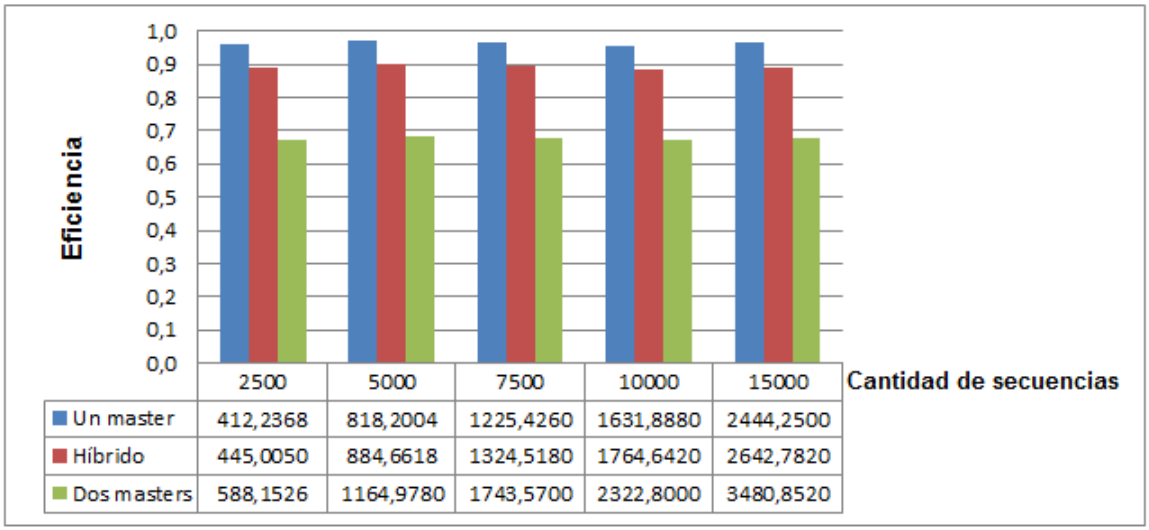
	2500	5000	7500	10000
Un master	63,8966	126,7074	189,8008	252,5258
Híbrido	68,9642	136,8910	204,9590	272,6442
Dos masters	91,9508	180,2318	269,7436	358,9836

Figura B.32. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



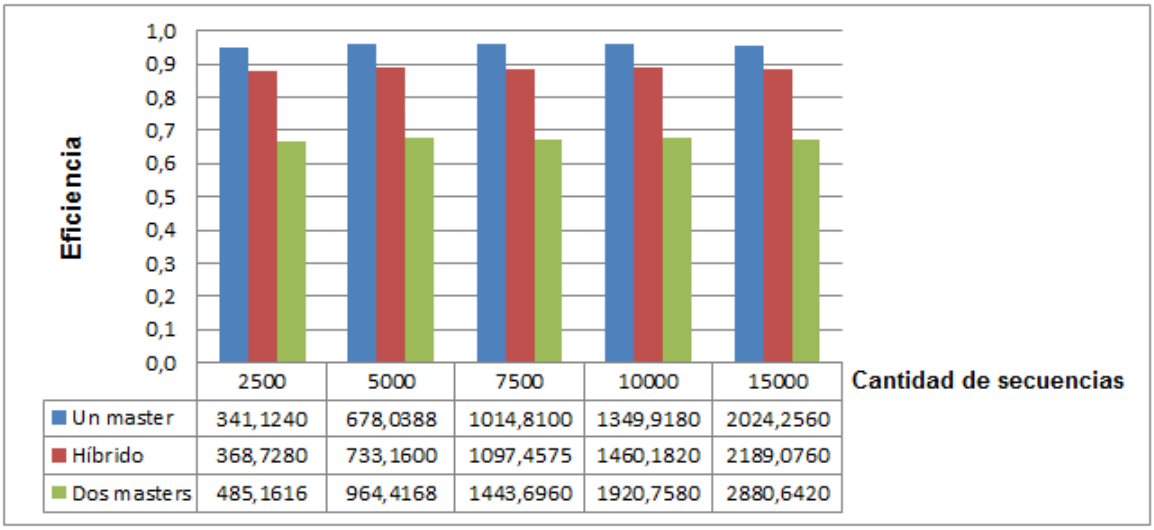
	2500	5000	7500	10000	15000
Un master	530,4742	1058,1020	1584,4380	2111,3160	3163,8120
Híbrido	575,0782	1143,6220	1714,0960	2283,2540	3421,7960
Dos masters	758,4374	1506,7980	2256,0700	3006,9260	4505,8340

Figura B.33. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



	2500	5000	7500	10000	15000
Un master	412,2368	818,2004	1225,4260	1631,8880	2444,2500
Híbrido	445,0050	884,6618	1324,5180	1764,6420	2642,7820
Dos masters	588,1526	1164,9780	1743,5700	2322,8000	3480,8520

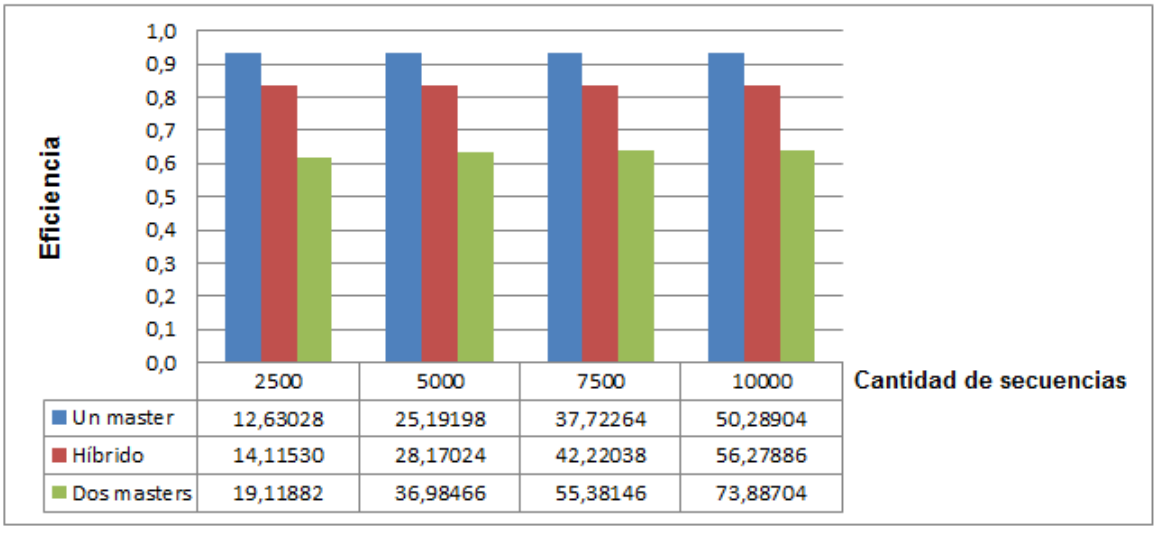
Figura B.34. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



	2500	5000	7500	10000	15000
Un master	341,1240	678,0388	1014,8100	1349,9180	2024,2560
Híbrido	368,7280	733,1600	1097,4575	1460,1820	2189,0760
Dos masters	485,1616	964,4168	1443,6960	1920,7580	2880,6420

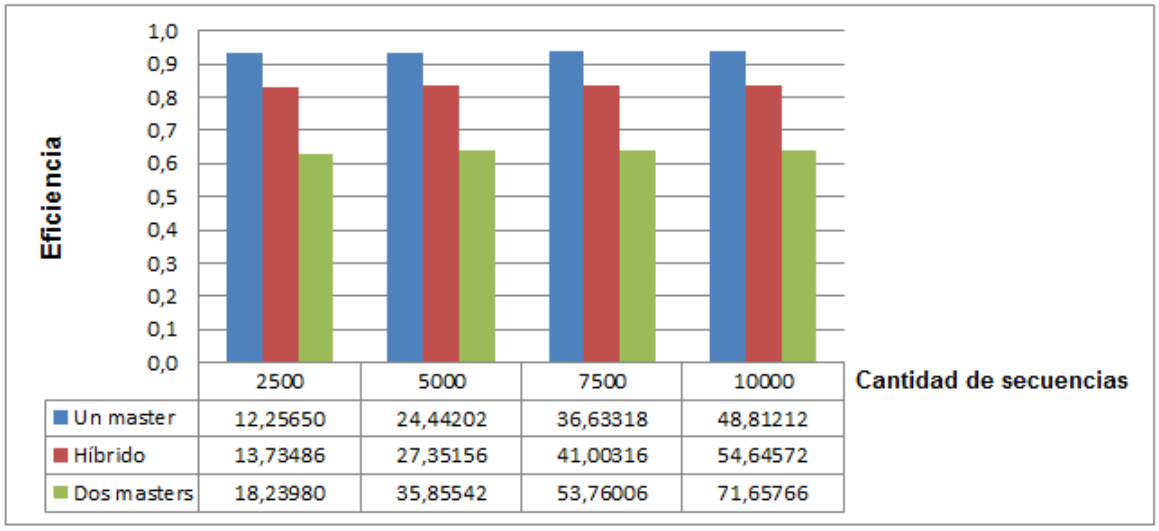
Figura B.35. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.

Resultados utilizando una arquitectura de 8 máquinas



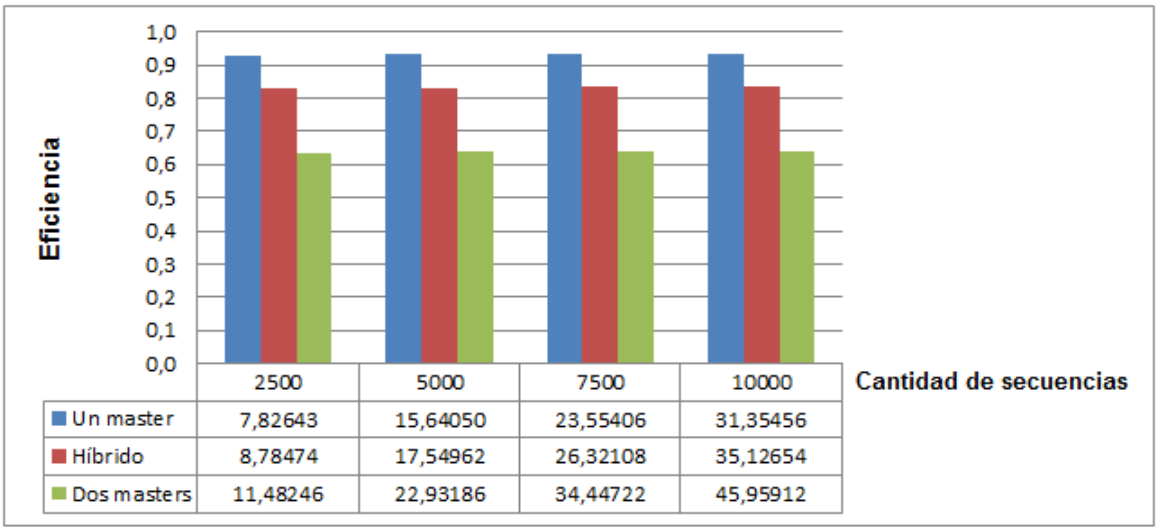
	2500	5000	7500	10000
Un master	12,63028	25,19198	37,72264	50,28904
Híbrido	14,11530	28,17024	42,22038	56,27886
Dos masters	19,11882	36,98466	55,38146	73,88704

Figura B.36. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



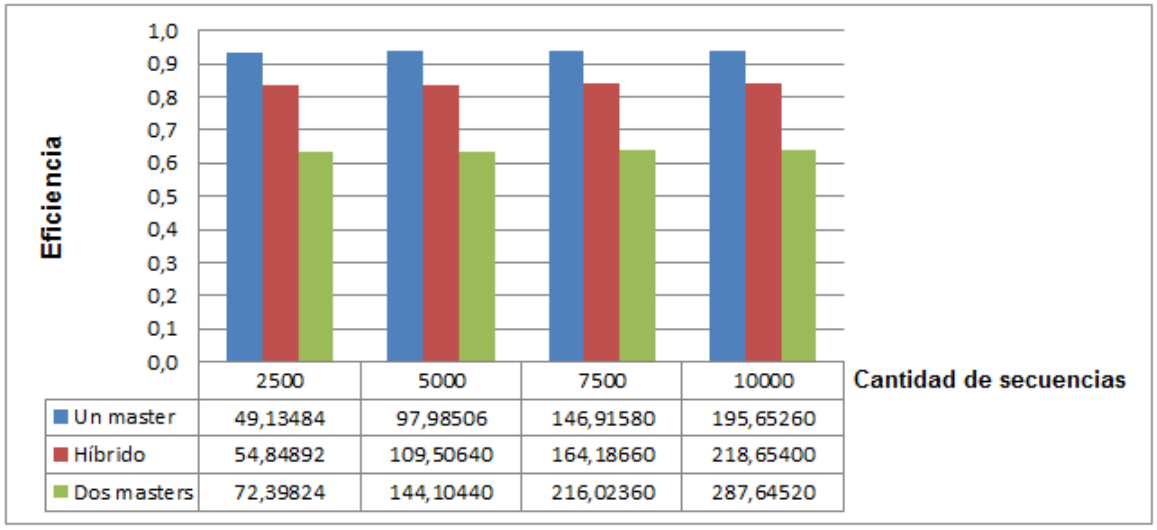
	2500	5000	7500	10000
Un master	12,25650	24,44202	36,63318	48,81212
Híbrido	13,73486	27,35156	41,00316	54,64572
Dos masters	18,23980	35,85542	53,76006	71,65766

Figura B.37. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



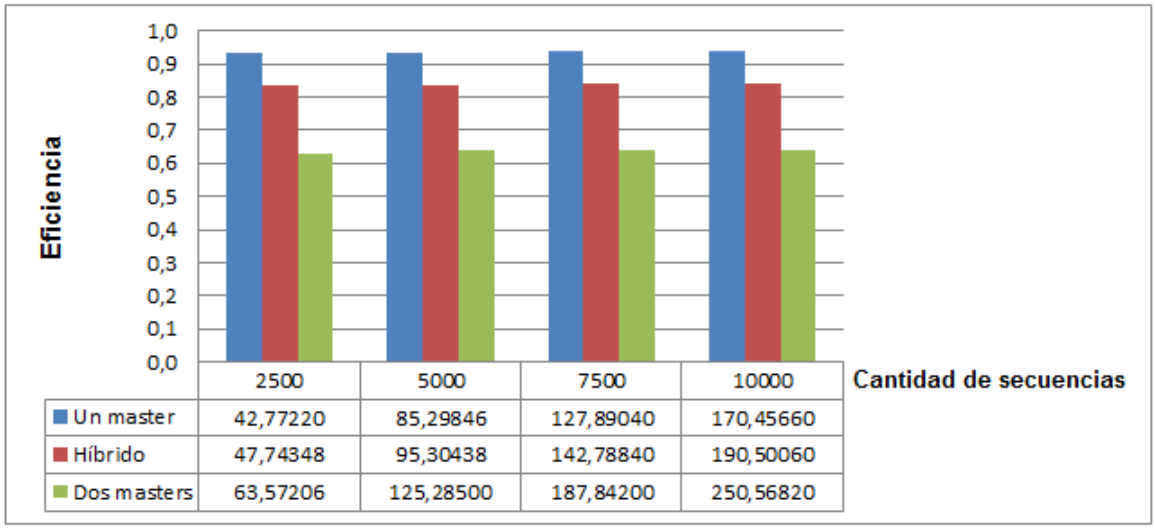
	2500	5000	7500	10000
Un master	7,82643	15,64050	23,55406	31,35456
Híbrido	8,78474	17,54962	26,32108	35,12654
Dos masters	11,48246	22,93186	34,44722	45,95912

Figura B.38. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



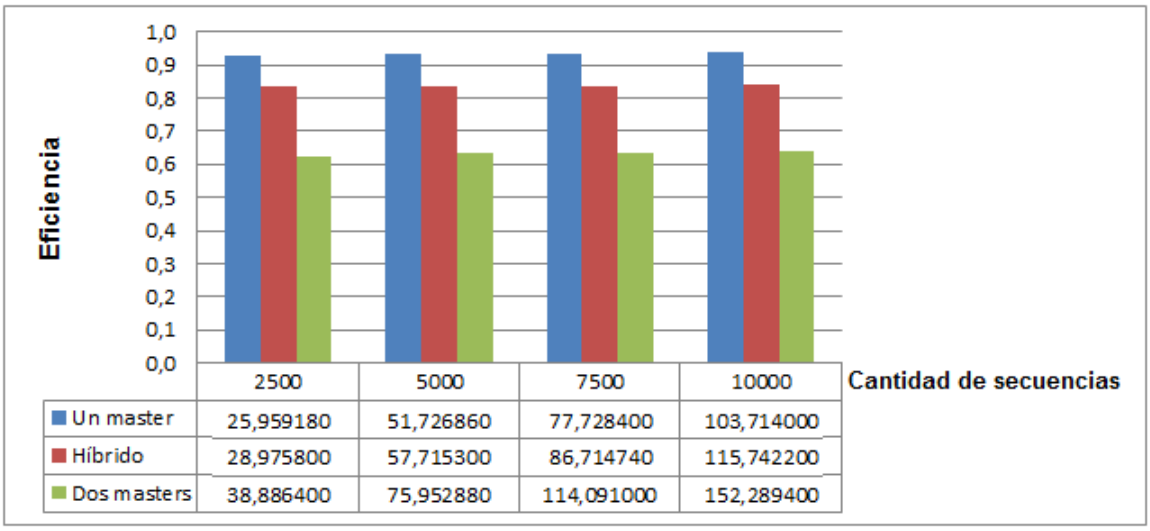
	2500	5000	7500	10000
Un master	49,13484	97,98506	146,91580	195,65260
Híbrido	54,84892	109,50640	164,18660	218,65400
Dos masters	72,39824	144,10440	216,02360	287,64520

Figura B.39. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



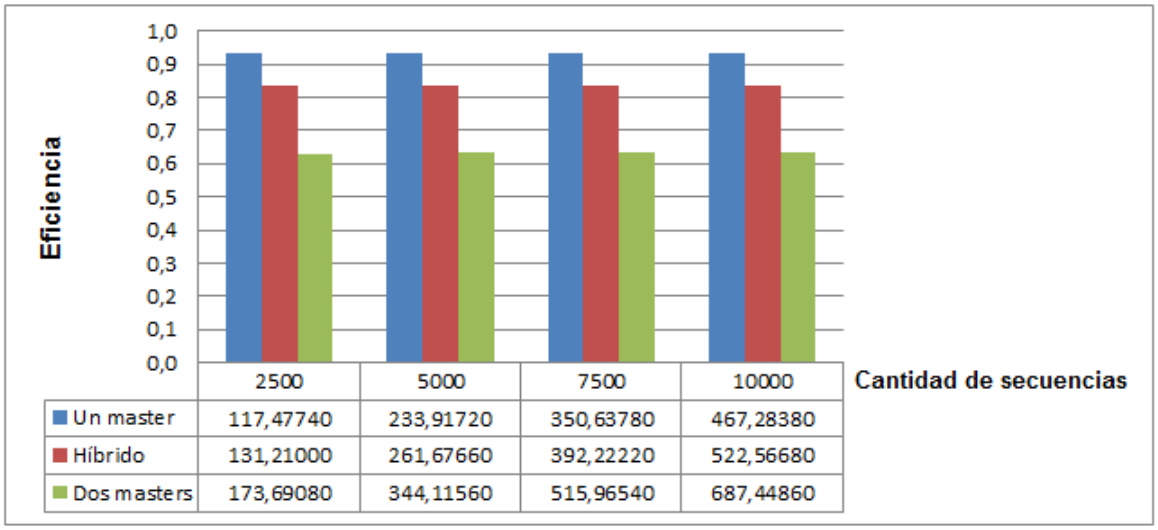
	2500	5000	7500	10000
Un master	42,77220	85,29846	127,89040	170,45660
Híbrido	47,74348	95,30438	142,78840	190,50060
Dos masters	63,57206	125,28500	187,84200	250,56820

Figura B.40. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



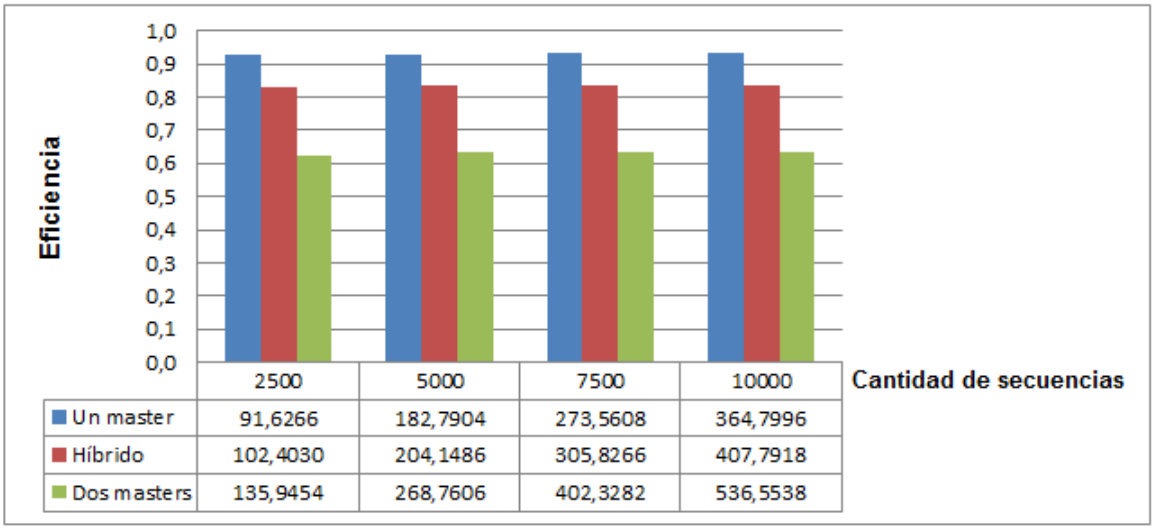
	2500	5000	7500	10000
Un master	25,959180	51,726860	77,728400	103,714000
Híbrido	28,975800	57,715300	86,714740	115,742200
Dos masters	38,886400	75,952880	114,091000	152,289400

Figura B.41. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



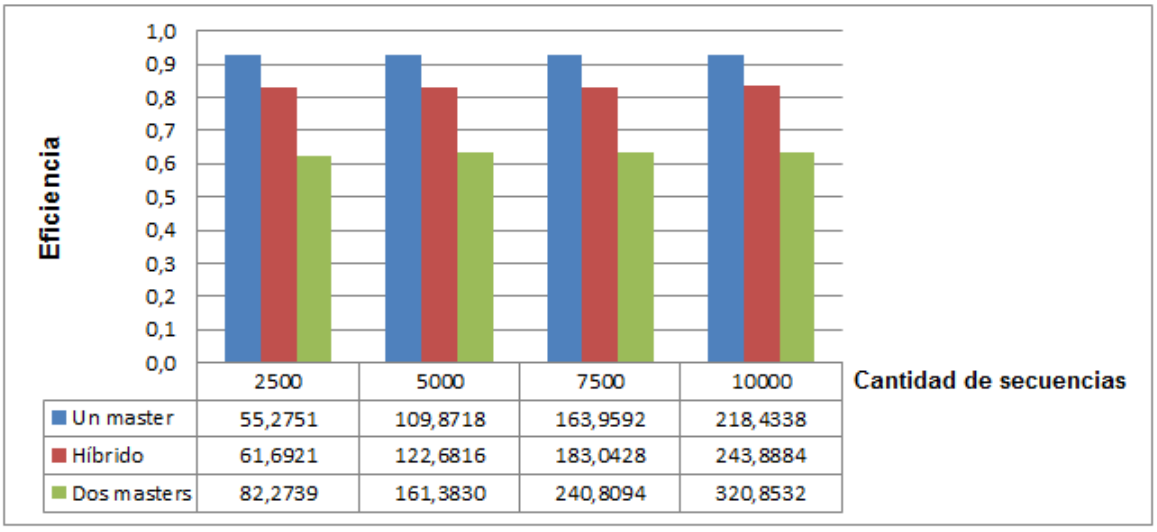
	2500	5000	7500	10000
Un master	117,47740	233,91720	350,63780	467,28380
Híbrido	131,21000	261,67660	392,22220	522,56680
Dos masters	173,69080	344,11560	515,96540	687,44860

Figura B.42. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



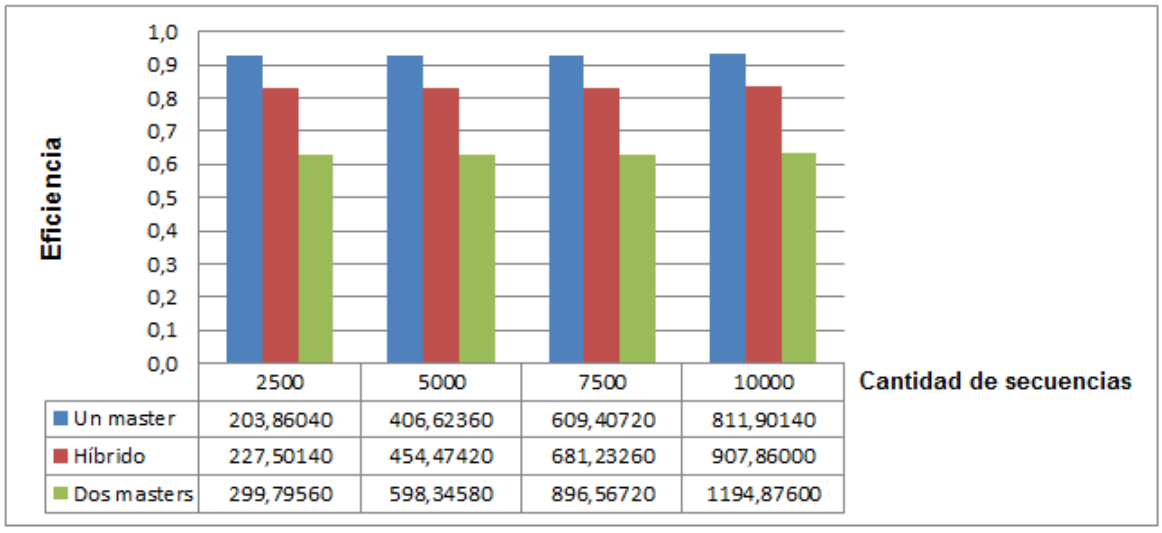
	2500	5000	7500	10000
Un master	91,6266	182,7904	273,5608	364,7996
Híbrido	102,4030	204,1486	305,8266	407,7918
Dos masters	135,9454	268,7606	402,3282	536,5538

Figura B.43. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



	2500	5000	7500	10000
Un master	55,2751	109,8718	163,9592	218,4338
Híbrido	61,6921	122,6816	183,0428	243,8884
Dos masters	82,2739	161,3830	240,8094	320,8532

Figura B.44. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



	2500	5000	7500	10000
Un master	203,86040	406,62360	609,40720	811,90140
Híbrido	227,50140	454,47420	681,23260	907,86000
Dos masters	299,79560	598,34580	896,56720	1194,87600

Figura B.45. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.

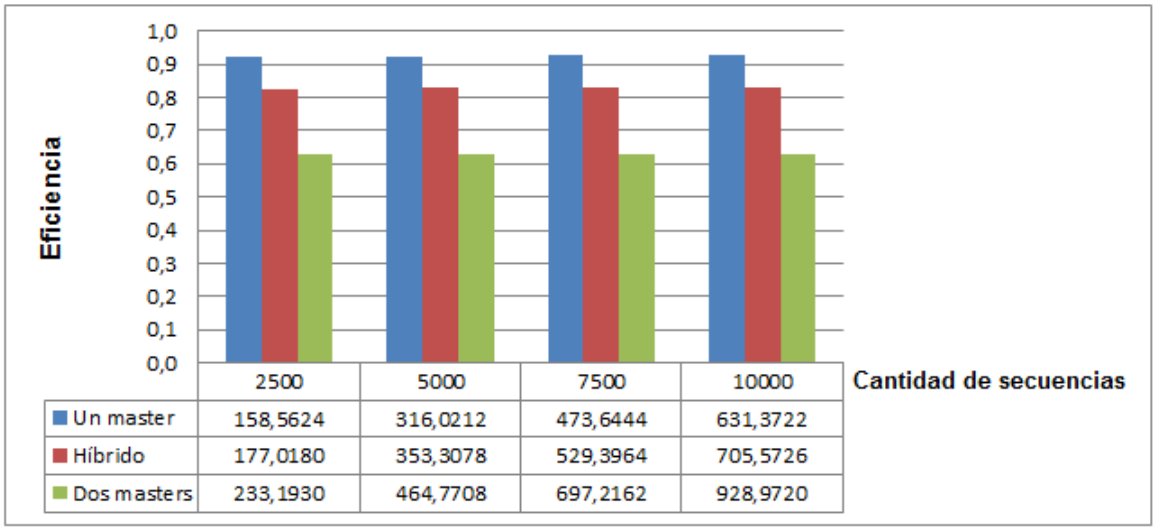


Figura B.46. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.

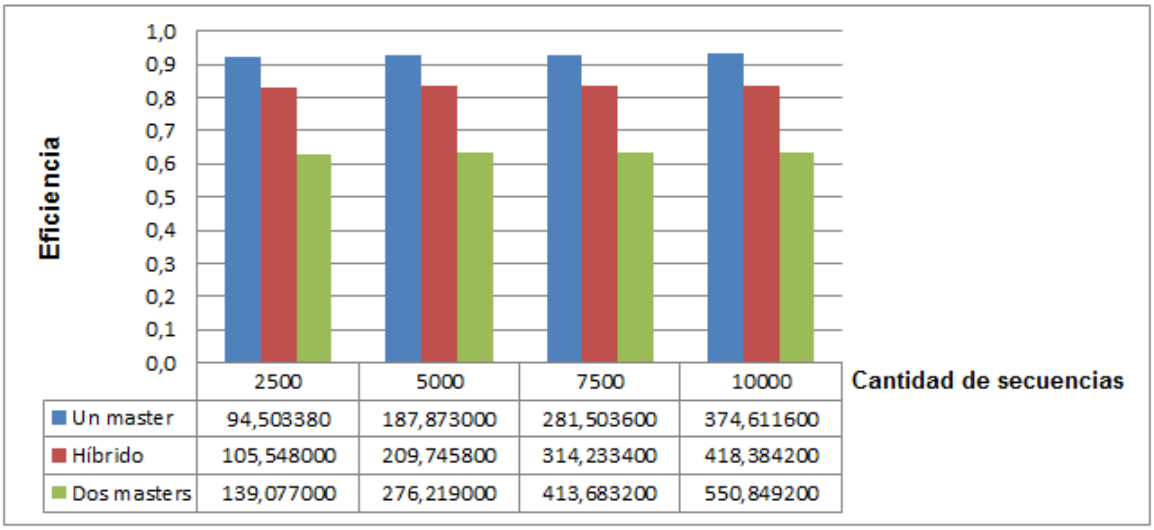


Figura B.47. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.

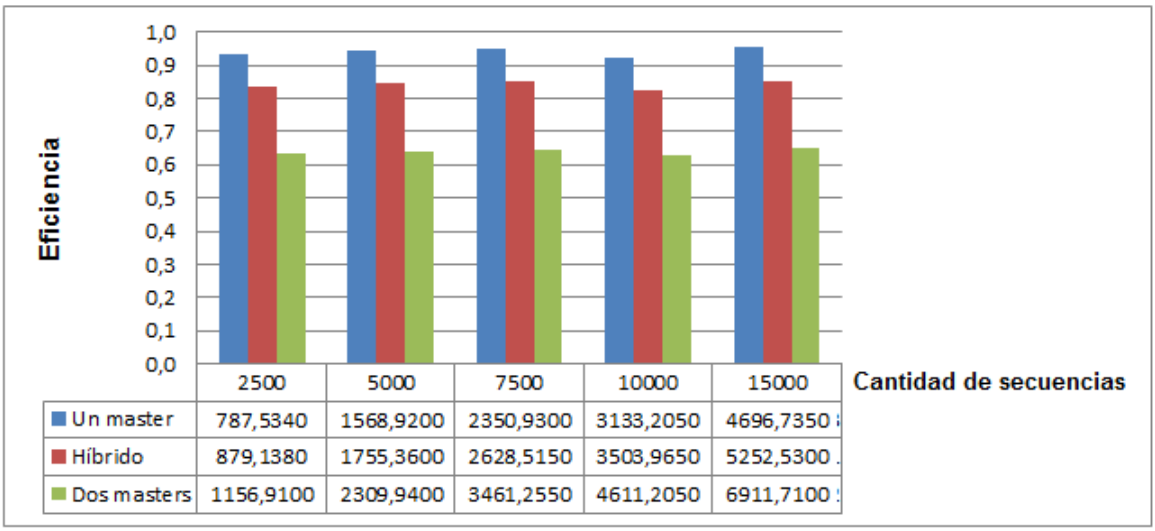
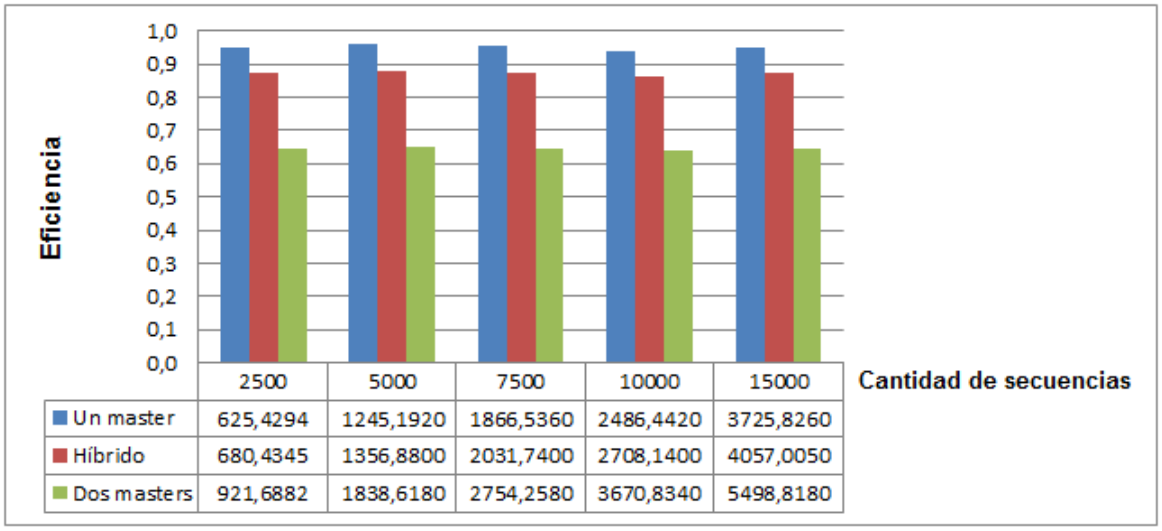
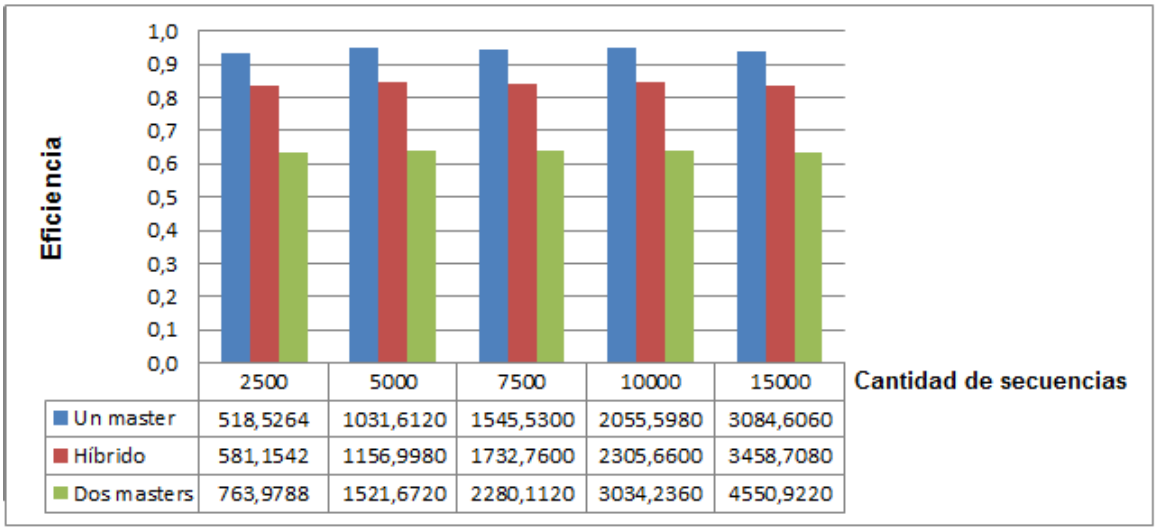


Figura B.48. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



	2500	5000	7500	10000	15000
Un master	625,4294	1245,1920	1866,5360	2486,4420	3725,8260
Híbrido	680,4345	1356,8800	2031,7400	2708,1400	4057,0050
Dos masters	921,6882	1838,6180	2754,2580	3670,8340	5498,8180

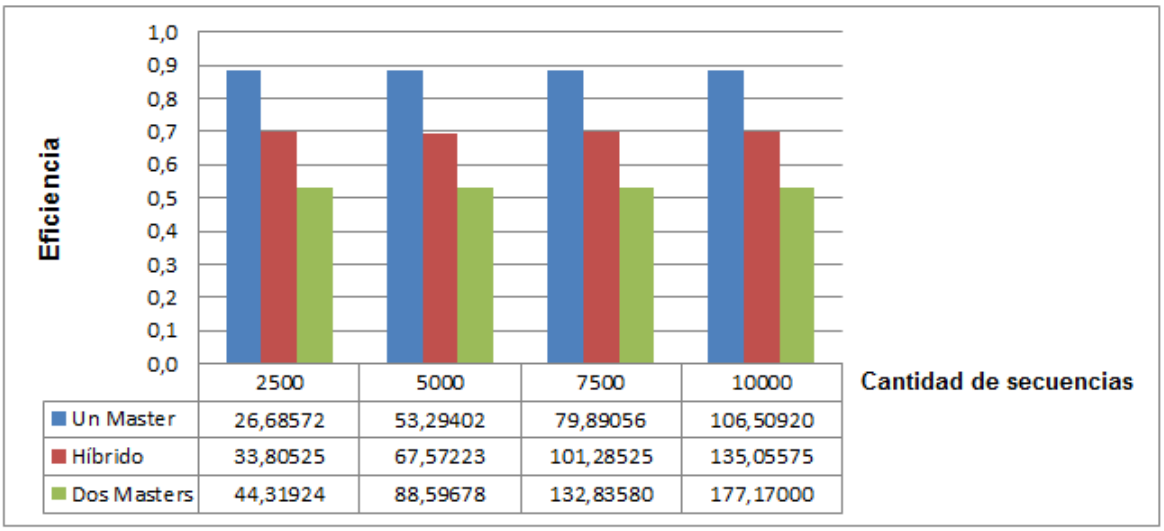
Figura B.49. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



	2500	5000	7500	10000	15000
Un master	518,5264	1031,6120	1545,5300	2055,5980	3084,6060
Híbrido	581,1542	1156,9980	1732,7600	2305,6600	3458,7080
Dos masters	763,9788	1521,6720	2280,1120	3034,2360	4550,9220

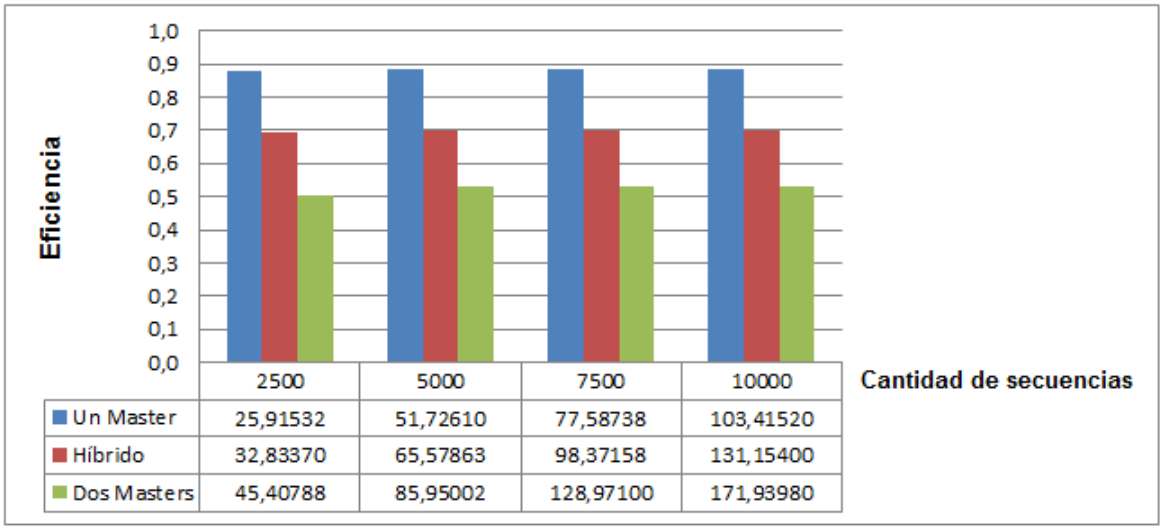
Figura B.50. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.

Resultados utilizando una arquitectura de 4 máquinas



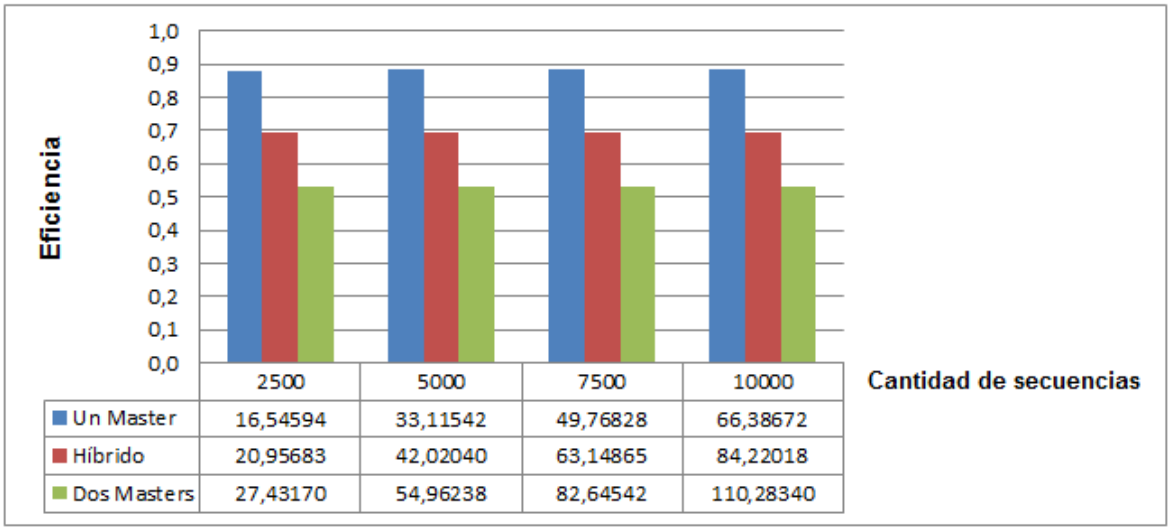
	2500	5000	7500	10000
Un Master	26,68572	53,29402	79,89056	106,50920
Híbrido	33,80525	67,57223	101,28525	135,05575
Dos Masters	44,31924	88,59678	132,83580	177,17000

Figura B.51. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



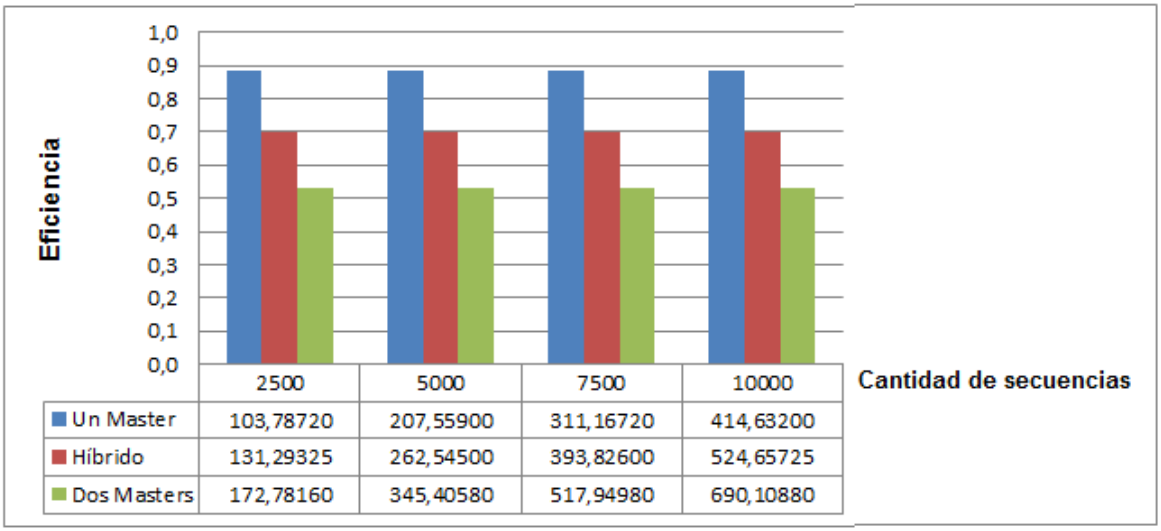
	2500	5000	7500	10000
Un Master	25,91532	51,72610	77,58738	103,41520
Híbrido	32,83370	65,57863	98,37158	131,15400
Dos Masters	45,40788	85,95002	128,97100	171,93980

Figura B.52. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



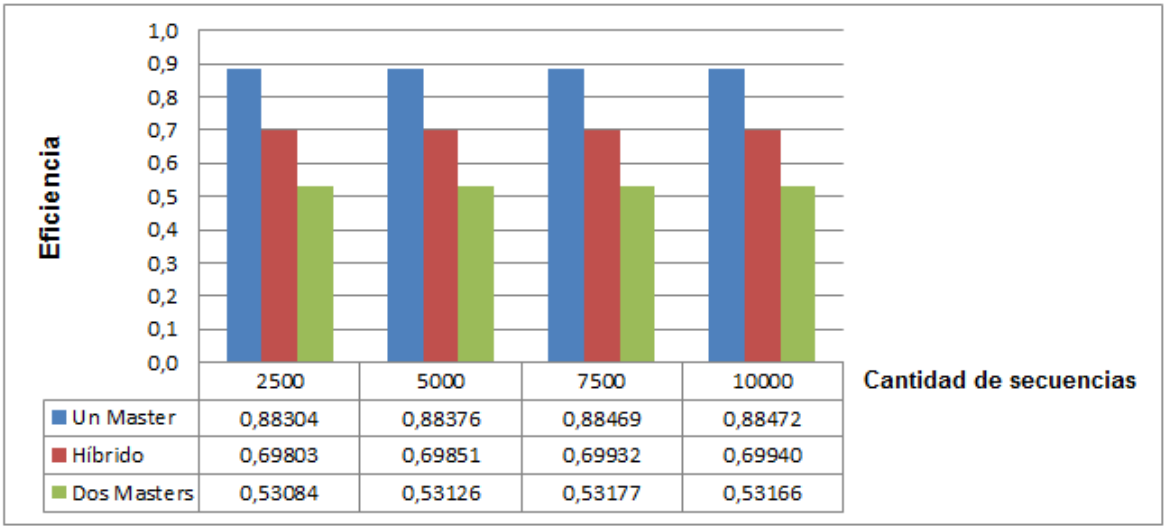
	2500	5000	7500	10000
Un Master	16,54594	33,11542	49,76828	66,38672
Híbrido	20,95683	42,02040	63,14865	84,22018
Dos Masters	27,43170	54,96238	82,64542	110,28340

Figura B.53. El gráfico representa la eficiencia usando secuencias de longitud 2500 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



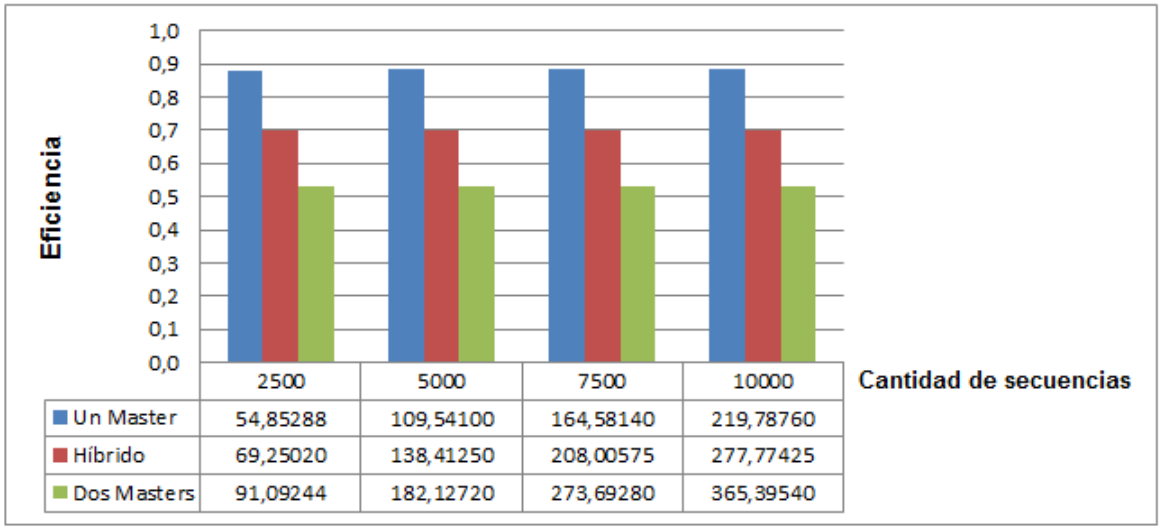
	2500	5000	7500	10000
Un Master	103,78720	207,55900	311,16720	414,63200
Híbrido	131,29325	262,54500	393,82600	524,65725
Dos Masters	172,78160	345,40580	517,94980	690,10880

Figura B.54. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



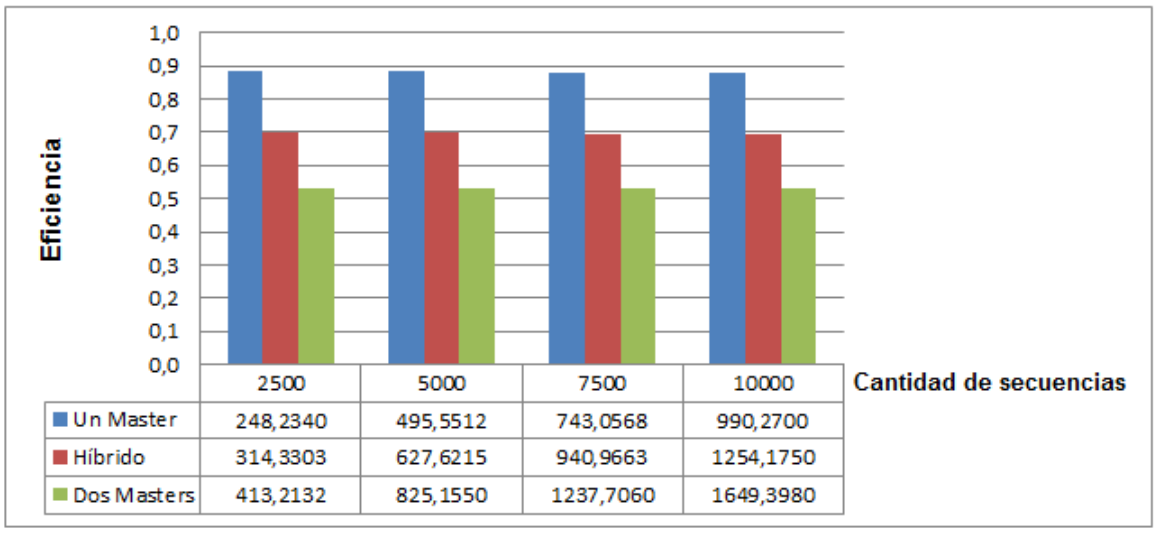
	2500	5000	7500	10000
Un Master	0,88304	0,88376	0,88469	0,88472
Híbrido	0,69803	0,69851	0,69932	0,69940
Dos Masters	0,53084	0,53126	0,53177	0,53166

Figura B.55. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



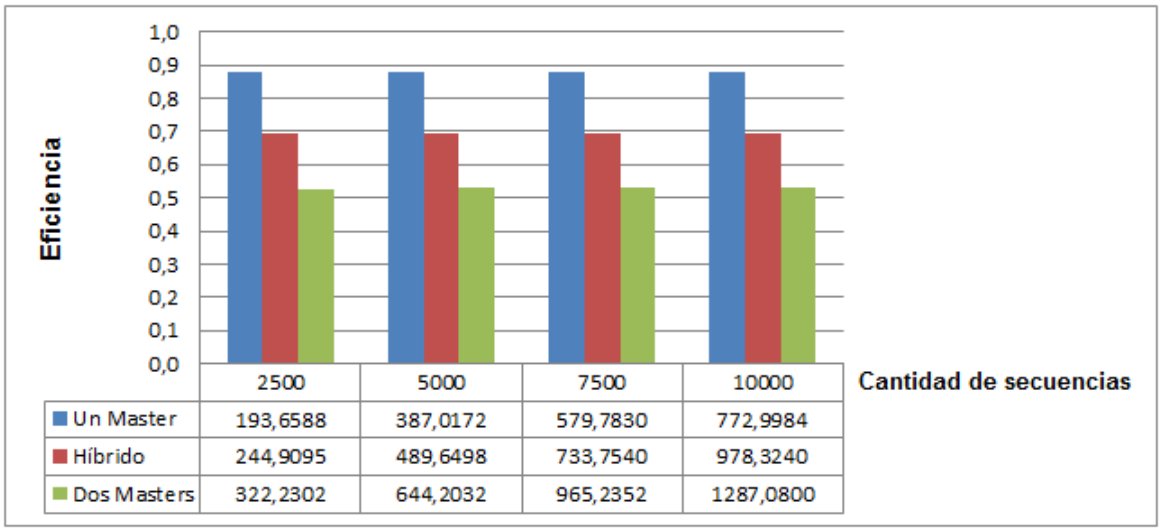
	2500	5000	7500	10000
Un Master	54,85288	109,54100	164,58140	219,78760
Híbrido	69,25020	138,41250	208,00575	277,77425
Dos Masters	91,09244	182,12720	273,69280	365,39540

Figura B.56. El gráfico representa la eficiencia usando secuencias de longitud 5000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



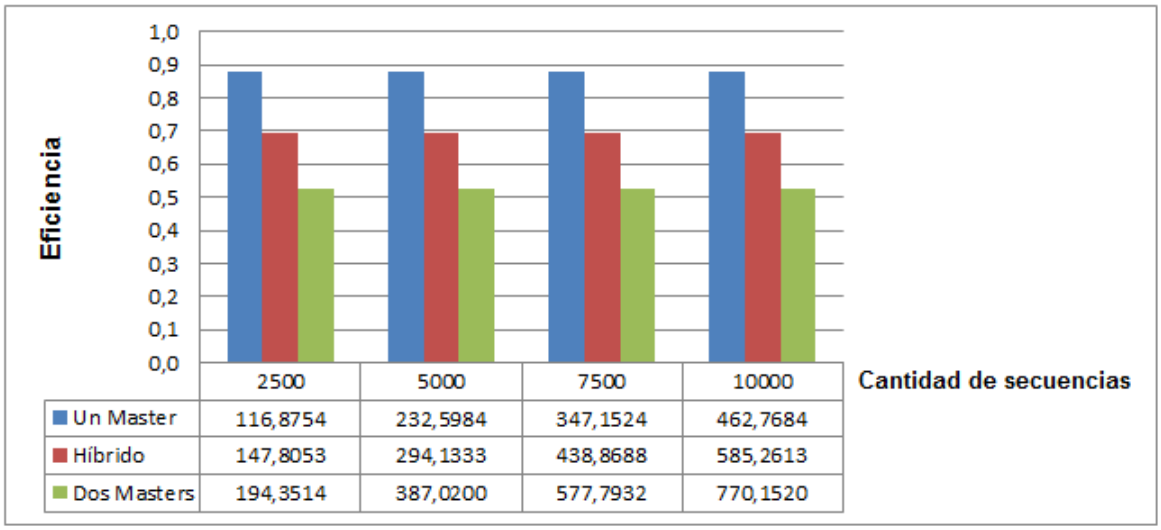
	2500	5000	7500	10000
Un Master	248,2340	495,5512	743,0568	990,2700
Híbrido	314,3303	627,6215	940,9663	1254,1750
Dos Masters	413,2132	825,1550	1237,7060	1649,3980

Figura B.57. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



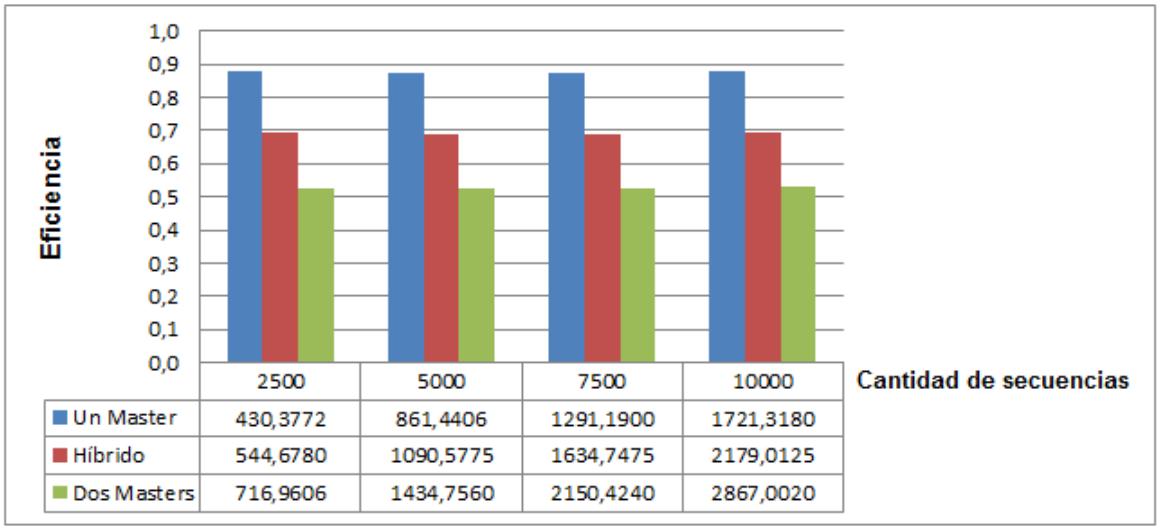
	2500	5000	7500	10000
Un Master	193,6588	387,0172	579,7830	772,9984
Híbrido	244,9095	489,6498	733,7540	978,3240
Dos Masters	322,2302	644,2032	965,2352	1287,0800

Figura B.58. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



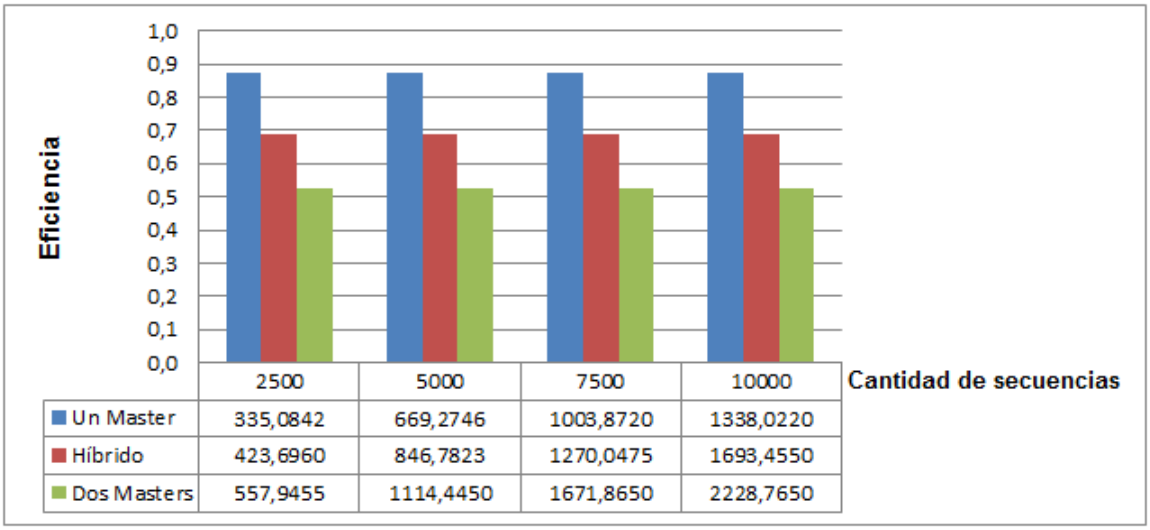
	2500	5000	7500	10000
Un Master	116,8754	232,5984	347,1524	462,7684
Híbrido	147,8053	294,1333	438,8688	585,2613
Dos Masters	194,3514	387,0200	577,7932	770,1520

Figura B.59. El gráfico representa la eficiencia usando secuencias de longitud 7500 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



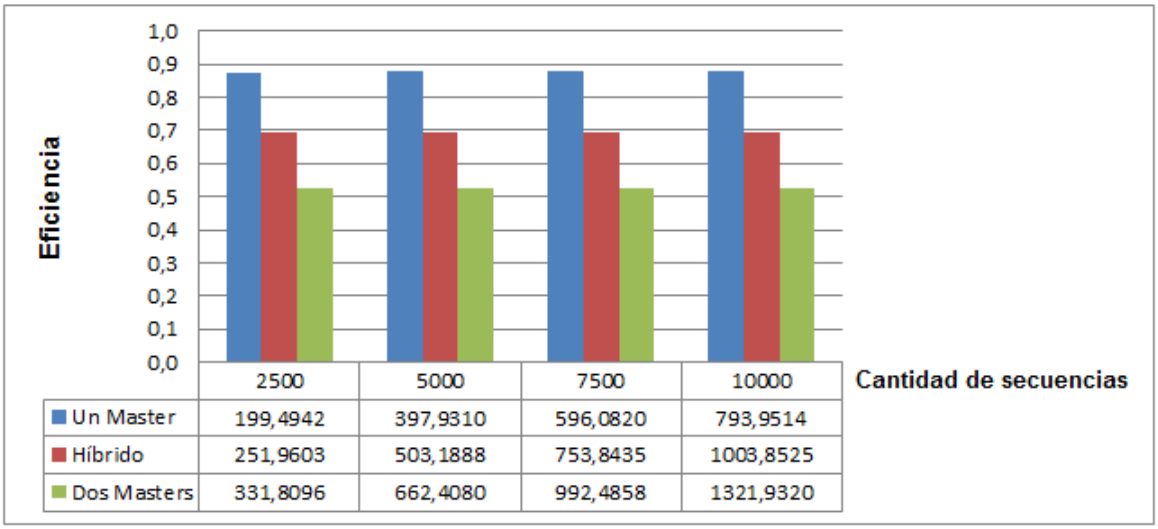
	2500	5000	7500	10000
Un Master	430,3772	861,4406	1291,1900	1721,3180
Híbrido	544,6780	1090,5775	1634,7475	2179,0125
Dos Masters	716,9606	1434,7560	2150,4240	2867,0020

Figura B.60. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



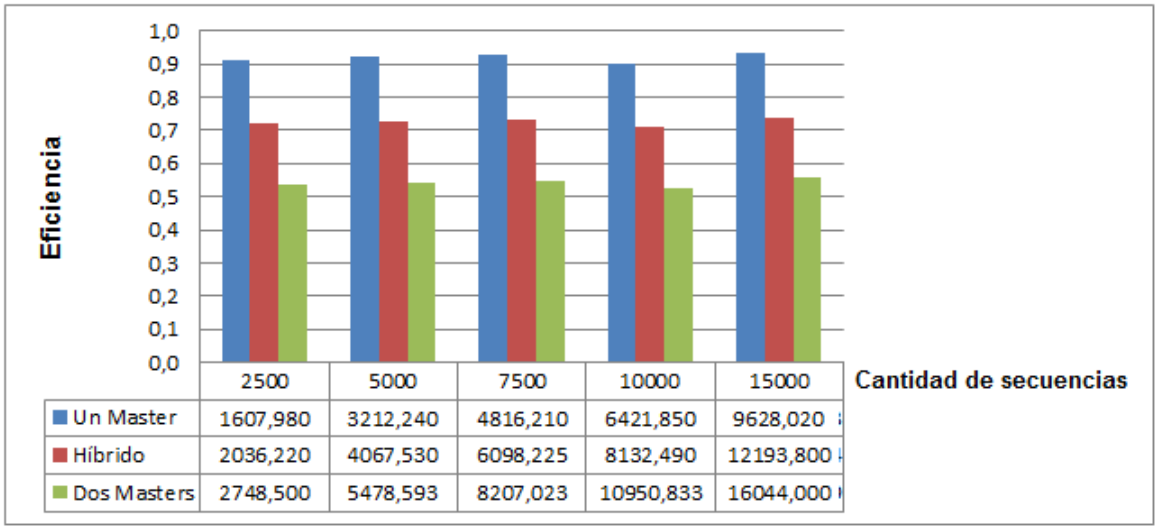
	2500	5000	7500	10000
Un Master	335,0842	669,2746	1003,8720	1338,0220
Híbrido	423,6960	846,7823	1270,0475	1693,4550
Dos Masters	557,9455	1114,4450	1671,8650	2228,7650

Figura B.61. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



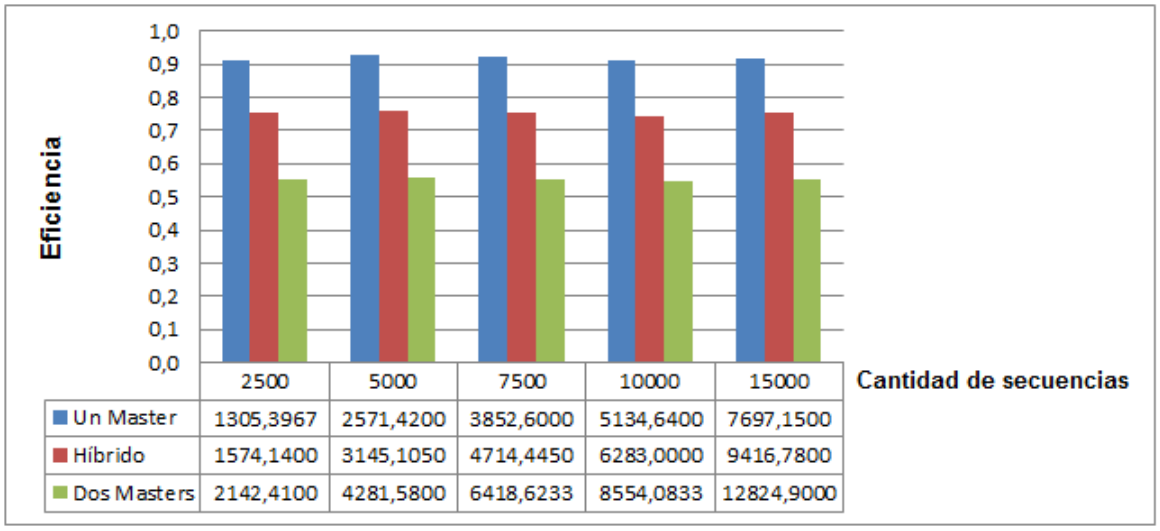
	2500	5000	7500	10000
Un Master	199,4942	397,9310	596,0820	793,9514
Híbrido	251,9603	503,1888	753,8435	1003,8525
Dos Masters	331,8096	662,4080	992,4858	1321,9320

Figura B.62. El gráfico representa la eficiencia usando secuencias de longitud 10000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.



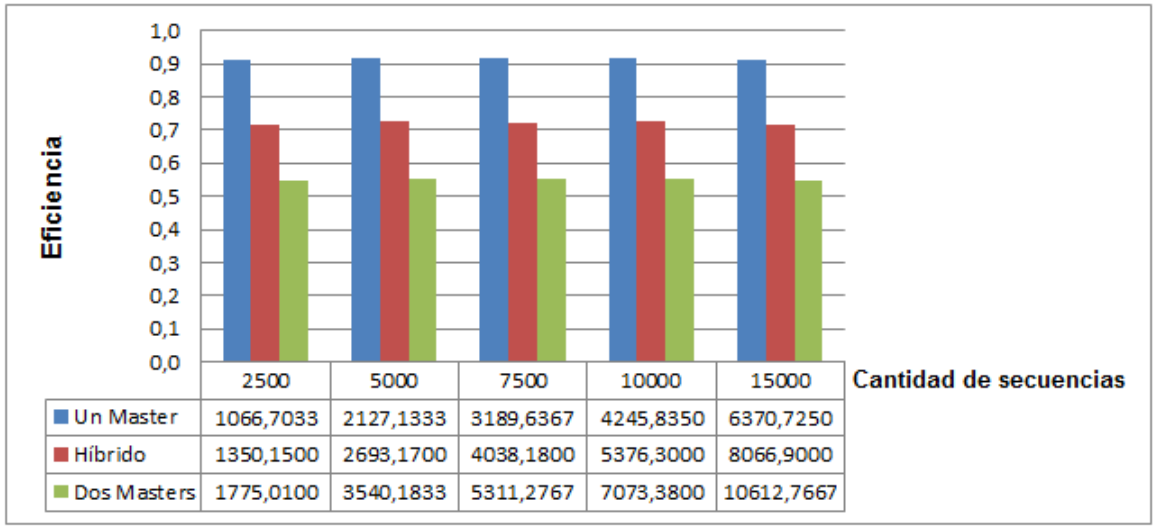
	2500	5000	7500	10000	15000
Un Master	1607,980	3212,240	4816,210	6421,850	9628,020
Híbrido	2036,220	4067,530	6098,225	8132,490	12193,800
Dos Masters	2748,500	5478,593	8207,023	10950,833	16044,000

Figura B.63. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 10%, para las distintas cantidades de secuencias.



	2500	5000	7500	10000	15000
Un Master	1305,3967	2571,4200	3852,6000	5134,6400	7697,1500
Híbrido	1574,1400	3145,1050	4714,4450	6283,0000	9416,7800
Dos Masters	2142,4100	4281,5800	6418,6233	8554,0833	12824,9000

Figura B.64. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 25%, para las distintas cantidades de secuencias.



	2500	5000	7500	10000	15000
Un Master	1066,7033	2127,1333	3189,6367	4245,8350	6370,7250
Híbrido	1350,1500	2693,1700	4038,1800	5376,3000	8066,9000
Dos Masters	1775,0100	3540,1833	5311,2767	7073,3800	10612,7667

Figura B.65. El gráfico representa la eficiencia usando secuencias de longitud 20000 con un porcentaje de variabilidad del 50%, para las distintas cantidades de secuencias.

Apéndice C

Códigos del problema "N-reinas"

Algoritmo Secuencial

```
#include <stdio.h>
#include <mpi.h>
#define  MAXSIZE  24
#define  MINSIZE   2

int  SIZE, SIZEE;
int  BOARD[MAXSIZE], *BOARDE, *BOARD1, *BOARD2;
int  MASK, TOPBIT, SIDEMASK, LASTMASK, ENDBIT;
int  BOUND1, BOUND2;

long double COUNT8, COUNT4, COUNT2;
long double  TOTAL, UNIQUE;

// Determina la cantidad de soluciones  por rotacion y simetria //
void Check(void) {
    int  *own, *you, bit, ptn;

    // 90-grados de rotación //
    if (*BOARD2 == 1) {
        for (ptn=2,own=BOARD+1; own<=BOARDE; own++,ptn<<=1) {
            bit = 1;
            for (you=BOARDE; *you!=ptn && *own>=bit; you--) bit <<= 1;
            if (*own > bit) return;
            if (*own < bit) break;
        }
        if (own > BOARDE) {
            COUNT2++;
            return;
        }
    }
    // 180-grados de rotación //
    if (*BOARDE == ENDBIT) {
        for (you=BOARDE-1,own=BOARD+1; own<=BOARDE; own++,you--) {
            bit = 1;
            for (ptn=TOPBIT; ptn!=*you && *own>=bit; ptn>>=1) bit <<= 1;
            if (*own > bit) return;
            if (*own < bit) break;
        }
        if (own > BOARDE) {
            COUNT4++;
            return;
        }
    }

    // 270-grados de rotación //
    if (*BOARD1 == TOPBIT) {
        for (ptn=TOPBIT>>1,own=BOARD+1; own<=BOARDE; own++,ptn>>=1) {
            bit = 1;
            for (you=BOARD; *you!=ptn && *own>=bit; you++) bit <<= 1;
            if (*own > bit) return;
            if (*own < bit) break;
        }
    }
```

```
    COUNT8++;
}

// Primer reina interna   //
void Backtrack2(int y, int left, int down, int right) {
    int  bitmap, bit;
    bitmap = MASK & ~(left | down | right);
    if (y == SIZEE) {
          if (bitmap) {
            if (!(bitmap & LASTMASK)) {
            BOARD[y] = bitmap;
                Check();
            }
        }
    } else {
        if (y < BOUND1) {
            bitmap |= SIDEMASK;
            bitmap ^= SIDEMASK;
        } else if (y == BOUND2) {
            if (!(down & SIDEMASK)) return;
            if ((down & SIDEMASK) != SIDEMASK) bitmap &= SIDEMASK;
        }
        while (bitmap) {
            bitmap ^= BOARD[y] = bit = -bitmap & bitmap;
            Backtrack2(y+1, (left | bit)<<1, down | bit, (right | bit)>>1);
        }
    }
}

// Primer reina en la esquina   //
void Backtrack1(int y, int left, int down, int right) {
    int  bitmap, bit;
    bitmap = MASK & ~(left | down | right);
    if (y == SIZEE) {
        if (bitmap) {
            BOARD[y] = bitmap;
            COUNT8++;
        }
    } else {
        if (y < BOUND1) {
            bitmap |= 2;
            bitmap ^= 2;
        }
       while (bitmap) {
            bitmap ^= BOARD[y] = bit = -bitmap & bitmap;
            Backtrack1(y+1, (left | bit)<<1, down | bit, (right | bit)>>1);
        }
    }
}

void NQueens(void) {
    int  bit, cant;
    COUNT8 = COUNT4 = COUNT2 = 0;
    SIZEE  = SIZE - 1;
    BOARDE = &BOARD[SIZEE];
    TOPBIT = 1 << SIZEE;
    MASK   = (1 << SIZE) - 1;
    BOARD[0] = 1;
    for (BOUND1=2; BOUND1<SIZEE; BOUND1++) {
        BOARD[1] = bit = 1 << BOUND1;
        Backtrack1(2, (2 | bit)<<1, 1 | bit, bit>>1);
    }
    SIDEMASK = LASTMASK = TOPBIT | 1;
```

```
    ENDBIT = TOPBIT >> 1;
    for (BOUND1=1,BOUND2=SIZE-2; BOUND1<BOUND2; BOUND1++,BOUND2--) {
        BOARD1 = &BOARD[BOUND1];
        BOARD2 = &BOARD[BOUND2];
        BOARD[0] = bit = 1 << BOUND1;
        Backtrack2(1, bit<<1, bit, bit>>1);
        LASTMASK |= LASTMASK>>1 | LASTMASK<<1;
        ENDBIT >>= 1;
    }
    UNIQUE = COUNT8     + COUNT4     + COUNT2;
    TOTAL = COUNT8 * 8 + COUNT4 * 4 + COUNT2 * 2;
}

//////////////////////////////////////////////////////////////////////
// N-Reinas Secuencial                                              //
// Parámetros:                                                      //
//            1- Tamaño del tablero                                 //
//////////////////////////////////////////////////////////////////////
int main(int argC, char *argV[]) {
    double tIni, tFin;

    SIZE=atoi(argV[1]);
    tIni= MPI_Wtime();
    NQueens();
    tFin= MPI_Wtime();

    printf("Tiempo Total: %f\nCantidad de resultados:%Lf\n", tFin-tIni,TOTAL);
    return 0;
}
```

Algoritmo Paralelo – Modelo Uno

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int  SIZE, SIZEE, CFIJA;
int  *BOARD, *BOARDE, *BOARD1, *BOARD2;
int  MASK, TOPBIT, SIDEMASK, LASTMASK, ENDBIT;
int  BOUND1, BOUND2;

long double  COUNT8, COUNT4, COUNT2;
long double  TOTAL;
double *TIEMPO;


// Determina la cantidad de soluciones  por rotacion y simetria //
void Check(void) {
    int  *own, *you, bit, ptn;

    // 90-grados de rotación //
    if (*BOARD2 == 1) {
        for (ptn=2,own=BOARD+1; own<=BOARDE; own++,ptn<<=1) {
            bit = 1;
            for (you=BOARDE; *you!=ptn && *own>=bit; you--) bit <<= 1;
            if (*own > bit) return;
            if (*own < bit) break;
        }
        if (own > BOARDE) {
            COUNT2++;
            return;
        }
    }
```

```
    // 180-grados de rotación //
    if (*BOARDE == ENDBIT) {
        for (you=BOARDE-1,own=BOARD+1; own<=BOARDE; own++,you--) {
            bit = 1;
            for (ptn=TOPBIT; ptn!=*you && *own>=bit; ptn>>=1) bit <<= 1;
            if (*own > bit) return;
            if (*own < bit) break;
        }
        if (own > BOARDE) {
            COUNT4++;
            return;
        }
    }
    // 270-grados de rotación //
    if (*BOARD1 == TOPBIT) {
        for (ptn=TOPBIT>>1,own=BOARD+1; own<=BOARDE; own++,ptn>>=1) {
            bit = 1;
            for (you=BOARD; *you!=ptn && *own>=bit; you++) bit <<= 1;
            if (*own > bit) return;
            if (*own < bit) break;
        }
    }
    COUNT8++;
}

// Primer reina interna                          //
void Backtrack2(int y, int left, int down, int right) {
    int  bitmap, bit;

    bitmap = MASK & ~(left | down | right);
    if (y == SIZEE) {
        if (bitmap) {
            if (!(bitmap & LASTMASK)) {
                BOARD[y] = bitmap;
                Check();
            }
        }
    } else {
        if (y < BOUND1) {
            bitmap |= SIDEMASK;
            bitmap ^= SIDEMASK;
        } else if (y == BOUND2) {
            if (!(down & SIDEMASK)) return;
            if ((down & SIDEMASK) != SIDEMASK) bitmap &= SIDEMASK;
        }
        while (bitmap) {
            bitmap ^= BOARD[y] = bit = -bitmap & bitmap;
            Backtrack2(y+1, (left | bit)<<1, down | bit, (right | bit)>>1);
        }
    }
}

void Backtrack2Fijo(int y, int left, int down, int right, int *otros)  {
    int  bitmap, bit;

    bitmap = MASK & ~(left | down | right);
    if (y < BOUND1) {
        bitmap |= SIDEMASK;
        bitmap ^= SIDEMASK;
    }
    else
       if (y == BOUND2) {
     if (!(down & SIDEMASK)) return;
```

```
            if ((down & SIDEMASK) != SIDEMASK) bitmap &= SIDEMASK;
         };
      BOARD[y] = bit = 1 << otros[y];
      if (bitmap&bit)
         if (y < (CFIJA-1)) {
              Backtrack2Fijo(y+1,(left|bit)<<1, down|bit, (right|bit)>>1, otros);
         }
         else Backtrack2(y+1,(left|bit)<<1, down|bit, (right|bit)>>1);
};

// Primer reina en la esquina                    //
void Backtrack1(int y, int left, int down, int right) {
    int  bitmap, bit;

    bitmap = MASK & ~(left | down | right);
    if (y == SIZEE) {
        if (bitmap) {
            BOARD[y] = bitmap;
            COUNT8++;
        }
    } else {
        if (y < BOUND1) {
            bitmap |= 2;
            bitmap ^= 2;
        }
       while (bitmap) {
            bitmap ^= BOARD[y] = bit = -bitmap & bitmap;
            Backtrack1(y+1, (left | bit)<<1, down | bit, (right | bit)>>1);
        }
    }
}

void Backtrack1Fijo(int y, int left, int down, int right, int *otros) {
    int  bitmap, bit;

    bitmap = MASK & ~(left | down | right);
    if (y < BOUND1) {
        bitmap |= 2;
        bitmap ^= 2;
    };
    BOARD[y] = bit = 1 << otros[y];
    if (bitmap&bit)
        if (y < (CFIJA-1)) {
            Backtrack1Fijo(y+1,(left|bit)<<1, down|bit, (right|bit)>>1, otros);
        }
        else Backtrack1(y+1,(left|bit)<<1, down|bit, (right|bit)>>1);
};

// Procesar una combinacion                      //
void ProcesoNodo(int com) {
    int *valores,i, bit;

    //Determina las posiciones fijas para cada fila
    valores= (int*) malloc (sizeof(int)*CFIJA);

    for (i=1; i<CFIJA; i++) {
  valores[i]= com%SIZE;
        com= com/SIZE;
    };
    valores[0]=com;

    if (valores[0]==0) { //Si la Primer reina está en la esquina
        if ((valores[1]>1)&&(valores[1]<SIZEE)) {
```

```
            BOARD[0]= 1;
            BOARD[1]= bit = 1 << valores[1];
          BOUND1= valores[1];
            Backtrack1Fijo(2, (2 | bit)<<1, 1 | bit, bit>>1, valores);
  };
    } else {  //Si la Primer reina está en el interior
        LASTMASK = SIDEMASK;
        for (i=1; i<valores[0]; i++) LASTMASK |= LASTMASK>>1 | LASTMASK<<1;
        BOUND1=valores[0];
        ENDBIT = TOPBIT >> BOUND1;
        BOUND2=SIZEE-valores[0];
        BOARD1 = &BOARD[BOUND1];
        BOARD2 = &BOARD[BOUND2];
        BOARD[0] = bit = 1 << BOUND1;
        Backtrack2Fijo(1, bit<<1, bit, bit>>1, valores);
      };
    free(valores);
};

////////////////////////////////////////////////////////////////////
//Parametros:                                                     //
// 1- Tamano del tablero                                          //
// 2- Cantidad de filas para formar las combinaciones             //
// 3- % de combinaciones que se reparten inicialmente             //
// 4- Cantidad de combinaciones inicial que se dan en cada pedido //
////////////////////////////////////////////////////////////////////

// Programa principal (MAIN)                                      //
int main(int argc, char *argv[])
{   int i,cantComb,act,tag;
    int idTarea,cantT,*mensaje, porciento, cantInicial, tamB, cantidades;
    double tII,*tiempos, tFF, tIni, tFin, tAux;
    long double cant;
    MPI_Status origen;
    MPI_Request req;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &idTarea);
    MPI_Comm_size(MPI_COMM_WORLD, &cantT);

    mensaje=(int*) malloc (sizeof(int)*2);

    SIZE=atoi(argv[1]);
    CFIJA=atoi(argv[2]);
    COUNT8 = COUNT4 = COUNT2=0;
    BOARD=(int *) malloc(sizeof(int)*SIZE);
    SIZEE  = SIZE - 1;
    BOARDE = &BOARD[SIZEE];
    MASK   = (1 << SIZE) - 1;
    TOPBIT = 1 << SIZEE;
    SIDEMASK = TOPBIT | 1;
    tag=1;

    tII=MPI_Wtime();
    if (idTarea==0) { //El master reparte la parte inciial y luego el resto
          cantT--;
          cantComb= SIZE/2;
          for (i=1; i<CFIJA; i++) cantComb*=SIZE;
          porciento=atoi(argv[3]);
          tamB=atoi(argv[4]);
          cantInicial= (cantComb*porciento/100);

          //Calcula la cantidad inicial para cada worker de forma equitativa
          cantidades = cantInicial/cantT;
```

```
        //Envia la parte inicial a cada uno
        act=0;
        for (i=1; i<=cantT; i++) {
            mensaje[0]=act;
              mensaje[1]=act+cantidades;
              MPI_Send(mensaje,2,MPI_INT,i,tag, MPI_COMM_WORLD);
              act=mensaje[1];
       };


        while (act<cantComb) {
              //recibe pedido
              MPI_Recv(mensaje,1, MPI_INT, MPI_ANY_SOURCE, tag,
              MPI_COMM_WORLD,&origen);
              //procesa pedido
              mensaje[0]=act;
              mensaje[1]=act+tamB;
              if (mensaje[1]>cantComb) mensaje[1]=cantComb;
              //envía pedido
               MPI_Send(mensaje,2,MPI_INT,origen.MPI_SOURCE,tag,
               MPI_COMM_WORLD);
               act+=tamB;
                  if (tamB > 1) tamB--;
         };

        //Se envia la senal de terminacion
        for (i=1; i<=cantT; i++) {
                  MPI_Recv(mensaje,1, MPI_INT, MPI_ANY_SOURCE, tag,
                  MPI_COMM_WORLD,&origen);
                  mensaje[0]=-1;
                  MPI_Send(mensaje,2,MPI_INT,origen.MPI_SOURCE,tag,
                  MPI_COMM_WORLD);
        };

        cant=0;
        TOTAL=0;
    } else { // Los worker reciben y piden trabajo

        //recibe carga de trabajo inicial
        MPI_Recv(mensaje,2, MPI_INT, 0, tag, MPI_COMM_WORLD,&origen);
        while (mensaje[0]>-1) {
             //procesa trabajo
            for (i=mensaje[0]; i< mensaje[1]; i++) ProcesoNodo(i);

                  //realiza un nuevo pedido
                  MPI_Send(mensaje,1,MPI_INT,0,tag, MPI_COMM_WORLD);
                  MPI_Recv(mensaje,2, MPI_INT,0,tag, MPI_COMM_WORLD,&origen);
        };

        cant = COUNT8 *8 + COUNT4 *4 + COUNT2 *2;

   };

   //Recibe e imprime los tiempos de los otrso procesos
   MPI_Reduce(&cant,&TOTAL, 1, MPI_LONG_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
   tFF=MPI_Wtime();

   if (idTarea==0) {
          printf("\n Tamanio del tablero: %d - Cantidad combinaciones: %d -
          Cantidad de resultados: %Lf -\n\n Tiempo Total: %f \n",SIZE,
          cantComb, TOTAL,tFF-tII);
```

```
     };
    MPI_Finalize();
    return 0;
};
```

Algoritmo Paralelo – Modelo Dos

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int  SIZE, SIZEE, CFIJA;
int  *BOARD, *BOARDE, *BOARD1, *BOARD2;
int  MASK, TOPBIT, SIDEMASK, LASTMASK, ENDBIT;
int  BOUND1, BOUND2;

long double  COUNT8, COUNT4, COUNT2;
long double  TOTAL;
double *TIEMPO;

// Determina la cantidad de soluciones  por rotacion y simetria //
void Check(void) {
    int  *own, *you, bit, ptn;

    // 90-grados de rotación //
    if (*BOARD2 == 1) {
        for (ptn=2,own=BOARD+1; own<=BOARDE; own++,ptn<<=1) {
            bit = 1;
            for (you=BOARDE; *you!=ptn && *own>=bit; you--) bit <<= 1;
            if (*own > bit) return;
            if (*own < bit) break;
        }
        if (own > BOARDE) {
            COUNT2++;
            return;
        }
    }
    // 180-grados de rotación //
    if (*BOARDE == ENDBIT) {
        for (you=BOARDE-1,own=BOARD+1; own<=BOARDE; own++,you--) {
            bit = 1;
            for (ptn=TOPBIT; ptn!=*you && *own>=bit; ptn>>=1) bit <<= 1;
            if (*own > bit) return;
            if (*own < bit) break;
        }
        if (own > BOARDE) {
            COUNT4++;
            return;
        }
    }
    // 270-grados de rotación //
    if (*BOARD1 == TOPBIT) {
        for (ptn=TOPBIT>>1,own=BOARD+1; own<=BOARDE; own++,ptn>>=1) {
            bit = 1;
            for (you=BOARD; *you!=ptn && *own>=bit; you++) bit <<= 1;
            if (*own > bit) return;
            if (*own < bit) break;
        }
    }
    COUNT8++;
}

// Primer reina interna                          //
void Backtrack2(int y, int left, int down, int right) {
```

```
    int  bitmap, bit;

    bitmap = MASK & ~(left | down | right);
    if (y == SIZEE) {
        if (bitmap) {
            if (!(bitmap & LASTMASK)) {
                BOARD[y] = bitmap;
                Check();
            }
        }
    } else {
        if (y < BOUND1) {
            bitmap |= SIDEMASK;
            bitmap ^= SIDEMASK;
        } else if (y == BOUND2) {
            if (!(down & SIDEMASK)) return;
            if ((down & SIDEMASK) != SIDEMASK) bitmap &= SIDEMASK;
        }
        while (bitmap) {
            bitmap ^= BOARD[y] = bit = -bitmap & bitmap;
            Backtrack2(y+1, (left | bit)<<1, down | bit, (right | bit)>>1);
        }
    }
}

void Backtrack2Fijo(int y, int left, int down, int right, int *otros)  {
    int  bitmap, bit;

    bitmap = MASK & ~(left | down | right);
    if (y < BOUND1) {
        bitmap |= SIDEMASK;
        bitmap ^= SIDEMASK;
    }
    else
       if (y == BOUND2) {
     if (!(down & SIDEMASK)) return;
           if ((down & SIDEMASK) != SIDEMASK) bitmap &= SIDEMASK;
       };
    BOARD[y] = bit = 1 << otros[y];
    if (bitmap&bit)
       if (y < (CFIJA-1)) {
            Backtrack2Fijo(y+1,(left|bit)<<1, down|bit, (right|bit)>>1, otros);
       }
       else Backtrack2(y+1,(left|bit)<<1, down|bit, (right|bit)>>1);
};

// Primer reina en la esquina                    //
void Backtrack1(int y, int left, int down, int right) {
    int  bitmap, bit;

    bitmap = MASK & ~(left | down | right);
    if (y == SIZEE) {
        if (bitmap) {
            BOARD[y] = bitmap;
            COUNT8++;
        }
    } else {
        if (y < BOUND1) {
            bitmap |= 2;
            bitmap ^= 2;
        }
       while (bitmap) {
            bitmap ^= BOARD[y] = bit = -bitmap & bitmap;
```

```
            Backtrack1(y+1, (left | bit)<<1, down | bit, (right | bit)>>1);
        }
    }
}

void Backtrack1Fijo(int y, int left, int down, int right, int *otros) {
    int  bitmap, bit;

    bitmap = MASK & ~(left | down | right);
    if (y < BOUND1) {
        bitmap |= 2;
        bitmap ^= 2;
    };
    BOARD[y] = bit = 1 << otros[y];
    if (bitmap&bit)
        if (y < (CFIJA-1)) {
            Backtrack1Fijo(y+1,(left|bit)<<1, down|bit, (right|bit)>>1, otros);
        }
        else Backtrack1(y+1,(left|bit)<<1, down|bit, (right|bit)>>1);
};

// Procesar una combinacion                    //
void ProcesoNodo(int com, int id) {
    int *valores,i, bit;

    //Determina las posiciones fijas para cada fila
    valores= (int*) malloc (sizeof(int)*CFIJA);

    for (i=1; i<CFIJA; i++) {
  valores[i]= com%SIZE;
        com= com/SIZE;
    };
    valores[0]=com;
    if (valores[0]==0) { //Si la Primer reina está en la esquina
        if ((valores[1]>1)&&(valores[1]<SIZEE)) {
             BOARD[0]= 1;
             BOARD[1]= bit = 1 << valores[1];
           BOUND1= valores[1];
             Backtrack1Fijo(2, (2 | bit)<<1, 1 | bit, bit>>1, valores);
  };
    } else {  //Si la Primer reina está en el interior
        LASTMASK = SIDEMASK;
        for (i=1; i<valores[0]; i++) LASTMASK |= LASTMASK>>1 | LASTMASK<<1;
        BOUND1=valores[0];
        ENDBIT = TOPBIT >> BOUND1;
        BOUND2=SIZEE-valores[0];
        BOARD1 = &BOARD[BOUND1];
        BOARD2 = &BOARD[BOUND2];
        BOARD[0] = bit = 1 << BOUND1;
        Backtrack2Fijo(1, bit<<1, bit, bit>>1, valores);
      };
    free(valores);
};

////////////////////////////////////////////////////////////////////
// Parámetros:                                                    //
// 1- Tamaño del tablero                                          //
// 2- Cantidad de filas para formar las combinaciones             //
// 3- % de combinaciones que se reparten inicialmente             //
// 4- Cantidad de combinaciones inicial que se dan en cada pedido //
//      5- Cantidad de procesadores por máquina                   //
////////////////////////////////////////////////////////////////////
```

```
// Programa principal (MAIN)                                          //
int main(int argc, char *argv[])
{    int i,cantComb,act,tag, cantProc;
     int idTarea,cantT,*mensaje, porciento, cantInicial, tamB, cantidades;
     double tII,*tiempos, tFF, tIni, tFin, tAux;
     long double cant;
     MPI_Status origen;
     MPI_Request req;
     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &idTarea);
     MPI_Comm_size(MPI_COMM_WORLD, &cantT);

     mensaje=(int*) malloc (sizeof(int)*2);
     cantProc=atoi(argv[5]);
     SIZE=atoi(argv[1]);
     CFIJA=atoi(argv[2]);
     COUNT8 = COUNT4 = COUNT2=0;
     BOARD=(int *) malloc(sizeof(int)*SIZE);
     SIZEE  = SIZE - 1;
     BOARDE = &BOARD[SIZEE];
     MASK   = (1 << SIZE) - 1;
     TOPBIT = 1 << SIZEE;
     SIDEMASK = TOPBIT | 1;
     tag=1;

     tII=MPI_Wtime();
     if (idTarea==0) { //El master reparte la parte inciial y luego el resto
           cantT--;
           cantComb= SIZE/2;
           for (i=1; i<CFIJA; i++) cantComb*=SIZE;
           porciento=atoi(argv[3]);
           tamB=atoi(argv[4]);

           //Calcula la cantidad inicial para cada worker de forma equitativa
           cantInicial= (cantComb*porciento/100);
         cantidades = cantInicial/cantProc;

           //Envia la parte inicial a cada uno
           act=0;
           int master2=cantProc;
           while( master2<=cantT) {
                     mensaje[0]=act;
                     mensaje[1]=act+cantidades;
                     MPI_Send(mensaje,2,MPI_INT,master2,tag,
                     MPI_COMM_WORLD);
                     act=mensaje[1];
                     master2+=cantProc;
         };

         //Recibe pedidos y envia trabajo
           while (act<cantComb) {
                 //recibe pedido
                 MPI_Recv(mensaje,1, MPI_INT, MPI_ANY_SOURCE, tag,
                 MPI_COMM_WORLD,&origen);
                 //procesa pedido
                 mensaje[0]=act;
                 mensaje[1]=act+tamB;
                 if (mensaje[1]>cantComb) mensaje[1]=cantComb;
                 //envía pedido
                 MPI_Send(mensaje,2,MPI_INT,origen.MPI_SOURCE,tag,
                 MPI_COMM_WORLD);
                 act+=tamB;
```

```
            if (tamB > 1) tamB--;
      };

      //Se envia la senal de terminacion
      master2=cantProc;
      while (master2<=cantT){
            MPI_Recv(mensaje,1, MPI_INT, MPI_ANY_SOURCE, tag,
            MPI_COMM_WORLD,&origen);
            mensaje[0]=-1;
            MPI_Send(mensaje,2,MPI_INT,origen.MPI_SOURCE,tag,
            MPI_COMM_WORLD);
            master2=master2+cantProc;
     };

   cant=0;
   TOTAL=0;
} else {
   if(idTarea % cantProc ==0 ){//MASTER NIVEL DOS
         int comb;
         //RECIBE CARGA DE TRABAJO INICIAL
         MPI_Recv(mensaje,2, MPI_INT, 0, tag, MPI_COMM_WORLD,&origen);
         while (mensaje[0]>-1) {
               for (i=mensaje[0]; i< mensaje[1]; i++){
                     MPI_Recv(&comb,1,MPI_INT,MPI_ANY_SOURCE,tag,
                     MPI_COMM_WORLD,&origen);
                     comb=i;
                     MPI_Send(&comb,1,MPI_INT,origen.MPI_SOURCE,tag,MPI_CO
                     MM_WORLD);
               };
               //Cuando se queda sin trabajo pide màs al master1
               MPI_Send(mensaje,1,MPI_INT,0,tag, MPI_COMM_WORLD);
               MPI_Recv(mensaje,2, MPI_INT,0,tag, MPI_COMM_WORLD,
               &origen);
               };

                     //ENVÍA SEÑALIZACIÓN DE FIN A SUS WORKER
               int worker;
               for (worker=1; worker<=cantProc-1; worker++) {

                     MPI_Recv(&comb,1,MPI_INT,MPI_ANY_SOURCE,tag,MPI
                     _COMM_WORLD,&origen);
                     comb=-1;
                     MPI_Send(&comb,1,MPI_INT,origen.MPI_SOURCE,
                     tag,MPI_COMM_WORLD);
                     };
                     cant = COUNT8 *8 + COUNT4 *4 + COUNT2 *2;
         }
         else{//WORKER
                     int comb;
                     int MASTER2=(idTarea /cantProc +1 )*cantProc;

                     //hace un pedido incial de trabajo a su master
                     MPI_Send(&comb,1,MPI_INT, MASTER2,tag,
                     MPI_COMM_WORLD);
                     MPI_Recv(&comb,1,MPI_INT,MASTER2, tag,
                     MPI_COMM_WORLD,&origen);

                     while (comb>-1) {
                           ProcesoNodo(comb, idTarea);

                           //realiza un nuevo pedido
                           MPI_Send(&comb,1,MPI_INT, MASTER2,tag,
                           MPI_COMM_WORLD);
```

```
                                MPI_Recv(&comb,1,MPI_INT, MASTER2, tag,
                                MPI_COMM_WORLD,&origen);
                            };
                            cant = COUNT8 *8 + COUNT4 *4 + COUNT2 *2;
                    };
            };

        //Recibe e imprime los tiempos de los otrso procesos
        MPI_Reduce(&cant,&TOTAL, 1, MPI_LONG_DOUBLE, MPI_SUM, 0,
    MPI_COMM_WORLD);
        tFF=MPI_Wtime();


        if (idTarea==0) {
            printf("\n Tamanio del tablero: %d - Cantidad combinaciones: %d -
            Cantidad de resultados: %Lf -\n\n Tiempo Total: %f \n",SIZE, cantComb,
            TOTAL,tFF-tII);
        };
       MPI_Finalize();
       return 0;
};
```

Algoritmo Paralelo – Modelo Tres

```
////////////////////////////////////////////////////////////////////////////
// Parametros:                                                            //
//       1- Tamano del tablero                                            //
//       2- Cantidad de filas para formar las combinaciones               //
//       3- % de combinaciones que se reparten inicialmente               //
//       4- Cantidad de combinaciones inicial que se dan en cada pedido   //
//            5- Cantidad de threads por máquina                          //
////////////////////////////////////////////////////////////////////////////
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <pthread.h>

int  SIZE, SIZEE, CFIJA;
int  MASK, TOPBIT, SIDEMASK;
long double  TOTAL;
double *TIEMPO;

typedef struct argumentos {
       int* soy_primero;
       pthread_mutex_t* sem_soy_primero;
       pthread_mutex_t* sem_siguiente;
       pthread_mutex_t* sem_tiempos;
       MPI_Status* origen;
       int* mensaje;
       int *sig;
       int *fin;
       double* tIni;
       double* tFin;
       double* tAux;
       int *id;
       long double *total;
} Argumento;



// Determina la cantidad de soluciones  por rotacion y simetria //
```

```
void Check(long double* COUNT8_b, long double* COUNT4_b, long double* COUNT2_b,
int *BOARD, int *BOARDE, int *BOARD1, int *BOARD2, int *ENDBIT) {
	int  *own, *you, bit, ptn;
	// 90-grados de rotación //
	if (*BOARD2 == 1) {
		for (ptn=2,own=BOARD+1; own<=BOARDE; own++,ptn<<=1) {
			bit = 1;
			for (you=BOARDE; *you!=ptn && *own>=bit; you--) bit <<= 1;
			if (*own > bit) return;
			if (*own < bit) break;
		}
		if (own > BOARDE) {
			(*COUNT2_b)++;
			return;
		}
	}
	// 180-grados de rotación //
	if (*BOARDE == (*ENDBIT)) {
		for (you=BOARDE-1,own=BOARD+1; own<=BOARDE; own++,you--) {
			bit = 1;
			for (ptn=TOPBIT; ptn!=*you && *own>=bit; ptn>>=1)
				bit <<= 1;
			if (*own > bit) return;
			if (*own < bit) break;
		}
		if (own > BOARDE) {
			(*COUNT4_b)++;
			return;
		}
	}
	// 270-grados de rotación //
	if (*BOARD1 == TOPBIT) {
		for (ptn=TOPBIT>>1,own=BOARD+1; own<=BOARDE; own++,ptn>>=1) {
			bit = 1;
			for (you=BOARD; *you!=ptn && *own>=bit; you++) bit <<= 1;
			if (*own > bit) return;
			if (*own < bit) break;
		}
	}
	(*COUNT8_b)++;
}

// Primer reina interna                          //
void Backtrack2(int y, int left, int down, int right, long double *COUNT8, long
double *COUNT4, long double* COUNT2, int *BOARD, int *BOARDE, int *BOARD1, int
*BOARD2, int *LASTMASK, int *ENDBIT, int *BOUND1,int *BOUND2) {
	int  bitmap, bit;

	bitmap = MASK & ~(left | down | right);
	if (y == SIZEE) {
		if (bitmap) {
			if (!(bitmap & (*LASTMASK))) {

				BOARD[y] = bitmap;
				Check(COUNT8, COUNT4, COUNT2, BOARD, BOARDE,
				BOARD1, BOARD2, ENDBIT);
			}
		}
	} else {
		if (y < (*BOUND1)) {
			bitmap |= SIDEMASK;
			bitmap ^= SIDEMASK;
		} else if (y == (*BOUND2)) {
```

```
                        if (!(down & SIDEMASK)) return;
                        if ((down & SIDEMASK) != SIDEMASK) bitmap &= SIDEMASK;
                }
                while (bitmap) {
                        bitmap ^= BOARD[y] = bit = -bitmap & bitmap;
                        Backtrack2(y+1, (left | bit)<<1, down | bit, (right |
                       bit)>>1, COUNT8, COUNT4, COUNT2, BOARD, BOARDE, BOARD1,
                       BOARD2, LASTMASK, ENDBIT, BOUND1, BOUND2);
                }
        }
}

void Backtrack2Fijo(int y, int left, int down, int right, int *otros, long
double *COUNT8, long double *COUNT4, long double* COUNT2, int *BOARD, int
*BOARDE, int *BOARD1, int *BOARD2, int *LASTMASK, int *ENDBIT, int *BOUND1,int
*BOUND2)  {
        int  bitmap, bit;

        bitmap = MASK & ~(left | down | right);
        if (y < (*BOUND1)) {
                bitmap |= SIDEMASK;
                bitmap ^= SIDEMASK;
        }
        else if (y == (*BOUND2)) {
                if (!(down & SIDEMASK)) return;
                if ((down & SIDEMASK) != SIDEMASK) bitmap &= SIDEMASK;
        };
        BOARD[y] = bit = 1 << otros[y];
        if (bitmap&bit)
                if (y < (CFIJA-1)) {
                        Backtrack2Fijo(y+1,(left|bit)<<1, down|bit,
                       (right|bit)>>1, otros, COUNT8,COUNT4, COUNT2, BOARD,
                       BOARDE, BOARD1, BOARD2, LASTMASK,ENDBIT, BOUND1, BOUND2);
                }
                else Backtrack2(y+1,(left|bit)<<1, down|bit, (right|bit)>>1,
                       COUNT8, COUNT4, COUNT2, BOARD, BOARDE, BOARD1, BOARD2,
                       LASTMASK, ENDBIT, BOUND1, BOUND2);
};

// Primer reina en la esquina                   //
void Backtrack1(int y, int left, int down, int right,long double* COUNT8_b, int
*BOARD, int *BOARDE, int *BOARD1, int *BOARD2, int *BOUND1,int *BOUND2) {
        int  bitmap, bit;

        bitmap = MASK & ~(left | down | right);
        if (y == SIZEE) {
                if (bitmap) {
                        BOARD[y] = bitmap;
                        (*COUNT8_b)++;
                }
        } else {
                if (y < (*BOUND1)) {
                        bitmap |= 2;
                        bitmap ^= 2;
                }
                while (bitmap) {
                        bitmap ^= BOARD[y] = bit = -bitmap & bitmap;
                        Backtrack1(y+1, (left | bit)<<1, down | bit, (right |
                       bit)>>1, COUNT8_b, BOARD, BOARDE, BOARD1, BOARD2, BOUND1,
                       BOUND2);
                }
        }
}
```

```
void Backtrack1Fijo(int y, int left, int down, int right, int *otros, long
double* COUNT8, int *BOARD, int *BOARDE, int *BOARD1, int *BOARD2, int
*BOUND1,int *BOUND2) {
	int  bitmap, bit;

	bitmap = MASK & ~(left | down | right);
	if (y < (*BOUND1)) {
		bitmap |= 2;
		bitmap ^= 2;
	};
	BOARD[y] = bit = 1 << otros[y];
	if (bitmap&bit)
		if (y < (CFIJA-1)) {
		  Backtrack1Fijo(y+1,(left|bit)<<1, down|bit, (right|bit)>>1,
		  otros, COUNT8, BOARD, BOARDE, BOARD1, BOARD2, BOUND1, BOUND2);
		}
		else Backtrack1(y+1,(left|bit)<<1, down|bit, (right|bit)>>1,
		COUNT8, BOARD, BOARDE, BOARD1, BOARD2, BOUND1, BOUND2);
};

// Procesar una combinacion                    //
void ProcesoNodo(int com, long double  *COUNT8, long double *COUNT4, long
double* COUNT2, int *BOARD, int *BOARDE, int *BOARD1, int *BOARD2, int
*LASTMASK, int *ENDBIT, int *BOUND1,int *BOUND2) {
	int *valores,i, bit;

	//Determina las posiciones fijas para cada fila
	valores= (int*) malloc (sizeof(int)*CFIJA);

	for (i=1; i<CFIJA; i++) {
		valores[i]= com%SIZE;
		com= com/SIZE;
	};
	valores[0]=com;

	if (valores[0]==0) { //Si la Primer reina esta en la esquina
		if ((valores[1]>1)&&(valores[1]<SIZEE)) {
			BOARD[0]= 1;
			BOARD[1]= bit = 1 << valores[1];
			(*BOUND1)= valores[1];
			Backtrack1Fijo(2, (2 | bit)<<1, 1 | bit, bit>>1, valores,
			COUNT8, BOARD, BOARDE, BOARD1, BOARD2, BOUND1, BOUND2);
		};
	} else {  //Si la Primer reina esta en el interior
		(*LASTMASK) = SIDEMASK;

		for (i=1; i<valores[0]; i++) (*LASTMASK) |= (*LASTMASK)>>1 |
		(*LASTMASK)<<1;
		(*BOUND1)=valores[0];
		(*ENDBIT) = TOPBIT >> (*BOUND1);
		(*BOUND2)=SIZEE-valores[0];
		BOARD1 = &BOARD[(*BOUND1)];
		BOARD2 = &BOARD[(*BOUND2)];
		BOARD[0] = bit = 1 << (*BOUND1);
		 Backtrack2Fijo(1, bit<<1, bit, bit>>1, valores, COUNT8, COUNT4,
		COUNT2,BOARD, BOARDE, BOARD1, BOARD2, LASTMASK, ENDBIT, BOUND1,
		BOUND2);
	};
	free(valores);
};

void * master_worker(void * arg) {
```

```
        Argumento variables;
        int nodo;
        long double  COUNT8, COUNT4, COUNT2, total_parcial, id;
        int  *BOARD, *BOARDE, *BOARD1, *BOARD2, LASTMASK, ENDBIT;
        int  BOUND1, BOUND2;
        int i;
        BOARD=(int *) malloc(sizeof(int)*SIZE);
        SIZEE  = SIZE - 1;
        BOARDE = &BOARD[SIZEE];
        pthread_t self;
        variables = *(Argumento *) arg;
        COUNT2=COUNT4=COUNT8=total_parcial=0;

        self=pthread_self();
        cpu_set_t cpuset;
        CPU_ZERO(&cpuset);

        pthread_mutex_lock(variables.sem_soy_primero);
        id=(*variables.id);
        (*variables.id)++;
        if (*variables.soy_primero) {
               *variables.soy_primero=0;
               //recibo el trabajo inicial
               MPI_Recv(variables.mensaje,2, MPI_INT, 0, 1,
               MPI_COMM_WORLD,variables.origen);
               pthread_mutex_unlock(variables.sem_siguiente);
        }
        pthread_mutex_unlock(variables.sem_soy_primero);

        CPU_SET(id, &cpuset);
        pthread_setaffinity_np(self,sizeof(cpu_set_t), &cpuset);

        pthread_mutex_lock(variables.sem_siguiente);
        while(variables.mensaje[0]>-1) {
               if(variables.mensaje[0]<variables.mensaje[1]) {
                      //se guardam los datos necesarios
                      nodo=variables.mensaje[0];
                      //se linera para que siga otro
                      variables.mensaje[0]=variables.mensaje[0]+1;
                      pthread_mutex_unlock(variables.sem_siguiente);
                      ProcesoNodo(nodo, &COUNT8, &COUNT4, &COUNT2, BOARD,
                      BOARDE, BOARD1, BOARD2, &LASTMASK, &ENDBIT, &BOUND1,
                     &BOUND2);
               } else {
                      MPI_Send(variables.mensaje,1,MPI_INT,0,1, MPI_COMM_WORLD);
                      MPI_Recv(variables.mensaje,2, MPI_INT,0,1,
                      MPI_COMM_WORLD,variables.origen);
                      pthread_mutex_unlock(variables.sem_siguiente);
               }
               pthread_mutex_lock(variables.sem_siguiente);
        }
        pthread_mutex_unlock(variables.sem_siguiente);

        total_parcial=  COUNT8 *8 + COUNT4 *4 + COUNT2 *2;

        pthread_mutex_lock(variables.sem_siguiente);
        (*variables.total)+=total_parcial;
        pthread_mutex_unlock(variables.sem_siguiente);
}


// Programa principal (MAIN)                                              //
```

```
int main(int argc, char *argv[])
{
        int i,cantComb,act,tag;
        int idTarea,cantT,*mensaje, porciento, cantInicial, tamB, cantidades;
        double tII,*tiempos, tFF, tIni, tFin, tAux;
        int cant_threads_x_maq=atoi(argv[5]);
        long double cant=0;
        MPI_Status origen;
        MPI_Request req;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &idTarea);
        MPI_Comm_size(MPI_COMM_WORLD, &cantT);

        mensaje=(int*) malloc (sizeof(int)*2);
        SIZE=atoi(argv[1]);
        CFIJA=atoi(argv[2]);
        SIZEE  = SIZE - 1;
        MASK   = (1 << SIZE) - 1;
        TOPBIT = 1 << SIZEE;
        SIDEMASK = TOPBIT | 1;
        tag=1;

        tII=MPI_Wtime();
        if (idTarea==0) { //El master reparte la parte inciial y luego el resto
                cantT--;
                cantComb= SIZE/2;
                for (i=1; i<CFIJA; i++) cantComb*=SIZE;
                porciento=atoi(argv[3]);
                tamB=atoi(argv[4]);

                //Calcula la cantidad inicial para cada worker de forma
                //equitativa
                cantInicial= (cantComb*porciento/100);
                cantidades = cantInicial/cantT;

                //Envia la parte inicial a cada uno
                act=0;
                for (i=1; i<=cantT; i++) {
                        mensaje[0]=act;
                        mensaje[1]=act+cantidades;
                        MPI_Send(mensaje,2,MPI_INT,i,tag, MPI_COMM_WORLD);
                        act=mensaje[1];
                };

                //Recibe pedidos y envia trabajo
                while (act<cantComb) {
                        //recibe pedido
                        MPI_Recv(mensaje,1, MPI_INT, MPI_ANY_SOURCE, tag,
                        MPI_COMM_WORLD,&origen);
                        //procesa pedido
                        mensaje[0]=act;
                        mensaje[1]=act+tamB;
                        if (mensaje[1]>cantComb) mensaje[1]=cantComb;
                        //envía pedido
                        MPI_Send(mensaje,2,MPI_INT,origen.MPI_SOURCE,tag,
                       MPI_COMM_WORLD);
                        act+=tamB;
                        if (tamB > 1) tamB--;
                };
                //Se envia la senal de terminacion
                for (i=1; i<=cantT; i++) {
                        MPI_Recv(mensaje,1, MPI_INT, MPI_ANY_SOURCE, tag,
            MPI_COMM_WORLD,&origen);
```

```
                        mensaje[0]=-1;
                        MPI_Send(mensaje,2,MPI_INT,origen.MPI_SOURCE,tag,
                MPI_COMM_WORLD);
                  };
                  cant=0;
                  TOTAL=0;
          } else { // Los worker reciben y piden trabajo

                  pthread_t thread[cant_threads_x_maq];
                  int i, soy_pri, sig, fin;
                  long double total;
                  total=0;
                  Argumento arg;
                  pthread_mutex_t soy_primero;
                  pthread_mutex_t empezar;
                  pthread_mutex_t siguiente;
                  pthread_mutex_t tiempos;
                  pthread_mutex_init(&soy_primero, NULL);
                  arg.sem_soy_primero=&soy_primero;
                  arg.total=&total;
                  soy_pri=1;
                  arg.soy_primero=&soy_pri;
                  pthread_mutex_init(&siguiente, NULL);
                  pthread_mutex_lock(&siguiente);
                  arg.sem_siguiente=&siguiente;
                  int id_id=0;
                  arg.id=&id_id;
                  arg.mensaje=mensaje;
                  arg.origen=&origen;
                  arg.sig=&sig;
                  arg.fin=&fin;

                  for( i=0; i<cant_threads_x_maq; i++) {
                        pthread_create(&thread[i], NULL, master_worker,&arg);
                  }

                  for( i=0; i<cant_threads_x_maq; i++) {
                        pthread_join(thread[i], NULL);
                  }
                  cant=total;
          };

          //Recibe e imprime los tiempos de los otros procesos
          MPI_Reduce(&cant,&TOTAL, 1, MPI_LONG_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD);
          tFF=MPI_Wtime();

          if (idTarea==0) {
                  printf("\n Tamanio del tablero: %d - Cantidad combinaciones: %d -
                  Cantidad de resultados: %Lf -\n\n Tiempo Total: %f \n",SIZE,
                  cantComb, TOTAL,tFF-tII);
          };
          MPI_Finalize();
          return 0;
};
```

Apéndice D

Códigos del problema "Búsqueda de similitud máxima en secuencias de ADN"

Algoritmo Secuencial

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

#define Valor(a,b) ((a==b)?1:(-0.3))
#define Maximo(a,b,c) (a>=b && a>=c && a>=0)?a:((b>=c && b>=0)?b:((c>=0)?c:0))

float CalcularMaximo(char *secV, int filas, char *secH, int columnas, float iG,
float eG){
  int i,j;

  //INICIALIZA TODOS LOS VECTORES QUE REPRESENTARAN LA MATRIZ DE SIMILITUD
  float *fila, *filaMax, *columnaMax, anterior;
  fila = (float *) malloc (columnas*sizeof(float));
  filaMax = (float *) malloc (columnas*sizeof(float));
  columnaMax = (float *) malloc (filas*sizeof(float));
  for (i=0; i<columnas; i++) {
        fila[i] = 0;
        filaMax[i] = 0;
  }

  for (i=0; i<filas; i++) columnaMax[i] = 0;
  anterior = 0;

  //COMIENZA EL CALCULO DE LA MATRIZ
  float max=0, aux, actual, v1, v2, v3;
  for (i=1; i<filas; i++){
        anterior = 0;
        for (j=1; j<columnas; j++){
              v1 = fila[j-1]+Valor(secV[i-1],secH[j-1]);
              v2 = filaMax[j] - eG;
              v3 = columnaMax[i] - eG;
              actual = Maximo(v1,v2,v3);
              if ((actual-iG) > (filaMax[j]-eG)) filaMax[j] = actual -iG;
              else filaMax[j] = filaMax[j]-eG;
              if ((actual-iG) > (columnaMax[i]-eG)) columnaMax[i] = actual-iG;
              else columnaMax[i] = columnaMax[i]-eG;
              fila[j-1] = anterior;
              anterior=actual;
              if (actual > max) max = actual;
        };
  };
  free(fila);
  free(filaMax);
  free(columnaMax);
  return max;
}
```

```
////////////////////////////////////////////
// PARAMETROS:                            //
//              1 - Test                  //
//              2 - Base                  //
//              3 - Cantidad Secuencias   //
//              4 - Tamaño de secuencia máximo //
////////////////////////////////////////////

int main (int *argc, char *argv[]) {
   int tamMaximo = atoi(argv[4])+1, tamBase = atoi(argv[3]);
   double tiempoIni, tiempoFin;
   FILE *archivo;

   //CARGA LA SECUENCIA TEST
   char *secTest;
   int tamTest;
   secTest = (char *) malloc (tamMaximo * sizeof(char));
   archivo = fopen(argv[1], "r");
   fscanf(archivo, "%s", secTest);
   tamTest = strlen(secTest)+1;
   fclose(archivo);

   tiempoIni = MPI_Wtime();
   //LEE Y PROCESA CADA SECUENCIA DE LA BD
   char *secPrueba;
   int tamPrueba, i, secMaxima = 0;
   float similitudMaxima = 0, similitud, penalidadInicioGAP=1,
   penalidadExtensionGAP=0.3;
   secPrueba = (char *) malloc (tamMaximo * sizeof(char));
   archivo = fopen(argv[2], "r");
   for (i=0; i<tamBase; i++){
         fscanf(archivo, "%s", secPrueba);
         tamPrueba = strlen(secPrueba)+1;
         similitud = CalcularMaximo(secTest, tamTest, secPrueba, tamPrueba,
         penalidadInicioGAP,
         penalidadExtensionGAP);
         if (similitud > similitudMaxima) {
               similitudMaxima = similitud;
               secMaxima = i;
         };
   };
   fclose(archivo);
   tiempoFin = MPI_Wtime();

   //INFORMA RESULTADO
   printf("TAM: %d - CANT: %d - Secuencia: %d - Similitud: %f - Tiempo: %g \n",
   tamMaximo, tamBase,
   secMaxima, similitudMaxima, tiempoFin-tiempoIni);
   free(secTest);
   free(secPrueba);

   return 1;
};
```

Algoritmo Paralelo – Modelo Uno

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>

#define Valor(a,b) ((a==b)?1:(-0.3))
#define Maximo(a,b,c) (a>=b && a>=c && a>=0)?a:((b>=c && b>=0)?b:((c>=0)?c:0))
```

```
float CalcularMaximo(char *secV, int filas, char *secH, int columnas, float iG,
float eG) {
    int i,j;

    //INICIALIZA TODOS LOS VECTORES QUE REPRESENTARAN LA MATRIZ DE
  SIMILITUD
    float *fila, *filaMax, *columnaMax, anterior;
    fila = (float *) malloc (columnas*sizeof(float));
    filaMax = (float *) malloc (columnas*sizeof(float));
    columnaMax = (float *) malloc (filas*sizeof(float));
    for (i=0; i<columnas; i++) {
        fila[i] = 0;
        filaMax[i] = 0;
    }
    for (i=0; i<filas; i++) columnaMax[i] = 0;
    anterior = 0;

    //COMIENZA EL CALCULO DE LA MATRIZ
    float max=0, aux, actual, v1, v2, v3;
    for (i=1; i<filas; i++) {
        anterior = 0;
        for (j=1; j<columnas; j++) {
            v1 = fila[j-1]+Valor(secV[i-1],secH[j-1]);
            v2 = filaMax[j] - eG;
            v3 = columnaMax[i] - eG;
            actual = Maximo(v1,v2,v3);
            if ((actual-iG) > (filaMax[j]-eG)) filaMax[j] = actual -iG;
            else filaMax[j] = filaMax[j]-eG;
            if ((actual-iG) > (columnaMax[i]-eG)) columnaMax[i] = actual -iG;
            else columnaMax[i] = columnaMax[i]-eG;
            fila[j-1] = anterior;
            anterior=actual;
            if (actual > max) max = actual;
        };
    };
    free(fila);
    free(filaMax);
    free(columnaMax);
    return max;
}

////////////////////////////////////////////////////////////////////////
// PARAMETROS:                                                        //
//            1 - test (secuencia a buscar)                           //
//       2 - base  (bbdd)                                             //
//       3 - Cantidad Secuencias                                      //
//       4 - Tamanio de secuencia maximo                              //
//       5 - Porcentaje de secuencias que se reparten inicialmente    //
//       6- Cantidad de secuencias que se envían en el primer pedido  //
////////////////////////////////////////////////////////////////////////
int main (int argc, char *argv[]) {
  int idTarea,sec,tamPrueba,tag=1,cantT,tamMaximo = atoi(argv[4])+1, tamBase =
  atoi(argv[3]),cantInicial,cantidades,porcentaje=atoi(argv[5]),cantSec=atoi(a
  rgv[6]), tamMensaje;
   double tiempoIni, tiempoFin, *tiempos,*TIEMPO, tII, tAux, tIni,tFin;
   char *secTest = 0, *secPrueba = 0, *secAux = 0;
   int secMaxima = 0;
  float    similitudMaxima    =    0,    similitud,    penalidadInicioGAP=1,
  penalidadExtensionGAP=0.3, *SECUENCIA, *TOTAL;

    int tamTest;
    MPI_Status origen;
```

```
MPI_Request req;

//INICIALIZA MPI
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &idTarea);
MPI_Comm_size(MPI_COMM_WORLD, &cantT);

//CALCULA LA CANTIDAD INICAL DE TRABAJO PARA CADA WORKER Y EN
//BASE A ESO EL TAMAÑO MÁXIMO PARA EL BUFFER secPrueba
cantInicial= (tamBase*porcentaje/100);
cantidades = cantInicial/(cantT-1);

if(cantidades>cantSec) {
    tamMensaje=tamMaximo*cantidades+cantidades;
} else {
    tamMensaje=tamMaximo*cantSec+cantSec;
}

//ALOCACIONES
secPrueba = (char *) malloc (tamMensaje* sizeof(char));
secTest = (char *) malloc ( (tamMaximo) * sizeof(char));
secAux = (char *) malloc ((tamMaximo) * sizeof(char));
SECUENCIA=(float *) malloc (sizeof(float)*2);

tiempoIni = MPI_Wtime();
//MASTER
if(idTarea==0) {

    cantT--;
    FILE *archivo;

    TOTAL=(float *) malloc (sizeof(float)*(cantT+1)*2);

    //CARGA LA SECUENCIA TEST
    archivo = fopen(argv[1], "r");
    fscanf(archivo, "%s", secTest);
    tamTest = strlen(secTest)+1;
    fclose(archivo);
    //se emvía la secuencia test a todos los worker
    MPI_Bcast(secTest,tamMaximo,MPI_CHAR,0, MPI_COMM_WORLD);

    int worker, sent=0;
    archivo = fopen(argv[2], "r");

    //REPARTE EL PORCENTAJE INICIAL ENTRE TODOS LOS WORKERS
    for(worker=1; worker<=cantT; worker++) {

        sec=sent;
        fscanf(archivo, "%s", secPrueba);
        strcat(secPrueba,"\n");
        sent++;
        int j;
        //para mandar varias secuencias en un mismo mensaje se
       //concatenan con el token de separación \n
        for(j=1; j<cantidades; j++) {
            fscanf(archivo, "%s", secAux);
            strcat(secAux,"\n");
            sent++;
            strcat(secPrueba,secAux);
        }

        MPI_Send(secPrueba,tamMensaje,MPI_CHAR,worker,sec, MPI_COMM_WORLD);
    }
```

```
    //RECIBE PEDIDOS Y ENVÍA SECUENCIAS
    while( sent<tamBase) {

        //recibe pedido
        MPI_Recv(secPrueba,tamMensaje, MPI_CHAR, MPI_ANY_SOURCE, tag,
         MPI_COMM_WORLD,&origen);

        //procesa pedido
        sec=sent;
        fscanf(archivo, "%s", secPrueba);
        strcat(secPrueba,"\n");
        sent++;
        int j=1;
        while((j<cantSec)&&(sent<tamBase)) {
            fscanf(archivo, "%s", secAux);
            strcat(secAux,"\n");
            strcat(secPrueba,secAux);
            j++;
            sent++;
        }
        if(cantSec>1)cantSec--;

        //envía pedido
        MPI_Send(secPrueba,tamMensaje,MPI_CHAR,origen.MPI_SOURCE,sec,
        MPI_COMM_WORLD);

    };

    fclose(archivo);

    //ENVÍA SEÑAL DE FINALIZACIÓN A CADA WORKER
    for(worker=1; worker<=cantT; worker++) {
        MPI_Recv(secPrueba,tamMensaje, MPI_CHAR, MPI_ANY_SOURCE, tag,
         MPI_COMM_WORLD,&origen);
        strncpy(secPrueba,"FIN\n",4);
        MPI_Send(secPrueba,tamMensaje,MPI_CHAR,origen.MPI_SOURCE,tag,
        MPI_COMM_WORLD);
    }

    //WORKER
} else {

    //recibe secuencia test
    MPI_Bcast(secTest,tamMaximo,MPI_CHAR,0, MPI_COMM_WORLD);
    tamTest = strlen(secTest)+1;

    //recibe una carga de trabajo inicial
    MPI_Recv(secPrueba,tamMensaje,MPI_CHAR,0,MPI_ANY_TAG,
  MPI_COMM_WORLD,&origen);

    //PROCESA Y HACE PEDIDOS HASTA QUE LLEGUE LA SEÑAL DE FIN
    while(strncmp(secPrueba,"FIN",3)!=0) {

        sec=origen.MPI_TAG;

        //saca secuencias del mensaje para comparar con test
        secAux=strtok(secPrueba,"\n");

        while(secAux!=NULL) {
            //CALCULA SIMILITUD Y VE SI ES MÁXIMA
            tamPrueba = strlen(secAux)+1;
            similitud = CalcularMaximo(secTest, tamTest, secAux, tamPrueba,
            penalidadInicioGAP, penalidadExtensionGAP);
```

```
                if (similitud > similitudMaxima) {
                    similitudMaxima = similitud;
                    secMaxima = sec;
                };

                //saca otra secuencia del mensaje para comparar con test
                secAux=strtok(NULL,"\n");
                sec++;
            }

            //pide más trabajo
            MPI_Send(secPrueba,tamMensaje,MPI_CHAR,0,tag, MPI_COMM_WORLD);
            MPI_Recv(secPrueba,tamMensaje,MPI_CHAR,0,MPI_ANY_TAG,
        MPI_COMM_WORLD,&origen);
        }

        SECUENCIA[0]=similitudMaxima;
        SECUENCIA[1]=secMaxima;

    }
    //El master recolecta los resultados locales de cada worker y el tiempo de
  procesamiento de cada uno
  MPI_Gather(SECUENCIA,2, MPI_FLOAT, TOTAL, 2, MPI_FLOAT, 0, MPI_COMM_WORLD);

    tiempoFin = MPI_Wtime();

    if(idTarea==0) {
        int w;
        float maxima=-1;

        //CALCULA EL MÁXIMO TOTAL ENTRE EL MÁXIMO LOCAL DE CADA WORKER
        for(w=1; w<=cantT; w++) {
            if(maxima<TOTAL[2*w]) {
                maxima=TOTAL[2*w];
                sec=TOTAL[(2*w)+1];
            }
        }

        printf("TAM: %d - CANT: %d - Secuencia %d - Similitud: %f - \n\nTiempo
        Total: %g \n", tamMaximo-1, tamBase,sec,  maxima, tiempoFin-tiempoIni);
        free(TOTAL);
    }
    free(SECUENCIA);
    free(secTest);
    free(secAux);
    free(secPrueba);

    MPI_Finalize();
    return 1;
};
```

Algoritmo Paralelo – Modelo Dos

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#define Valor(a,b) ((a==b)?1:(-0.3))
#define Maximo(a,b,c) (a>=b && a>=c && a>=0)?a:((b>=c && b>=0)?b:((c>=0)?c:0))

float CalcularMaximo(char *secV, int filas, char *secH, int columnas, float iG,
float eG) {
    int i,j;
```

```
    //INICIALIZA TODOS LOS VECTORES QUE REPRESENTARAN LA MATRIZ DE SIMILITUD
    float *fila, *filaMax, *columnaMax, anterior;
    fila = (float *) malloc (columnas*sizeof(float));
    filaMax = (float *) malloc (columnas*sizeof(float));
    columnaMax = (float *) malloc (filas*sizeof(float));
    for (i=0; i<columnas; i++) {
        fila[i] = 0;
        filaMax[i] = 0;
    }
    for (i=0; i<filas; i++) columnaMax[i] = 0;
    anterior = 0;

    //COMIENZA EL CALCULO DE LA MATRIZ
    float max=0, aux, actual, v1, v2, v3;
    for (i=1; i<filas; i++) {
        anterior = 0;
        for (j=1; j<columnas; j++) {
            v1 = fila[j-1]+Valor(secV[i-1],secH[j-1]);
            v2 = filaMax[j] - eG;
            v3 = columnaMax[i] - eG;
            actual = Maximo(v1,v2,v3);
            if ((actual-iG) > (filaMax[j]-eG)) filaMax[j] = actual -iG;
            else filaMax[j] = filaMax[j]-eG;
            if ((actual-iG) > (columnaMax[i]-eG)) columnaMax[i] = actual -iG;
            else columnaMax[i] = columnaMax[i]-eG;
            fila[j-1] = anterior;
            anterior=actual;
            if (actual > max) max = actual;
        };
    };
    free(fila);
    free(filaMax);
    free(columnaMax);
    return max;
}
////////////////////////////////////////////////////////////////////////////
// PARAMETROS:                                                            //
//      1 - test (secuencia a buscar)                                     //
//      2 - base  (bbdd)                                                  //
//      3 - Cantidad Secuencias                                           //
//      4 - Tamanio de secuencia maximo                                   //
//      5 - Porcentaje de secuencias que se reparten inicialmente         //
//      6 - Cantidad de secuencias que se envían en el primer pedido      //
//      7 - Cantidad de procesadores por máquina                          //
////////////////////////////////////////////////////////////////////////////

int main (int argc, char *argv[]) {
  int idTarea,sec,tamPrueba,tag=1,cantT,tamMaximo = atoi(argv[4])+1, tamBase =
  atoi(argv[3]),cantInicial,cantidades,
  porcentaje=atoi(argv[5]),cantSec=atoi(argv[6]), tamMensaje,
  cantProc=atoi(argv[7]);
    double tiempoIni, tiempoFin, *tiempos,*TIEMPO, tII, tAux, tIni,tFin;
    char *secTest = 0, *secPrueba = 0, *secAux = 0;
    int secMaxima = 0;
    float similitudMaxima = 0, similitud, penalidadInicioGAP=1,
  penalidadExtensionGAP=0.3, *SECUENCIA, *TOTAL;

    int tamTest;
    MPI_Status origen;
    MPI_Request req;

    //INICIALIZA MPI
    MPI_Init(&argc,&argv);
```

```
  MPI_Comm_rank(MPI_COMM_WORLD, &idTarea);
  MPI_Comm_size(MPI_COMM_WORLD, &cantT);

//CALCULA LA CANTIDAD INICAL DE TRABAJO PARA CADA WORKER Y EN BASE A ESO EL
TAMAÑO MÁXIMO PARA EL BUFFER secPrueba
  cantInicial= (tamBase*porcentaje/100);
  cantidades = cantInicial/(cantT/cantProc);

  if(cantidades>cantSec) {
      tamMensaje=tamMaximo*cantidades+cantidades;
  } else {
      tamMensaje=tamMaximo*cantSec+cantSec;
  }

  //ALOCACIONES
  secPrueba = (char *) malloc (tamMensaje* sizeof(char));
  secTest = (char *) malloc ( (tamMaximo) * sizeof(char));
  secAux = (char *) malloc ((tamMaximo) * sizeof(char));
  SECUENCIA=(float *) malloc (sizeof(float)*2);


  tiempoIni = MPI_Wtime();
  //MASTER
  if(idTarea==0) {

      cantT--;
      FILE *archivo;

      TOTAL=(float *) malloc (sizeof(float)*(cantT+1)*2);

      //CARGA LA SECUENCIA TEST
      archivo = fopen(argv[1], "r");
      fscanf(archivo, "%s", secTest);
      tamTest = strlen(secTest)+1;
      fclose(archivo);
      // se envia la secuencia  a todos los procesos,
      MPI_Bcast(secTest,tamMaximo,MPI_CHAR,0, MPI_COMM_WORLD);

      int master2, sent=0;
      archivo = fopen(argv[2], "r");

      //REPARTE EL PORCENTAJE INICIAL ENTRE TODOS LOS WORKERS
      master2=cantProc;
      while(master2<=cantT) {

          sec=sent;
          fscanf(archivo, "%s", secPrueba);
          strcat(secPrueba,"\n");
          sent++;
          int j;
          for(j=1; j<cantidades; j++) {
              fscanf(archivo, "%s", secAux);
              strcat(secAux,"\n");
              sent++;
              strcat(secPrueba,secAux);
          }

     MPI_Send(secPrueba,tamMensaje,MPI_CHAR,master2,sec, MPI_COMM_WORLD);
      master2+=cantProc;
       }
```

```
        //RECIBE PEDIDOS Y ENVÍA SECUENCIAS
        while( sent<tamBase) {

            //recibe pedido
           MPI_Recv(secPrueba,tamMensaje, MPI_CHAR, MPI_ANY_SOURCE, tag,
        MPI_COMM_WORLD,&origen);

            //procesa pedido
            sec=sent;
            fscanf(archivo, "%s", secPrueba);
            strcat(secPrueba,"\n");
            sent++;
            int j=1;
            while((j<cantSec)&&(sent<tamBase)) {
                fscanf(archivo, "%s", secAux);
                strcat(secAux,"\n");
                strcat(secPrueba,secAux);
                j++;
                sent++;
            }
            if(cantSec>1)cantSec--;

            //envía pedido
            MPI_Send(secPrueba,tamMensaje,MPI_CHAR,origen.MPI_SOURCE,sec,
            MPI_COMM_WORLD);
        };

        fclose(archivo);

        //ENVÍA SEÑAL DE FINALIZACIÓN A CADA MASTER DE NIVEL 2
        master2=cantProc;
        while( master2<=cantT) {
            MPI_Recv(secPrueba,tamMensaje, MPI_CHAR, MPI_ANY_SOURCE, tag,
            MPI_COMM_WORLD,&origen);
            strncpy(secPrueba,"FIN\n",4);
            MPI_Send(secPrueba,tamMensaje,MPI_CHAR,origen.MPI_SOURCE,tag,
            MPI_COMM_WORLD);
            master2+=cantProc;
        }

    } else {
        //MASTER NIVEL 2
        if(idTarea%cantProc==0) {
            char *secTemp = (char *) malloc ( (tamMaximo) * sizeof(char));
            int worker;

            //recibe secuencia test
            MPI_Bcast(secTest,tamMaximo,MPI_CHAR,0, MPI_COMM_WORLD);
            tamTest = strlen(secTest)+1;

            //recibe una carga de trabajo inicial
            MPI_Recv(secPrueba,tamMensaje,MPI_CHAR,0,MPI_ANY_TAG,
            MPI_COMM_WORLD,&origen);


            //PROCESA Y HACE PEDIDOS HASTA QUE LLEGUE LA SEÑAL DE FIN
            while(strncmp(secPrueba,"FIN",3)!=0) {
                sec=origen.MPI_TAG;

                //saca secuencias del mensaje para enviar a los worker cuando
                //hacen un pedido
                secAux=strtok(secPrueba,"\n");
                while(secAux!=NULL) {
```

```
                MPI_Recv(secTemp,tamMaximo, MPI_CHAR, MPI_ANY_SOURCE, tag,
                MPI_COMM_WORLD,&origen);
                MPI_Send(secAux,tamMaximo,MPI_CHAR,origen.MPI_SOURCE,sec,
                MPI_COMM_WORLD);

                secAux=strtok(NULL,"\n");
                sec++;
            }

            //pide más trabajo
            MPI_Send(secPrueba,tamMensaje,MPI_CHAR,0,tag, MPI_COMM_WORLD);
            MPI_Recv(secPrueba,tamMensaje,MPI_CHAR,0,MPI_ANY_TAG,
            MPI_COMM_WORLD,&origen);
        }

        //envía señal de finalización a todos sus worker
        for(worker=1; worker<cantProc; worker++) {

            MPI_Recv(secTemp,tamMaximo, MPI_CHAR, MPI_ANY_SOURCE, tag,
            MPI_COMM_WORLD,&origen);
            strncpy(secTemp,"FIN\n",4);
            MPI_Send(secTemp,tamMaximo,MPI_CHAR,origen.MPI_SOURCE,tag,
            MPI_COMM_WORLD);

        }

        free(secTemp);

    } else {

        int MASTER2=(idTarea /cantProc +1 )*cantProc;

        //recibe secuencia test
        MPI_Bcast(secTest,tamMaximo,MPI_CHAR,0, MPI_COMM_WORLD);
        tamTest = strlen(secTest)+1;

        //le pide trabajo inicial su master de nivel 2
        MPI_Send(secAux,tamMaximo,MPI_CHAR,MASTER2,tag,
        MPI_COMM_WORLD);
        MPI_Recv(secAux,tamMaximo,MPI_CHAR,MASTER2,MPI_ANY_TAG,
        MPI_COMM_WORLD,&origen);

        //PROCESA Y HACE PEDIDOS HASTA QUE LLEGUE LA SEÑAL DE FIN
        while(strncmp(secAux,"FIN",3)!=0) {

            sec=origen.MPI_TAG;

            //saca secuencias del mensaje para comparar con test
            tamPrueba = strlen(secAux)+1;
            similitud = CalcularMaximo(secTest, tamTest, secAux, tamPrueba,
            penalidadInicioGAP, penalidadExtensionGAP);
            if (similitud > similitudMaxima) {
                similitudMaxima = similitud;
                secMaxima = sec;
            };

            //pide más trabajo
            MPI_Send(secAux,tamMaximo,MPI_CHAR,MASTER2,tag,MPI_COMM_WORLD);
            MPI_Recv(secAux,tamMaximo,MPI_CHAR,MASTER2,MPI_ANY_TAG,
            MPI_COMM_WORLD,&origen);
        }
```

```
                SECUENCIA[0]=similitudMaxima;
                SECUENCIA[1]=secMaxima;
            }
        }

        MPI_Gather(SECUENCIA,2, MPI_FLOAT, TOTAL, 2, MPI_FLOAT, 0,MPI_COMM_WORLD);

        tiempoFin = MPI_Wtime();

        if(idTarea==0) {
            int w;
            float maxima=-1;

            //CALCULA EL MÁXIMO TOTAL ENTRE EL MÁXIMO LOCAL DE CADA WORKER
            for(w=1; w<=cantT; w++) {
                if(w%cantProc!=0) {
                    if(maxima<TOTAL[2*w]) {
                        maxima=TOTAL[2*w];
                        sec=TOTAL[(2*w)+1];
                    }
                }
            }

            printf("TAM: %d - CANT: %d - Secuencia %d - Similitud: %f - \n\nTiempo
            Total: %g \n", tamMaximo-1, tamBase,sec,  maxima, tiempoFin-
            tiempoIni);

            free(TOTAL);
        }
        free(SECUENCIA);
        free(secTest);
        free(secAux);
        free(secPrueba);
        MPI_Finalize();
        return 1;
};
```

Algoritmo Paralelo – Modelo Tres

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <pthread.h>

#define Valor(a,b) ((a==b)?1:(-0.3))
#define Maximo(a,b,c) (a>=b && a>=c && a>=0)?a:((b>=c && b>=0)?b:((c>=0)?c:0))
float similitudMaxima = 0,  penalidadInicioGAP=1, penalidadExtensionGAP=0.3;
int secMaxima = 0;

typedef struct argumentos {

       int* soy_primero;
       pthread_mutex_t* sem_soy_primero;
       pthread_mutex_t* sem_siguiente;
       pthread_mutex_t* sem_calculos;
       MPI_Status* origen;
       char* secPrueba;
       char* secTest;
       char* secAux;
       int *id;
       int *sec;
       int *pri;
```

```
	int tamMaximo;
	int tamMensaje;
	double* tIni;
	double* tFin;
	double* tAux;
} Argumento;

float CalcularMaximo(char *secV, int filas, char *secH, int columnas, float iG,
float eG) {
	int i,j;

	//INICIALIZA TODOS LOS VECTORES QUE REPRESENTARAN LA MATRIZ DE SIMILITUD
	float *fila, *filaMax, *columnaMax, anterior;
	fila = (float *) malloc (columnas*sizeof(float));
	filaMax = (float *) malloc (columnas*sizeof(float));
	columnaMax = (float *) malloc (filas*sizeof(float));
	for (i=0; i<columnas; i++) {
		fila[i] = 0;
		filaMax[i] = 0;
	}
	for (i=0; i<filas; i++) columnaMax[i] = 0;
	anterior = 0;

	//COMIENZA EL CALCULO DE LA MATRIZ
	float max=0, aux, actual, v1, v2, v3;
	for (i=1; i<filas; i++) {
		anterior = 0;
		for (j=1; j<columnas; j++) {
			v1 = fila[j-1]+Valor(secV[i-1],secH[j-1]);
			v2 = filaMax[j] - eG;
			v3 = columnaMax[i] - eG;
			actual = Maximo(v1,v2,v3);
			if ((actual-iG) > (filaMax[j]-eG)) filaMax[j] =actual -iG;
			else filaMax[j] = filaMax[j]-eG;
			if ((actual-iG)>(columnaMax[i]-eG))columnaMax[i=actual-iG;
			else columnaMax[i] = columnaMax[i]-eG;
			fila[j-1] = anterior;
			anterior=actual;
			if (actual > max) max = actual;
		};
	};
	free(fila);
	free(filaMax);
	free(columnaMax);
	return max;
}

void* master_worker(void * arg) {

	Argumento variables;
	variables= *(Argumento*) arg;
	int id;
	char *secAux;
	int tamTest,tamPrueba;
	double tII, tFin, tAux=0;
	float simi, similitudMaximaLocal=-1;
	int sec, secMaximaLocal;
	char * cmp = (char *) malloc (((variables.tamMaximo)+2) * sizeof(char));
	cpu_set_t cpuset;
	pthread_t self;
	self=pthread_self();
	CPU_ZERO(&cpuset);
```

```
        pthread_mutex_lock(variables.sem_soy_primero);
        id=(*variables.id);
        (*variables.id)++;
        if (*variables.soy_primero) {
                *variables.soy_primero=0;
                //recibe secuencia test
                MPI_Bcast(variables.secTest,variables.tamMaximo,MPI_CHAR,0,
         MPI_COMM_WORLD);
                //recibe trabajo inicial
              MPI_Recv(variables.secPrueba,variables.tamMensaje,MPI_CHAR,0,
         MPI_ANY_TAG, MPI_COMM_WORLD,variables.origen);
                *variables.sec=(*variables.origen).MPI_TAG;
                //desbloquea el semaforo para que puedan seguir con la sig etapa
                pthread_mutex_unlock(variables.sem_siguiente);
        }
        pthread_mutex_unlock(variables.sem_soy_primero);
        CPU_SET(id, &cpuset);
        pthread_setaffinity_np(self,sizeof(cpu_set_t), &cpuset);
}
pthread_mutex_lock(variables.sem_siguiente);

tamTest = strlen(variables.secTest)+1;

while(strncmp(variables.secPrueba,"FIN",3)!=0) {
        //SACA UNA SECUENCIA DE LA CADENA DE SECUENCIAS
        if(*variables.pri) {
                variables.secAux=strtok(variables.secPrueba,"\n");
                *variables.pri=0;
        } else {
                variables.secAux=strtok(NULL,"\n");
        }

        if(variables.secAux!=NULL) {

                tamPrueba = strlen(variables.secAux)+1;
                strncpy(cmp,variables.secAux,tamPrueba);

                sec=*variables.sec;
                (*variables.sec)++;
                //después de guardarme los datos que necesito para procesar
                //libero el sem
                pthread_mutex_unlock(variables.sem_siguiente);

                //realizo el procesamiento
                simi = CalcularMaximo(variables.secTest, tamTest, cmp, tamPrueba,
          penalidadInicioGAP, penalidadExtensionGAP);
                if (simi > similitudMaximaLocal) {
                        similitudMaximaLocal = simi;
                        secMaximaLocal = sec;
                };

        } else {
                //si no había mas secuencias, realizo otro pedido
                MPI_Send(variables.secPrueba,variables.tamMensaje,MPI_CHAR,0,1,
                MPI_COMM_WORLD);

                MPI_Recv(variables.secPrueba,variables.tamMensaje,MPI_CHAR,0,MPI
                _ANY_TAG, MPI_COMM_WORLD,variables.origen);
                *variables.sec=(*variables.origen).MPI_TAG;
                *variables.pri=1;
                pthread_mutex_unlock(variables.sem_siguiente);
        }
        pthread_mutex_lock(variables.sem_siguiente);
```

```
}
pthread_mutex_unlock(variables.sem_siguiente);

pthread_mutex_lock(variables.sem_calculos);
if (similitudMaxima < similitudMaximaLocal) {
       similitudMaxima = similitudMaximaLocal;
       secMaxima = secMaximaLocal;
};
pthread_mutex_unlock(variables.sem_calculos);
free(cmp);
}

////////////////////////////////////////////////////////////////////////
// PARAMETROS:                                                        //
//       1 - test (secuencia a buscar)                                //
//       2 - base  (bbdd)                                             //
//       3 - Cantidad Secuencias                                      //
//       4 - Tamanio de secuencia maximo                              //
//       5 - Porcentaje de secuencias que se reparten inicialmente    //
//       6- Cantidad de secuencias que se envían en el primer pedido  //
//       7 - Cantidad de procesadores por máquina                     //
////////////////////////////////////////////////////////////////////////

int main (int argc, char *argv[]) {
       int idTarea,tamTest,sec,tamPrueba,tag=1,cantT,
cantProc=atoi(argv[7]),tamMaximo = atoi(argv[4])+1, tamBase =
atoi(argv[3]),cantInicial,cantidades,
porcentaje=atoi(argv[5]),cantSec=atoi(argv[6]);
       double tiempoIni, tiempoFin, *tiempos,*TIEMPO, tAux, tIni,tFin;
       char *secTest, *secPrueba, *secAux;
       float  *SECUENCIA, *TOTAL;
       int tamMensaje;
       MPI_Status origen;
       MPI_Request req;

       //INICIALIZA MPI
       MPI_Init(&argc,&argv);
       MPI_Comm_rank(MPI_COMM_WORLD, &idTarea);
       MPI_Comm_size(MPI_COMM_WORLD, &cantT);

       //CALCULA LA CANTIDAD INICAL DE TRABAJO PARA CADA WORKER Y EN
      //BASE A ESO EL TAMAÑO MÁXIMO PARA EL BUFFER secPrueba
       cantInicial= (tamBase*porcentaje/100);
       cantidades = cantInicial/(cantT-1);

       if(cantidades>cantSec) {
              tamMensaje=tamMaximo*cantidades+cantidades;
       } else {
              tamMensaje=tamMaximo*cantSec+cantSec;
       }

       //ALOCACIONES
       secPrueba = (char *) malloc (tamMensaje* sizeof(char));
       secTest = (char *) malloc (tamMaximo * sizeof(char));
       secAux = (char *) malloc ((tamMaximo+2) * sizeof(char));
       SECUENCIA=(float *) malloc (sizeof(float)*2);


       tiempoIni = MPI_Wtime();
       //MASTER
       if(idTarea==0) {
              cantT--;
              FILE *archivo;
```

```
TOTAL=(float *) malloc (sizeof(float)*(cantT+1)*2);

//CARGA LA SECUENCIA TEST
archivo = fopen(argv[1], "r");
fscanf(archivo, "%s", secTest);
tamTest = strlen(secTest)+1;
fclose(archivo);
MPI_Bcast(secTest,tamMaximo,MPI_CHAR,0, MPI_COMM_WORLD);
int worker, sent=0;
archivo = fopen(argv[2], "r");

//REPARTE EL PORCENTAJE INICIAL ENTRE TODOS LOS WORKERS
for(worker=1; worker<=cantT; worker++) {

        sec=sent;
        fscanf(archivo, "%s", secPrueba);
        strcat(secPrueba,"\n");
        sent++;
        int j;
        for(j=1; j<cantidades; j++) {
                fscanf(archivo, "%s", secAux);
                strcat(secAux,"\n");
                sent++;
                strcat(secPrueba,secAux);
        }

        MPI_Send(secPrueba,tamMensaje,MPI_CHAR,worker,sec,
        MPI_COMM_WORLD);
}

//RECIBE PEDIDOS Y ENVÍA SECUENCIAS
while( sent<tamBase) {

        //recibe pedido
        MPI_Recv(secPrueba,tamMensaje, MPI_CHAR, MPI_ANY_SOURCE,
       tag, MPI_COMM_WORLD,&origen);

        //procesa pedido
        sec=sent;
        fscanf(archivo, "%s", secPrueba);
        strcat(secPrueba,"\n");
        sent++;
        int j=1;
        while((j<cantSec)&&(sent<tamBase)) {
                fscanf(archivo, "%s", secAux);
                strcat(secAux,"\n");
                strcat(secPrueba,secAux);
                j++;
                sent++;
        }
        if(cantSec>1)cantSec--;

        //envía pedido

       MPI_Send(secPrueba,tamMensaje,MPI_CHAR,origen.MPI_SOURCE,
       sec, MPI_COMM_WORLD);

};

fclose(archivo);

//ENVÍA SEÑAL DE FINALIZACIÓN A CADA WORKER
for(worker=1; worker<=cantT; worker++) {
```

```
                MPI_Recv(secPrueba,tamMensaje, MPI_CHAR, MPI_ANY_SOURCE,
                tag,MPI_COMM_WORLD,&origen);
                strncpy(secPrueba,"FIN\n",4);

                MPI_Send(secPrueba,tamMensaje,MPI_CHAR,origen.MPI_SOURCE,
                tag,MPI_COMM_WORLD);
        }
        //WORKER
} else {
        int w,soy_pri, pri=1;
        Argumento arg;
        pthread_mutex_t soy_primero;
        pthread_mutex_t sem_siguiente;
        pthread_mutex_t calculos;
        pthread_t thread[cantProc];
        int thread_id=0;
        pthread_mutex_init(&soy_primero, NULL);
        arg.sem_soy_primero=&soy_primero;

        soy_pri=1;
        arg.soy_primero=&soy_pri;
        arg.id=0;
        arg.pri=&pri;
        arg.id=&thread_id;
        pthread_mutex_init(&sem_siguiente, NULL);
        pthread_mutex_lock(&sem_siguiente);
        arg.sem_siguiente=&sem_siguiente;

        pthread_mutex_init(&calculos, NULL);
        arg.sem_calculos=&calculos;

        arg.sec=&sec;
        arg.tamMaximo=tamMaximo;
        arg.tamMensaje=tamMensaje;
        arg.secPrueba=secPrueba;
        arg.secAux=secAux;
        arg.secTest=secTest;
        arg.origen=&origen;


        for( w=0; w<cantProc; w++) {
                pthread_create(&thread[w], NULL, master_worker,&arg);
        }

        for( w=0; w<cantProc; w++) {
                pthread_join(thread[w], NULL);
        }

        SECUENCIA[0]=similitudMaxima;
        SECUENCIA[1]=secMaxima;
}

MPI_Gather(SECUENCIA,2, MPI_FLOAT, TOTAL, 2, MPI_FLOAT, 0,
MPI_COMM_WORLD);

tiempoFin = MPI_Wtime();

if(idTarea==0) {
        int w;
        float maxima=-1;
```

```
		//CALCULA EL MÁXIMO TOTAL ENTRE EL MÁXIMO LOCAL DE CADA WORKER
			for(w=1; w<=cantT; w++) {
				if(maxima<TOTAL[2*w]) {
					maxima=TOTAL[2*w];
					sec=TOTAL[(2*w)+1];
				}
			}

		printf("TAM: %d - CANT: %d - Secuencia %d - Similitud: %f -\n\n Tiempo
		Total: %g \n", tamMaximo-1, tamBase,sec,  maxima, tiempoFin-tiempoIni);
			free(TOTAL);
		}

		free(SECUENCIA);
		free(secTest);
		free(secAux);
		free(secPrueba);
		MPI_Finalize();

		return 1;

};
```

ANEXO A

Phtreads

A continuación se muestran una colección de rutinas que es suficiente para los algoritmos desarrollados en esta tesina. Detalles adicionales pueden encontrarse en la siguiente referencia [Bra96].

El archivo <pthread.h> contiene las definiciones del tipo pthread_t entre otros y los prototipos de las funciones, por lo cual debe incluirse en el código para usar las funciones que la librería provee.

Manejo de hilos

- Para crear un hilo se utiliza la función `pthread_create` cuyo prototipo es:

```
int pthread_create(pthread_t *thread,
   const pthread_attr_t *attr,
   void *(*start_routine)(void*), void *arg);
```

La función es usada para crear un nuevo hilo, con los atributos especificador por *attr*. Si tiene éxito la función retorna el valor cero, y cualquier otro número en caso contrario.

- Para esperar que un hilo termine se utiliza `pthread_join` cuyo prototipo es:

```
int pthread_join(pthread_t thread, void **value_ptr);
```

La función es usada para suspender la ejecución de quién la invoque hasta que el hilo apuntado por *thread* termine su ejecución. Si tiene éxito la función retorna el valor cero, y cualquier otro número en caso contrario.

- Para esperar que un hilo termine se utiliza `pthread_setaffinity_np` cuyo prototipo es:

```
int pthread_setaffinity_np(pthread_t thread, size_t cpusize,
                           const cpu_set_t *cpuset);
```

La función es usada setear la máscara de afinidad de CPU de un thread, es decir las CPU en las que el thread tiene permitido ejecutarse. Si tiene éxito la función retorna el valor cero, y cualquier otro número en caso contrario. Para usar esta función debe estar previamente seteada la máscara correspondiente en la variable `cpuset`, lo cual se logra por medio de las funciones `void CPU_ZERO(cpu_set_t *set);` para inicializar el conjunto vacio y `void CPU_SET(int cpu, cpu_set_t *set);` para añadir una CPU al conjunto.

Sincronización de hilos

- Para crear un mutex se utiliza la función `pthread_init` cuyo prototipo es:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
```

  Esta función es usada para inicializar un `mutex` con los atributos especificados en `attr`. Si tiene éxito la función retorna el valor cero, y cualquier otro número en caso contrario. Además si la función tiene éxito un `mutex` es inicializado listo para usar, y queda en estado desbloqueado.

- Para bloquear un mutex se utiliza la función `pthread_mutex_lock` cuyo prototipo es:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

  Esta función permite bloquear un `mutex` de modo que ningún otro hilo pueda hacerlo hasta que el `mutex` no sea liberado, bloqueándose a la espera de esto. Si tiene éxito retorna cero, y cualquier otro valor en caso contrario.

- Para bloquear un mutex se puede utilizar también la función `pthread_mutex_trylock` cuyo prototipo es:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

  Esta función permite bloquear un `mutex` de modo que ningún otro hilo pueda hacerlo hasta que el `mutex` no sea liberado, pero a diferencia del anterior devuelve un valor que permite saber si el `mutex` fue bloqueado o no permitiéndole al hilo continuar con su ejecución dependiendo de este valor. Si el bloqueo del `mutex` fue exitoso la función retorna cero, y cualquier otro valor en caso contrario.

- Para liberar un mutex se utiliza la función `pthread_mutex_unlock` cuyo prototipo es:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

  Esta función permite liberar un `mutex` de modo el hilo o uno de los hilos que estaban esperando por él puedan tener acceso. Si tiene éxito retorna cero, y cualquier otro valor en caso contrario.

- Para destruir un mutex se utiliza la función `pthread_mutex_destroy` cuyo prototipo es:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

  Esta función permite borrar un `mutex` dejándolo inutilizable para cualquier hilo. Si tiene éxito retorna cero, y cualquier otro valor en caso contrario.

ANEXO B

MPI

A continuación se muestran una colección de rutinas que es suficiente para los algoritmos desarrollados en esta tesina. Detalles adicionales pueden encontrarse en la siguiente referencia [Ope04].

El archivo <mpi.h> contiene los prototipos de las funciones, por lo cual debe incluirse en el código para usar las funciones que la librería provee.

Manejo del ambiente

- Para inicializar el ambiente MPI se utiliza la función `MPI_Init` cuyo prototipo es:

```
int MPI_Init(int *argc, char ***argv);
```

  Esta función es usada para inicializar el ambiente MPI y recibe como parámetros los mismos parámetros que recibe el `main` del programa y debe ser invocada por todos los procesos MPI. Esta función entre otras cosas setea el comunicador `MPI_COMM_WORLD` el cual contiene a todos los procesos de la aplicación.

- Para finalizar la ejecución del ambiente MPI se utiliza la función `MPI_Finalize` cuyo prototipo es:

```
int MPI_Finalize(void);
```

  Esta función es usada para cerrar el ambiente MPI y no recibe parámetros. Debe ser invocada por todos los procesos MPI.

- Para conocer el id o Rank de un proceso MPI dentro de un comunicador se utiliza la función `MPI_Comm_Rank` cuyo prototipo es:

```
int MPI_Comm_rank( MPI_Comm comm, int *rank );
```

  Esta función es invocada por un proceso MPI cuando desea conocer su Rank dentro de un comunicador y devuelve este valor en la variable `rank`, y es un número entre cero y la cantidad de procesos en el comunicador menos uno.

- Para conocer la cantidad de procesos MPI dentro de un comunicador se utiliza la función `MPI_Comm_size` cuyo prototipo es:

```
int MPI_Comm_size( MPI_Comm comm, int *size);
```

  Esta función devuelve en la variable `size` la cantidad de procesadores que existen en el comunicador `comm`.

Tipos de datos

MPI define varios tipos de datos para MPI_Datatype, la mayoría se corresponden con los tipos de C.

Tipo de dato MPI	Tipo de dato C
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI _LONG_DOUBLE	Long double

Pasaje de mensajes

- Para enviar un mensaje MPI se utiliza la función `MPI_Send` cuyo prototipo es:

```
int MPI_Send(const void *buf, int count,
                    MPI_Datatype datatype, int dest, int tag,
                    MPI_Comm comm);
```

Esta función es para enviar el mensaje almacenado en `buf` de `count` bytes por medio del comunicador `comm` al procesos cuyo Rank dentro de `comm` es `dest`. La función retorna el control una vez que el mensaje fue transmitido.

- Para recibir un mensaje MPI se utiliza la función `MPI_Recv` cuyo prototipo es:

```
int MPI_Recv(const void *buf, int count,
                    MPI_Datatype datatype, int source, int tag,
                    MPI_Comm comm, MPI_Status *status);
```

Esta función es utilizada para recibir un mensaje proveniente del proceso cuyo Rank dentro del comunicador `comm` es `source`. El mensaje recibido es de `count` bytes y es almacenado en `buf`. La función retorna el control una vez que el mensaje fue recibido. El parámetro `tag` podría declararse con el valor `MPI_ANY_TAG` y así cualquier tag sería válido. Así también como el parámetro `source` podría ser reemplazado por `MPI_ANY_SOURCE` y así sin importar de donde provenga el mensaje si está dirigido a ese proceso y chequea que los demás parámetros coinciden lo va a recibir. Para recibir un mensaje es necesario tener el atributo `status` del tipo `MPI_Status` el cual es la estructura que contiene tres atributos, `MPI_SOURCE`, `MPI_TAG` y `MPI_ERROR`.

- Para enviar un mensaje de forma no bloqueante se utiliza la función `MPI_Isend` cuyo prototipo es:

```
int MPI_Isend(const void *buf, int count,
```

```
                MPI_Datatype datatype, int dest, int tag,
                             MPI_Comm comm, MPI_Request request);
```

Esta función permite el envío de un mensaje de forma no bloqueante, es decir que ni bien se inicia la transferencia la función retorna sin importa si el mensaje se terminó de transferir o no.

- Para recibir un mensaje de forma no bloqueante se utiliza la función `MPI_Irecv` cuyo prototipo es:

```
int MPI_Irecv(const void *buf, int count,
              MPI_Datatype datatype, int dest, int tag,
                           MPI_Comm comm, MPI_Request request);
```

Esta función permite la recepción de un mensaje de forma no bloqueante, es decir que ni bien se inicia la transferencia la función retorna sin importa si el mensaje se terminó de transferir o no.

- Para comprobar si un mensaje enviado de forma no bloqueante ha sido transmitido de forma completa se utiliza la función `MPI_Test` cuyo prototipo es:

```
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

Operaciones colectivas

- Para enviar un mensaje MPI a todos los procesos de un comunicador se utiliza la función `MPI_Bcast` cuyo prototipo es:

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm);
```

Esta función es usada para enviar un mensaje a todos los procesos en `comm` y a sí mismo. Todos los procesos del comunicador deben invocar a esta función.

- Para recolectar y concatenar datos de todos los procesos de un comunicador se utiliza la función `MPI_Gather` cuyo prototipo es:

```
MPI_Gather (void *sendbuf, int sendcount,
             MPI_Datatype sendtype, void *recvbuf,

             int recvcount, MPI_Datatype recvtype, int root,

             MPI_Comm comm);
```

Esta función permite que el proceso con rank `root` dentro del comunicador `comm` reciba datos alojados en `sendbuf` de todos los procesos de `comm` y los concatene por orden de Rank en `recvbuf`.

- Para combinar los datos enviados por todos los procesos de un comunicador se usa la función `MPI_Reduce` cuyo prototipo es:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op, int root,
               MPI_Comm comm);
```

Esta función combina los valores de tipo `datatype` alojados en `sendbuf` de todos los procesos del comunicador `comm` con la operación declarada en `MPI_Op` (la cual puede ser `MPI_MAX` para calcular el máximo, `MPI_MIN` para calcular el mínimo, `MPI_SUM` para calcular la suma, y `MPI_PROD` para calcular el producto). Este valor calculado se almacena en `recvbuf` en el proceso con Rank `root` dentro del comunicador.

Bibliografía

[Ana05] **Ananias Pablo Itaim y Spading Andreas** El problema de las N-reinas. Valparaíso, 2005.

[Ana03] **Ananth Gramma Anshul Gupta, George Karpis, Vivin Kuman** Introduction to Parallel Computing. Second Edition - Pearson Education, 2003.

[And00] **Andrews Gregory R.** Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley, 2000.

[Luz04] **Antonio Luz Marina Moreno De** Computación paralela y entornos Herelogéneos, 2004.

[Att99] **Attwood Teresa K. y Parry-Smith David J.** Intruuction to Bioinformatics. Prentice Hall, 1999.

[Bar10] **Barney Blaise** Introduction to parallel computing (Lawrence Livermore National) [En línea], 2010. Consulta realizada el 10 de Octubre de 2014. - https://computing.llnl.gov/tutorials/parallel_comp/.

[Bar05] **Barry Wilkinson Michael Allen** Parallel Programming - Techniques and Aplication Using Networked Workstations adn Parallel Computer. Pearson Pretince Hall, 2005.

[Bra96] **Bradford Nichols, Dick Buttlar y Proulx Farrell Jacqueline** Pthreads Programming. Estados Unidos. O'Reilly & Associates, Inc, 1996.

[Bru75] **Bruen A. y Dixon R.** Then N-queens Problem. Discrete Mathematics, 1975.

[Fra13] **Chichizola Franco** Efecto de la distribución de trabajo en aplicaciones paralelas irregulares sobre clusters heterogéneos. La Plata, Buenos Aires, Argentina, 2013.

[Cho09] **Chow Edmond and Hyson David** Assesing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters, 2009.

[Cra92] **Crawford K. D.** Solving the N-queens problem using genetic algorithms. Kansas City, 1992.

[DeG05] **De Giusti Armando Eduardo Naiouf Marcelo, De Giusti Laura, Chichizola Franco** Balance dinámico de carga en procesamiento paralelo sobre clusters no-homogéneos , 2005.

[Don03] **Dongarra Jack [y otros]** Sourcebook Of Parallel Computing, Elsevier Science, 2003.

[ELe04] **E. Leiss y Aguilar J.** Introducción a la Computación Paralela. Mérido, Venezuela, 2004.

[Fab12] **Fabiana Libocih Franco Chichizola, Laura De Giusti, Marcelo Naiouf. Francisco Tirado Fernández, Armando De Giusti** Programación híbrida en clusters de multicore. Análisis del impacto de la jerarquía de memoria, 2012.

[Fab10] **Fabiana Libovich Armando De Giusti, Marcelo Naiouf, Laura De Giusti, Franco Chichizola** Programación híbrida en arquitecturas cluster de multicore. Escalabilidad y comparación con memoria compartida y pasaje de mensajes, 2010.

[Fer04] **Fernando G. Tinetti Andrés Barbieri, Mónica Denham, Franco Chichizola, Laura De Giusti, Marcelo Naiouf, Armando De Giusti** Procesamiento Paralelo en Clusters, 2004.

[Fer08] **Fernando G. Tinetti Gustavo Wolfmann** Análisis de Paralelización con Memoria Compartida y Memoria Distribuida en Cluster de Nodos con Múltiples Núcleos, 2008.

[Fos95] **Foster Ian** Designing and building parallel programs. Addison-Wesley Longman Publishing Co, 1995.

[Lau08] **Giusti Laura De** Mapping sobre Arquitecturas Heterogéneas, 2008.

[Eli13] **Gurovich Elisa Viso** El reto de las arquitecturas multinúcleo (multicore), 2013.

[Lau99] **Laura De Giusti Diego Tarrío** Escalabilidad en algoritmos paralelos de cálculo del costo mínimo de caminos en grafos, 1999.

[Cla00] **Leopold Claudia** Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches, 2000.

[Gor98] **Moore Gordon E.** Cramming More Components onto Integrated Circuits, 1998.

[Mou99] **Moura Luis y Buyya Rajkumar** High performance cluster computing. Prentice Hall PTR, 1999. Vol. 2.

[Pau07] **Paul Werstein Hailing Situ, Zhiyi Huang** Load Balancing in a Cluster Computer, 2007.

[Ope04] **Project Open MPI** Open MPI - Open Source High Performance Computing [En línea], 2004. Consulta realizada el 12 de Octubre de 2014. - http://www.open-mpi.org/.

[QUE14] **QUEENS@Technische** Universitat Drenden [En línea]. Technische Universitat Drenden. Consulta realizada el 17 de Julio de 2014. - http://queens.inf.tu-dresden.de/?l=en&n=f.

[Raf05] **Rafael B. Garcia Jorge R. Ardenghi** Consistencia de ejecución: una propuesta No Caché Coherente, 2005.

[Rau10] **Rauber T. Runger G.** Parallel programming for multicore and cluster systems. Springer, 2010.

[Sar95] **Sartaj Sahni Venkat Thanvantri** Parallel Computing: Performance Metrics and Models, 1995.

[Tin04] **Tinneti Fernando** Tesis Doctoral: Cómputo Paralelo en Redes Locales de Computadoras. España. Universidad Autónoma de Barcelona, 2004.

[Fer98] **Tinneti Fernando G. y Giusti Armando De** Procesamiento Paralelo - Conceptors de Arquitecturas y Algoritmos. La Plata, Buenos Aires, Argentina, 1998.