Efecto de la distribución de trabajo en aplicaciones paralelas irregulares sobre clusters heterogéneos

Autor: Franco Chichizola

Director: Armando E. De Giusti

Codirector: R. Marcelo Naiouf

Trabajo Final presentada para obtener el grado de Especialista en Cómputo de Altas Prestaciones y Tecnología GRID

Facultad de Informática - Universidad Nacional de La Plata

Abril 2013

Índice

Objetivo

El objetivo de este Trabajo Final es comparar el efecto de la distribución de trabajo estática y dinámica sobre arquitecturas de cluster heterogéneo, analizando al mismo tiempo el speedup paralelo teórico y el obtenido experimentalmente para un determinado tipo de problema.

En particular, se elige una aplicación clásica (Parallel N-Queens) con un algoritmo de solución paralela en la que predomina el procesamiento sobre el tamaño de los datos, de modo de profundizar en los aspectos del balance de carga (estático o dinámico) sin una distorsión de los resultados producida por aspectos relacionados al uso de la memoria y/o al tamaño de los mensajes a comunicar.

Se presentan tres estrategias de distribución de carga en los procesadores, estudiando en cada caso el desbalance de carga y el speedup paralelo teórico y real en función del tamaño del problema y los procesadores utilizados.

Se ha empleado una combinación de 4 clusters interconectados, donde las máquinas dentro de cada cluster poseen procesadores homogéneos, pero diferentes entre clusters. De este modo el conjunto puede verse como un cluster heterogéneo de 43 procesadores.

Los temas a abordar comprenden el análisis del problema, el estudio de distintas implementaciones del algoritmo secuencial, posibles optimizaciones al mismo, y el desarrollo de soluciones paralelas (utilizando la librería de programación para memoria distribuida MPI) con diferentes políticas de distribución de trabajo que se adecuen a la arquitectura y a la aplicación utilizada. Por otro lado, se analiza la manera de determinar el rendimiento de las aplicaciones en arquitecturas heterogéneas de forma tal de tener una métrica real del mismo teniendo en cuenta las diferencias de las máquinas del cluster.

Este trabajo pretende determinar la mejor opción de distribución de trabajo para este tipo de problemas sobre arquitecturas heterogéneas de memoria distribuida, analizando el rendimiento de las soluciones para diferentes configuraciones de tamaño del problema y de la arquitectura.

Capítulo 1- Introducción

En la Ciencia de la Computación existe una continua demanda de mayor poder de procesamiento por parte de los sistemas informáticos. Existen numerosas áreas en las que se requiere realizar grandes cantidades de cálculos, o trabajar con un importante volumen de datos, y en todos los casos realizarlo en un período de tiempo razonable de acuerdo al problema a resolver [Grama, et al., 2003].

Desde el surgimiento de las computadoras seriales, su velocidad se ha incrementado para cumplir con las necesidades de las aplicaciones del mundo real. Sin embargo, la limitación física fundamental impuesta por la velocidad de la luz hace imposible obtener mejoras indefinidamente, y una manera natural de evitar esta saturación e incrementar el poder de cómputo es utilizar múltiples procesadores (en una o más máquinas) trabajando de manera conjunta para resolver el problema. En este sentido, el paralelismo es un concepto intuitivo porque el mundo real es esencialmente paralelo [Wilkinson & Allen, 2004].

Las causas anteriormente mencionadas han llevado a aumentar la importancia del procesamiento paralelo y distribuido dentro de la Ciencia de la Computación e incrementar el interés en su investigación y desarrollo, convirtiéndose en una de las áreas que más profundamente han transformado a la disciplina [Jordan & Alaghband, 2002].

Los objetivos principales del paralelismo están relacionados con la posibilidad de ajustar el modelo de arquitectura y software al mundo real, y con reducir el tiempo necesario para resolver problemas mediante la utilización de múltiples procesadores, además de mejorar la eficiencia de los mismos.

Existen diferentes razones que justifican la importancia que ha adquirido el procesamiento paralelo y distribuido. Entre ellas pueden mencionarse [Naiouf, 2004]:

- El crecimiento de la potencia de cómputo, dado por la evolución de la tecnología de los componentes y las arquitecturas de procesamiento.
- La existencia de problemas computacionales en los cuales el tiempo de resolución utilizando una máquina secuencial es inaceptable.
- La transformación y creación de algoritmos que exploten la concurrencia implícita del problema a resolver, de modo de distribuir el procesamiento minimizando el tiempo de respuesta. Naturalmente esta transformación también debe adaptarse a la arquitectura física de soporte.

- La capacidad del cómputo distribuido/paralelo de reducir el tiempo de procesamiento en problemas de cálculo intensivo (simulaciones, búsquedas, cómputo científico) o de grandes volúmenes de información (bases de datos, imágenes, entre otros).
- La existencia de sistemas en los que no es tan importante la velocidad de cómputo sino con tiempos de respuesta críticos.
- Las posibilidades que el paradigma paralelo ofrece en términos de investigación de técnicas para análisis, diseño y evaluación de algoritmos.

Esta evolución conduce a un gran esfuerzo por transformar el procesamiento secuencial en paralelo. Naturalmente, la creación de algoritmos paralelos/distribuidos, o la transformación de un algoritmo secuencial en paralelo, se encuentran lejos de ser un proceso directo y está influida por la arquitectura física de soporte. Un *sistema paralelo* es la combinación de un algoritmo paralelo y la máquina sobre la cual éste se ejecuta [Ben-Ari, 2006].

La performance obtenida en el sistema paralelo está dada por una compleja relación en la que intervienen factores como el tamaño del problema, la arquitectura de soporte, la distribución de procesos en procesadores, la existencia o no de un algoritmo de balanceo de carga, entre otros. Existe un gran número de métricas para evaluar sistemas paralelos (tiempo efectivo, speedup, eficiencia), y las mismas deben ser adaptadas de acuerdo a la evolución de las arquitecturas paralelas.

Los clusters constituyen una plataforma de cómputo paralelo muy utilizada por sus ventajas en cuanto a la relación costo/performance, es decir que se puede lograr una importante potencia de cómputo a un costo relativamente bajo. Es un tipo de arquitectura de procesamiento paralelo/distribuido que consiste de un conjunto de computadoras interconectadas que pueden actuar como una máquina única. Las máquinas que componen un cluster pueden ser homogéneas o heterogéneas, lo que constituye un factor importante en el análisis de la performance que se puede obtener de un cluster como máquina paralela.

1.1. Balance de carga en arquitecturas heterogéneas

El balance de carga de una aplicación incide directamente en el speedup alcanzable y en el rendimiento del sistema paralelo. En general, cuando se trabaja con clusters heterogéneos las diferentes potencias de cálculo de las máquinas intervinientes son un factor que puede computarse para analizar una mejor distribución del trabajo a realizar.

En las clases de problemas de trabajo conocido (por ejemplo, multiplicación de matrices) puede obtenerse un balance de carga estático "predictivo" en función de la potencia de cálculo de los procesadores del cluster; sin embargo, muchos problemas reales tienen una carga de trabajo variable o dinámica en función de los datos. En estos casos es necesario ajustar

dinámicamente la asignación de datos o procesos a los diferentes procesadores, a medida que la aplicación se ejecuta.

Por otra parte, en un esquema en el que se resuelven aplicaciones con el paradigma *Master/Worker*, cualquier solución de balance dinámico implica un overhead de comunicaciones que es afectado por la complejidad del esquema de comunicación entre los nodos de los diferentes clusters.

1.2. Clases de problemas con carga de trabajo variable

Existen clases de problemas de paralelismo de datos en los que es posible realizar una asignación estática equilibrada de la carga de trabajo total (en general, los que tienen un patrón de ejecución regular, como por ejemplo algunas soluciones al problema de multiplicación de matrices). En estos casos, si se cuenta con una arquitectura heterogénea se puede definir una función predictiva $F(P_i,W_t)$ con la cual distribuir los datos "a priori" entre los procesadores, donde P_i es la potencia de cálculo del procesador i y W_t el trabajo total [Bohn & Lamont, 2002].

En el caso de tener una carga de trabajo variable en función de características de los datos (porque tienen un patrón de ejecución irregular o son de naturaleza dinámica, como por ejemplo ordenación de datos o reconocimiento de patrones en imágenes), no es posible definir una función predictiva que asegure el balance de carga entre los procesadores. En estos casos es necesario tener alguna política de asignación dinámica que puede combinarse con una distribución inicial predictiva de un porcentaje de los datos totales [Naiouf, 2004][Watts & Taylor, 1998][Baiardi, et al., 2001].

Cualquier política de asignación dinámica implica algún grado de overhead de comunicaciones, que será más complejo de modelizar y predecir en una arquitectura heterogénea.

1.3. Dificultades del procesamiento paralelo

En algunos casos el costo del paralelismo puede ser alto en término del esfuerzo de programación requerido: debe pensarse en la aplicación de técnicas nuevas, rescribiendo completamente el código serial, y las técnicas de *debugging* y *tuning* de performance secuenciales no se extienden fácilmente al mundo paralelo.

Los problemas son paralelizables en distintos grados. Para algunos, asignar particiones a otros procesadores podría significar mayor consumo de tiempo que realizar el procesamiento localmente. Otros problemas pueden ser esencialmente secuenciales. Un problema puede

tener distintas formulaciones paralelas, lo que puede resultar en beneficios variados, y todos los problemas no son igualmente adecuados para el procesamiento paralelo.

Esto lleva a que la construcción de algoritmos paralelos significa un desafío ya que, entre otras cuestiones, es necesario tener en cuenta la arquitectura sobre la cual se ejecutará el programa, el modelo de comunicación utilizado y la manera de dividir el problema y los datos.

Por otro lado, existen distintas alternativas de implementación sobre sistemas de cómputo paralelo con hardware de procesamiento homogéneo o heterogéneo y con acoplamiento fuerte o débil de sus componentes. Estas alternativas abren una amplia gama de posibilidades que se traducen a su vez en costos que se deben evaluar fundamentalmente desde el punto de vista de la relación costo/rendimiento. Se debe tener en cuenta que el costo involucrado no está referido *solamente* al hardware específico de procesamiento sino que involucra también el desarrollo y/o la adaptación de los algoritmos a la arquitectura de procesamiento.

No debe perderse de vista el hecho de que un programa paralelo implica la necesidad de asignar (mapear) procesos lógicos a procesadores físicos. Este es un tema complejo y de fundamental importancia para el éxito o el fracaso en la resolución eficiente del problema.

Capítulo 2 – Arquitectura paralela

Los Sistemas Paralelos integran, los algoritmos de procesamiento (software) con la arquitectura de soporte (hardware). Cuando se quiere obtener buen rendimiento de una arquitectura multiprocesador, es necesario conocer con un alto nivel de detalle la configuración de hardware disponible y analizar (en base a este dato) los métodos y algoritmos de software a utilizar [De Giusti, 2011].

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, y/o resolver problemas de mayor tamaño. En este trabajo en particular interesan las arquitecturas distribuidas llamadas *Cluster Heterogéneos*.

2.1. Cluster heterogéneo de PCs.

El término cluster se aplica a los conjuntos de computadoras construidos mediante la utilización de componentes de hardware estándar y que se comportan como si fuesen una única computadora. Estas arquitecturas juegan un papel importante en la solución de problemas de las ciencias, las ingenierías y del comercio moderno [Juhasz, et al., 2004]. La tecnología de clusters ha evolucionado en apoyo de actividades que van desde aplicaciones de supercómputo y software de misión crítica, servidores web y comercio electrónico, hasta bases de datos de alto rendimiento, entre otros usos.

El cómputo con clusters surge como resultado de la convergencia de varias tendencias que incluyen la disponibilidad de microprocesadores económicos de alto rendimiento y redes de alta velocidad, el desarrollo de herramientas de software para cómputo distribuido de alto rendimiento, así como la creciente necesidad de potencia computacional para aplicaciones que la requieran [Juhasz, et al., 2004]. De un cluster se espera que presente combinaciones de los siguientes servicios:

- Alto rendimiento (High Performance).
- Alta disponibilidad (High Availability).
- Balance de carga (Load Balancing).
- Escalabilidad (Scalability).

Para que un cluster funcione como tal, no basta sólo con conectar entre sí las computadoras, sino que es necesario proveer un sistema de gestión del cluster, el cual se

encargue de interactuar con el usuario y los procesos que corren en él para optimizar el funcionamiento.

Entre los componentes de un cluster se pueden mencionar: los nodos, el sistema operativo, la red de interconexión, el middleware (capa de abstracción entre el usuario y el sistema operativo), los protocolos de comunicación y servicios, y por último las aplicaciones (que pueden ser paralelas o no).

Este tipo de arquitecturas presenta dos características importantes que deben ser tenidas en cuenta en el momento de utilizarlas para realizar cómputo paralelo:

- Los nodos que forman el cluster están débilmente acoplados. Esto se puede sintetizar en los siguientes aspectos:
  - La memoria física de las computadoras en la red local está totalmente distribuida y no hay ningún medio de hardware que facilite compartirla.
  - El procesamiento en una red local es totalmente asincrónico en cada máquina, y no hay ningún medio de hardware que facilite la sincronización.
  - La única forma de hacer que un procesador acceda a información de otra máquina es utilizando la red de comunicaciones. De la misma manera, la única forma de sincronización entre los procesadores de cada computadora de la red local es utilizando la red de comunicaciones.
  - Descartar el uso de un espacio de direcciones común implica replantear algunos algoritmos diseñados para computadoras paralelas con memoria compartida.
- El rendimiento de la red de interconexión se puede ver afectado por diferentes aspectos. Entre ellos se pueden mencionar:
  - El tiempo propio de las comunicaciones es alto frente a los accesos a memoria.
  - El costo del tiempo de latencia (startup) de inicio de la comunicación entre dos procesadores.
  - El bajo ancho de banda (normalmente total o parcialmente compartido) disponible para los procesadores de cada cluster.

Ambas características llevan a priorizar el desarrollo de algoritmos con la mayor capacidad de cómputo local y la posibilidad de solapamiento entre comunicaciones inter-procesadores y procesamiento de datos sobre cada procesador.

La construcción de los nodos del cluster es relativamente fácil y económica debido a su flexibilidad, logrando una buena relación costo/rendimiento y al mismo tiempo ser fácilmente escalables [Andrews, 2000]. En estas arquitecturas paralelas pueden tener todos los procesadores la misma configuración de hardware y sistema operativo (cluster homogéneo), o

tener diferente hardware y/o sistema operativo (cluster heterogéneo). Esta flexibilidad constituye un factor importante en el análisis de la performance que se puede obtener de un cluster como máquina paralela [Al-Jaroodi, et al., 2003].

En general, las redes locales instaladas pueden ser ampliamente heterogéneas en cuanto al hardware de procesamiento; en ellas se pueden encontrar diferentes computadoras (PCs, SMPs, Workstations). Más aún, dentro de una línea de PCs con igual procesador, pueden existir diferentes capacidades de procesamiento (asociadas por ejemplo al tamaño y velocidad de memoria caché o frecuencia de acceso al bus) [Juhasz, et al., 2004].

Esta diferencia de velocidad de cómputo relativa de los procesadores que forman la arquitectura es un aspecto relevante que agrega aún más complejidad en el proceso de generar aplicaciones paralelas. Para obtener el máximo rendimiento posible en un cluster heterogéneo necesariamente se debe llevar a cabo un balance de carga de procesamiento adecuado. Este balance puede partir de un análisis estático de las características de la arquitectura (capacidad de cómputo, memoria, caché, entre otras), aunque puede requerir un estudio dinámico en función de la aplicación.

2.2. Modelo de programación de Pasaje de Mensajes

En este tipo de arquitecturas paralelas distribuidas (cluster heterogéneo), se utiliza el modelo de programación de Pasaje de Mensajes en el cual varios procesos ejecutan en paralelo y se comunican enviando y recibiendo mensajes. No tienen acceso a una memoria compartida, es decir que operan sobre espacios de direcciones disjuntos y toda la comunicación se realiza por intercambio explícito de mensajes.

El intercambio de mensajes puede servir a distintos propósitos. El más obvio es compartir datos entre un emisor y un receptor que se conocen y cuya interacción fue planeada por el programador. El segundo es el establecimiento de una conexión emisor/receptor no planeada de antemano (mediante un *receive* anónimo hecho por el receptor). Y el tercer propósito es la *sincronización*, que es un caso especial de comunicación. En las rutinas de emisión y recepción el programador debe especificar detalles de bajo nivel como la identidad del receptor, la dirección y tamaño de un buffer donde dejar o de donde sacar los datos, y la longitud del mensaje [Naiouf, 2004].

El modelo de pasaje de mensajes se basa claramente en la arquitectura de máquinas de memoria distribuida y por lo tanto es fácil de implementar, sin embargo opera a bajo nivel y tiene las siguientes consecuencias:

- *Programabilidad*. Impone una pesada carga al programador al ser el responsable de manejar todos los detalles de distribución de datos y procesos, así como la comunicación entre éstos.

- *Eficiencia*. Dado que todo está bajo el control del programador, éste puede lograr un rendimiento cercano al óptimo si le dedica suficiente tiempo al ajuste de la aplicación a la arquitectura.
- *Portabilidad*. Dado que el pasaje de mensajes fue establecido en forma temprana, se han desarrollado estándares. Esto no garantiza la portabilidad de rendimiento, sino sólo de código.

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.

Capítulo 3 - Algoritmos paralelos

El diseño de algoritmos paralelos no se reduce a simples recetas, sino que requiere un pensamiento integrador con cierto grado de *creatividad*. Sin embargo, puede beneficiarse por un enfoque metódico que maximice el rango de opciones consideradas, provea mecanismos para evaluar alternativas, y reduzca el costo de *backtracking* por malas elecciones.

La mayoría de los problemas tienen distintas soluciones paralelas, y la mejor puede diferir esencialmente de las secuenciales existentes. Una posibilidad de diseño es considerar temas tales como independencia de la máquina y concurrencia en etapas iniciales, dejando aspectos específicos de la máquina para el final. Esta metodología estructura el proceso en cuatro etapas: las dos primeras se enfocan en la concurrencia y escalabilidad buscando algoritmos con estas cualidades, mientras que en la tercera y cuarta la atención se mueve a la localidad y temas relativos a la performance:

1. *Particionamiento*. Esta etapa se refiere a la descomposición del cómputo y los datos sobre los que se trabaja en muchas tareas pequeñas (de *grano fino*). Se busca evitar la replicación de cómputo y datos; este aspecto puede ser revisado luego, ya que puede convenir hacerlo para reducir los requerimientos de comunicación [Moura e Silva & Buyya, 1999] [Naiouf, 2004].
2. *Comunicación*. Se intenta que las tareas generadas en la etapa anterior ejecuten concurrentemente, pero en general no son totalmente independientes. El cómputo que debe realizar una tarea a menudo requiere datos asociados a otra, los cuales deben transferirse. El flujo de información es especificado en esta fase mediante dos etapas: definir la estructura de canales que enlacen las tareas, y especificar los mensajes que viajaran sobre ellos [Naiouf, 2004].
3. *Aglomeración*. El algoritmo resultante de las etapas anteriores es aún abstracto en el sentido de que no se tiene en cuenta la arquitectura particular sobre la que se ejecutará. En la etapa de aglomeración se revisan las decisiones tomadas con el objetivo de obtener un algoritmo que ejecute de manera eficiente en una arquitectura real [Naiouf, 2004]. Se considera si es útil: combinar o aglomerar las tareas identificadas obteniendo otras de mayor granularidad para mejorar la performance; empaquetar varias comunicaciones en una super-comunicación; replicar datos y/o cómputo.
4. *Mapeo*. En esta última etapa se especifica donde ejecutar cada tarea. Generalmente el objetivo del mapeo es minimizar el tiempo de ejecución total (buscando balancear la carga de trabajo), y pueden usarse dos estrategias: ubicar tareas que son capaces de ejecutar concurrentemente sobre diferentes procesadores para mejorar la

concurrencia, o poner las tareas que se comunican frecuentemente en el mismo procesador para incrementar la localidad.

Un *paradigma de programación* es una clase de algoritmos que resuelve problemas distintos, pero que tienen la misma estructura de control [Kumar, et al., 2012]. Para cada paradigma puede escribirse un programa general que define la estructura de control común; éste es llamado *esqueleto algorítmico*, *programa genérico* o "*template*" *de programa* [McCool, et al., 2012]. En la programación paralela pueden distinguirse una serie de paradigmas que permiten encuadrar los problemas en alguno de ellos y simplifican la tarea del programador. Unos pocos de ellos son usados repetidamente para desarrollar la mayoría de las aplicaciones paralelos.

La selección del paradigma es determinado por la disponibilidad de recursos de cómputo paralelo, y por el tipo de paralelismo inherente en el problema. Los recursos de cómputo pueden definir el nivel de granularidad que puede soportar en forma eficiente el sistema, mientras que el tipo de paralelismo refleja la estructura paralela de la aplicación y/o los datos.

Hay muchos autores que presentan clasificaciones de paradigmas de programación paralela. No todos proponen exactamente lo mismo pero coinciden parcialmente, por lo tanto, se puede formar un subconjunto que englobe a la gran mayoría de las aplicaciones paralelas [Moura e Silva & Buyya, 1999]. En este trabajo en particular se utiliza el paradigma *Master/Worker* (o *Master/Slave*, dependiendo la bibliografía), el cual se describe a continuación.

3.1. Paradigma de programación paralela Master/Worker

En el paradigma *Master/Worker*, un proceso actúa como *coordinador* (*master*) dirigiendo a todos los otros que funcionan como *trabajadores* (*workers*). El *master* es responsable de descomponer el problema en tareas pequeñas, distribuirlas entre los diferentes procesos, y recibir los resultados parciales para componer la solución final del problema. Los *workers* realizan un procesamiento muy simple: reciben una tarea a realizar, la procesan, y envían los resultados al *master*. Este ciclo puede repetirse hasta que el *master* no tenga más tareas para resolver [Moura e Silva & Buyya, 1999].

Los resultados parciales que recibe el *master* pueden generar nuevas tareas para ser distribuidas entre los *workers* [Naiouf, 2004]. Pueden distinguirse dos casos básicos de acuerdo a la dependencia entre las iteraciones:

- *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. En estos casos, el *master* debe sincronizar a todos los *workers* en cada iteración.

- *Entradas de datos independientes*. Los datos arriban (o se generan) en el *master* sin necesitar los resultados anteriores para realizar una nueva distribución de los mismos. Los *workers* no requieren sincronización entre ellos.

Respecto al momento en el cual el *master* realiza la distribución de los datos, existen dos opciones básicas [Naiouf, et al., 2006]:

- *Distribución estática*. Distribuir todas las tareas cuando el *master* ha finalizado de hacer la descomposición. Este tipo de distribución es válido cuando la cantidad de tareas (o trabajo) para repartir y el tamaño de las mismas es conocido desde el comienzo de la aplicación. Una de las ventajas es que el *master* puede realizar parte de las tareas una vez que ha repartido el resto de ellas a los *workers*. La Figura 3.1 muestra una representación esquemática de este tipo de distribución. De acuerdo a las características de la arquitectura se puede dividir en: *directa* (cuando la arquitectura es homogénea, las tareas se distribuyen en forma equitativa) y *preditiva* (cuando la arquitectura es heterogénea, las tareas se distribuyen proporcionalmente a la potencia de cómputo de cada procesador).

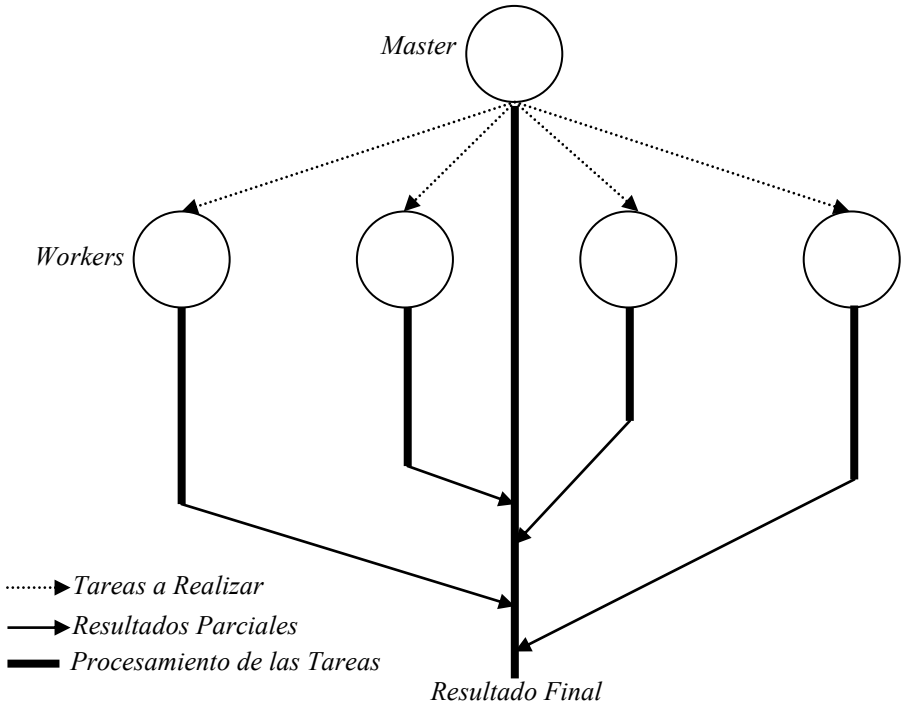


Figura 3.1. Representación esquemática de la *distribución estática*.

- *Distribución dinámica bajo demanda*. En primera instancia se distribuye un cierto porcentaje de trabajo en forma *estática* (*directa* o *predictiva* de acuerdo a la arquitectura), y el resto se reparte *bajo demanda,* es decir que cuando el *worker* está libre le pide más trabajo al *master*. Cuando a priori no se conoce la cantidad de tareas a realizar o el tamaño de las mismas, este método reduce el desbalance de carga, a costa de incrementar la cantidad de comunicación y sincronización.

Este paradigma se puede generalizar a un *master/worker multinivel*. Por ejemplo, en uno de dos niveles, se posee un *master* de nivel superior que envía subconjuntos de tareas a los de nivel inmediatamente inferior; cada uno de estos *master* de segundo nivel distribuye las tareas recibidas entre los *workers* asociados a él [Grama, et al., 2003].

Se debe tener cuidado de que el proceso *master* no genere un cuello de botella donde los procesos *worker* permanezcan ociosos mientras esperan a que se les asigne más trabajo. Si las tareas a realizar son muy pequeñas, los *workers* estarían continuamente pidiendo más trabajo en lugar de procesándolo, de manera que los diferentes procesos compiten por la atención por parte del *master*. Para resolver este problema, la granularidad de la aplicación paralela debe ser tal que el costo de procesar cada tarea sea superior al de transferir el trabajo (comunicación) y sincronizar entre ambos procesos (*master* y *worker* respectivo) [Grama, et al., 2003].

Capítulo 4 - Evaluación de sistemas paralelos

En la programación secuencial el rendimiento, generalmente, es medido teniendo en cuenta los requerimientos de tiempo y memoria de un programa. En la práctica, lo segundo sólo es importante para determinar la posibilidad de llevar a cabo la solución con la cantidad de memoria existente en el procesador. En consecuencia, los algoritmos secuenciales son evaluados, usualmente, en términos de su tiempo de ejecución, expresado en función del volumen de trabajo de su entrada.

En el caso de los algoritmos paralelos esto es más complejo, debido a que este tiempo además depende de la cantidad de elementos de procesamiento y su conexión, y sus velocidades relativas de cómputo y comunicación. Por esta razón los algoritmos paralelos no pueden ser analizados en forma aislada de la arquitectura sobre la cual se ejecuta, lo que lleva a hablar de evaluar los *sistemas paralelos*.

Otra de las razones que complican el análisis de performance del sistema paralelo es la variedad de algoritmos que se pueden utilizar para resolver un mismo problema. Esto lleva a la necesidad de seleccionar el mejor de ellos para ser considerado. Algunas de las preguntas que se pueden realizar son, ¿qué interesa medir?, ¿qué indica que un sistema paralelo es mejor que otro?, ¿qué sucede si se utiliza un algoritmo con mayor cantidad de procesadores? [De Giusti, 2011].

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 en programas paralelos

Al aumentar los recursos de hardware utilizados para la ejecución de programas paralelos se espera reducir el tiempo de ejecución del mismo en forma proporcional a ese crecimiento. Es decir que si se utiliza el doble de recursos, se espera que el tiempo se reduzca a la mitad. Sin embargo, esto raramente se da en sistemas paralelos reales debido a diferentes fuentes de overhead asociados con el paralelismo.

Es importante analizar estas fuentes de overhead porque tienen una influencia directa en el rendimiento de los sistemas paralelos; por lo tanto su estudio puede llevar a optimizar y así mejorar los algoritmos desarrollados. En un programa paralelo, el tiempo consumido por los

diferentes elementos de procesamiento de la arquitectura sobre la cual se ejecuta está compuesto por [Grama, et al., 2003]:

- *Cómputo esencial*. El trabajo que realiza el algoritmo secuencial.
- *Interacción entre procesos*. Los sistemas paralelos en general requieren que los procesos interactúen y comuniquen datos entre sí. El tiempo requerido para estas comunicaciones es la principal fuente de overhead en el procesamiento paralelo.
- *Ociosidad de los procesadores*. Los elementos de procesamiento utilizados en un sistema paralelo pueden permanecer ociosos durante la ejecución del mismo por diferentes razones, como mala distribución del trabajo, esperas por sincronización, y/o presencia de componentes seriales de código que solo realiza uno de los procesos.
- *Cómputo extra asociado a la paralelización*. Existen muchos casos en los cuales se debe paralelizar un algoritmo secuencial diferente del óptimo pero con un mayor grado de concurrencia potencial, lo que lleva a que el algoritmo paralelo deba hacer cómputo que no se requería en el programa serie. Por otro lado, algunas veces el algoritmo paralelo debe realizar pasos extras, como por ejemplo repartir el trabajo entre los diferentes procesos.

4.2. Métricas de rendimiento en sistemas paralelos

Es importante estudiar el rendimiento de los programas paralelos para determinar cuál es el "mejor" (más meritorio) para resolver un determinado problema, evaluando la plataforma de hardware que se debe utilizar y examinando el beneficio de la paralelización. Hay diferentes métricas que permiten realizar este análisis centrándose en distintas características del sistema paralelo.

4.2.1. Tiempo de ejecución

Es la más simple (pero fundamental) de las métricas de rendimiento para sistemas paralelos, y está dada por el tiempo transcurrido desde que comienza hasta que termina la ejecución de un algoritmo sobre una determinada arquitectura [De Giusti, 2011].

En el caso de un algoritmo serie, el tiempo de ejecución serie o secuencial (Ts) está dado por el tiempo transcurrido entre el comienzo y la finalización de la ejecución sobre un único elemento de procesamiento.

En el caso de los sistemas paralelos, el tiempo de ejecución paralelo (Tp) es el tiempo transcurrido entre el inicio del primero de los procesos y la finalización del último de ellos, ejecutándose sobre los p elementos de procesamiento de la arquitectura.

Esta métrica tiene el problema de que no sirve para evaluar el mérito del algoritmo en sí, dado que sus resultados no pueden ser extrapolados a otra instancia del problema o de la arquitectura.

4.2.2. Speedup

Cuando se evalúa un sistema paralelo, a menudo interesa saber cuál es la mejora en rendimiento al paralelizar una aplicación con respecto al mejor programa secuencial. El ***speedup*** (*S)* es una medida que permite cuantificar el beneficio relativo de resolver un problema en paralelo, es decir cuan "rápido" ejecuta el algoritmo paralelo respecto al secuencial.

La función de *speedup* se define como la relación entre el tiempo del mejor algoritmo secuencial sobre un simple procesador (*Ts*) y el tiempo requerido para resolver el mismo problema en una arquitectura paralela con una cierta cantidad (*p*) de elementos de procesamiento [Hwang & Xu, 1998]. Esta relación se muestra en la Ecuación 4.1.

$$S = \frac{Ts}{Tp} \tag{4.1}$$

En teoría, cuando la arquitectura es homogénea, el speedup nunca puede exceder la cantidad (*p*) de elementos de procesamiento de la arquitectura paralela. Si eso ocurre, significa que el programa paralelo realiza menos trabajo que el mejor de los algoritmos secuenciales, y esto es una contradicción. A este valor límite se lo llama *speedup óptimo* o *teórico* (S_{opt}) de la arquitectura paralela. En la práctica existen casos muy específicos en los que el speedup supera este límite y se dice que el sistema logra un *speedup superlineal*:

- Se puede dar en problemas de *descomposición exploratoria* donde se avanza en la solución del problema hasta llegar al objetivo buscado, y la cantidad de trabajo realizado depende del lugar donde esté ubicada la meta perseguida [Grama, et al., 2003].
- Al poder dividir los datos con que trabaja la aplicación en porciones que se distribuyen entre los procesadores, permite tener un mejor aprovechamiento de la memoria cache, reduciendo de forma drástica el tiempo de acceso a memoria, y por lo tanto el tiempo final de la aplicación. La mayoría de las veces que se da esta situación el algoritmo secuencial puede optimizarse dividiendo los datos de la misma forma que el paralelo y trabajando secuencialmente en cada porción. Pero hay un reducido grupo de problemas en los que esto no se puede realizar, y por lo tanto se obtiene speedup superlineal.

Cuando la arquitectura utilizada por el sistema paralelo es heterogénea los *p* procesadores que la forman no son idénticos, y por lo tanto el tiempo secuencial (*Ts*) se debe tomar sobre el

mejor de los elemento de procesamiento que la forman. Dado que algunos procesadores poseen una menor potencia de cómputo que el procesador en el que se calcula *Ts*, estos emplearán más tiempo para realizar igual cantidad de trabajo. Esta característica lleva a que el speedup óptimo o teórico ya no está dado por la cantidad de elementos de procesamiento que forman la arquitectura porque eso significaría que todos los procesadores trabajan a la misma velocidad [Bubak, et al., 1997].

Este problema se subsana teniendo en cuenta la potencia de cada procesador, en lo que se denomina *potencia de cómputo relativa* (*pcr*) respecto del mejor procesador de la arquitectura (Ecuación 4.2) [Al-Jaroodi, et al., 2003].

$$pcr_i = \frac{potencia\ (procesador_i)}{potencia\ (procesador_{mejor})} \tag{4.2}$$

Se define como *potencia de cómputo total* (*pct*) de la arquitectura paralela en relación al mejor procesador, a la suma de la *potencia de cómputo relativa* de cada procesador (pcr_i) respecto a la potencia del mejor de los procesadores de la arquitectura. La Ecuación 4.3 muestra este cálculo [Al-Jaroodi, et al., 2003].

$$pct = \sum_{i=0}^{p-1} pcr_i \tag{4.3}$$

De las Ecuaciones 4.2 y 4.3 se puede deducir que en las arquitecturas paralelas homogéneas el valor de *pct* equivale a *p*, debido a que todos los procesadores tienen la misma potencia de cómputo, y por lo tanto la potencia de cómputo relativa (pcr_i) de cada uno de ellos es igual a 1.

Una vez definida la *potencia de cómputo total de una arquitectura paralela* se puede generalizar el concepto de speedup óptimo o teórico para que pueda ser utilizado tanto por las arquitecturas homogéneas como por las heterogéneas. Para esto se redefine S_{opt} como la potencia de cómputo total (*pct*) de la arquitectura paralela en relación al poder de cómputo del mejor de los procesadores de la misma [Tinetti, 2004].

Ejemplo 4.1. Si se cuenta con una arquitectura compuesta por 8 procesadores $p_0 ... p_7$ donde p_0 tiene la mayor potencia de cómputo, $p_1 ... p_4$ tienen un 75 % de la misma y $p_5 ... p_7$ un 50 % de la potencia de p_0, se tienen:

$$\begin{aligned} pcr_0 &= 1 \\ pcr_i &= 0{,}75 \qquad (1 \le i \le 4) \\ pcr_i &= 0{,}50 \qquad (5 \le i \le 7) \end{aligned}$$

$$pct = \sum_{i=0}^{7} pcr_i = 1 + 4 \times 0{,}75 + 3 \times 0{,}50 = 5{,}5$$

$$S_{opt} = pct = 5{,}5$$

De esta manera, el S_{opt} alcanzable en esta arquitectura sería ***5,5***.

4.2.3. Eficiencia

Solo en un sistema paralelo ideal con una arquitectura de p elementos de procesamiento idénticos se puede alcanzar un speedup igual a p. En general, en la práctica esto no es logrado porque los elementos de procesamiento no pueden estar el 100% del tiempo realizando cómputo esencial de la aplicación serie, porque parte de él lo utilizan para realizar comunicaciones y sincronizaciones con otros procesadores.

La ***eficiencia*** (E) es una medida de la fracción de tiempo promedio para la cual los elementos de procesamiento del sistema paralelo son usados útilmente (realizando cómputo esencial). Esta métrica se define como la relación entre el speedup logrado por el sistema paralelo y el número de procesadores de la arquitectura homogénea utilizada [Grama, et al., 2003]. La Ecuación 4.4 muestra esta función.

$$E = \frac{S}{p} \tag{4.4}$$

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 de cómputo total de la misma.

Este problema lleva a generalizar la definición de *eficiencia* (E) de manera que pueda ser utilizado como métrica de rendimiento en arquitecturas paralelas homogéneas y heterogéneas. La función se define como la relación entre el speedup logrado por el sistema paralelo y la

potencia de cómputo total (o lo que es lo mismo el speedup óptimo) de la arquitectura utilizada [Tinetti, 2004]. La Ecuación 4.5 muestra esta función generalizada.

$$E = \frac{S}{pct} \qquad o \qquad E = \frac{S}{S_{opt}} \tag{4.5}$$

Ejemplo 4.2. En el ejemplo 4.1, la eficiencia del sistema paralelo es:

$$E = \frac{S}{pct} = \frac{4}{5{,}5} = 0{,}727$$

En un sistema paralelo ideal el speedup logrado (S) y el óptimo (S_{opt}) son iguales, por lo que la eficiencia es igual a 1. En la práctica toma valores entre 0 y 1, dependiendo de la efectividad con la que se usen los elementos de procesamiento de la arquitectura paralela. Aquellos casos en los que se obtiene speedup superlineal son los únicos en los que el valor de la eficiencia supera este límite superior.

4.3. Desbalance de carga

Una de los principales motivos que decrementan el rendimiento de un Sistema Paralelo es el ocio producido en los procesadores. Esto ocurre principalmente cuando el trabajo a realizar no está distribuido equitativamente (en cuanto a tiempo de ejecución y no en cuanto a cantidad de trabajo) entre los diferentes elementos de procesamiento que forman la arquitectura paralela.

Una métrica importante para analizar el comportamiento en este aspecto de un sistema paralelo es el ***desbalance de carga*** (D) entre los procesadores que intervienen. Este valor se calcula como la diferencia del tiempo de cómputo dedicado a resolver ese problema por los diferentes elementos de procesamiento (T_i). La forma de calcular este valor se indica en la Ecuación 4.6 [Bohn & Lamont, 2002][Naiouf, 2004].

$$D = \frac{máximo_{i=0..g-1}(T_i) - mínimo_{i=0..g-1}(T_i)}{promedio_{i=0..g-1}(T_i)} \tag{4.6}$$

$$siendo\ T_i\ el\ tiempo\ de\ cómputo\ del\ proceso\ i$$

Ejemplo 4.3. Notar que en un caso como el del ejemplo 4.1, si se le da igual volumen de trabajo a los 8 procesadores, se obtiene:

$$\begin{aligned} T_0 &= t \\ T_i &= 1{,}33\,t \qquad (1 \le i \le 4) \\ T_i &= 2\,t \qquad (5 \le i \le 7) \end{aligned}$$

$$promedio_{i=0..g-1}(T_i) = \frac{1\,t + 4 \times 1{,}33\,t + 3 \times 2\,t}{8} = 1{,}54\,t$$

$$D = \frac{2\,t - 1\,t}{1{,}54\,t} = 0{,}649 = 64{,}9\,\%$$

4.4. Escalabilidad

En muchos casos las aplicaciones paralelas son diseñadas y testeadas para pequeños problemas y/o en un número reducido de elementos de procesamiento, a pesar de que se deban utilizar en problemas y arquitecturas mucho mayores. Mientras que el desarrollo y testeo de la solución paralela se simplifica al utilizar instancias reducidas del problema y de la arquitectura, el rendimiento es muy difícil de predecir en forma precisa basado en esos resultados. Esto se debe a que los algoritmos pueden tener distinto comportamiento (referido al rendimiento) para los diferentes tamaños de problema y dimensiones de la arquitectura utilizada.

Existen métricas que permiten extrapolar el rendimiento a arquitecturas y/o problemas de mayor tamaño; con ellas se evalúa la escalabilidad de los sistemas paralelos utilizando herramientas analíticas.

La *escalabilidad* de un sistema paralelo es una medida de la capacidad de incrementar el speedup en proporción al número de procesadores de la arquitectura, reflejando la habilidad de utilizar efectivamente el incremento de los recursos de procesamiento. Su análisis es necesario para elegir la mejor combinación de algoritmo y arquitectura para resolver un problema bajo distintas restricciones sobre el crecimiento del tamaño del mismo o de los recursos de procesamiento.

Es importante analizar el efecto de incrementar el tamaño de la arquitectura dejando fijo el del problema, y por otro lado incrementar el tamaño del problema dejando fijo el número de procesadores. Las Figuras 3.1 y 3.2 esquematizan estos comportamientos, junto con las siguientes observaciones:

- Para un determinado tamaño de problema, al aumentar la cantidad de elementos de procesamiento (recursos de cómputo) cae la eficiencia del sistema paralelo completo (Figura 3.1). Este es un fenómeno común a todos los sistemas paralelos.
- Para una determinada cantidad de elementos de procesamiento (recursos de cómputo), al incrementar el tamaño del problema la eficiencia sube (Figura 3.2). Este es un fenómeno que no se da en todos los sistemas paralelos.

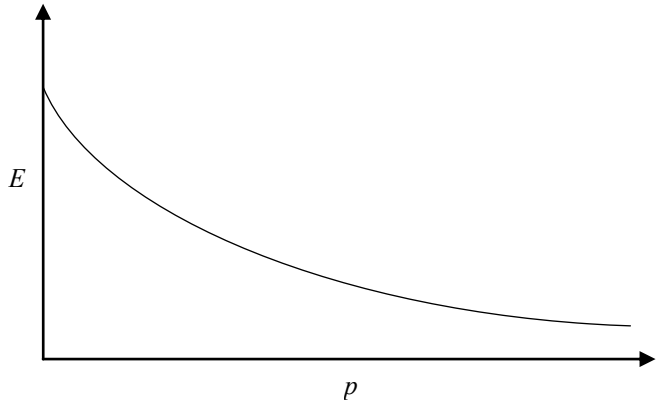


Figura 3.1. Variación de la eficiencia al aumentar los recursos de cómputo y dejar fijo el tamaño del problema.

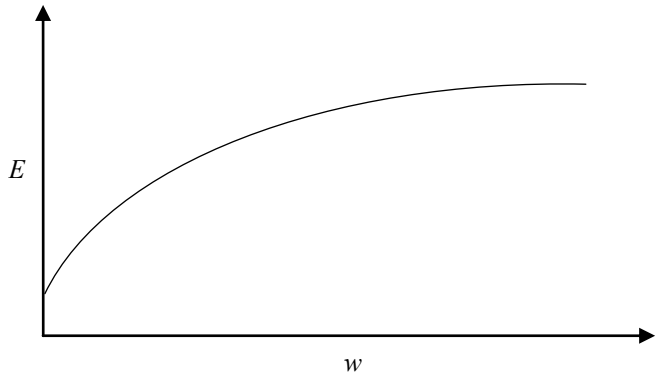


Figura 3.2. Variación de la eficiencia al aumentar el tamaño del problema y dejar fijos los recursos de cómputo.

En base a estas dos observaciones, se define como *sistema paralelo escalable* a aquel que puede mantener constante la eficiencia al incrementar el tamaño del problema y los recursos de cómputo (ambos al mismo tiempo). Es importante determinar el ritmo o velocidad al cual el problema debe crecer con respecto al número de elementos de procesamiento para conseguir mantener fija la eficiencia; éste es particular de cada sistema paralelo y es el que define el Grado de Escalabilidad del mismo. Un ritmo de crecimiento bajo implica un mayor aprovechamiento del crecimiento de la arquitectura, debido a que el overhead depende más del problema en sí que de la cantidad de procesadores.

Capítulo 5 - Problema "N-Reinas"

Para aprovechar al máximo las ventajas que proveen las arquitecturas paralelas, se deben diseñar algoritmos que se adapten a las características de las mismas. Para obtener su mejor rendimiento se debe lograr que todos los procesadores realicen una cantidad similar de trabajo, es decir, reducir el desbalance de carga entre los mismos [Dongarra, et al., 2003] [Leopold, 2001]. En este sentido es fundamental determinar la forma más adecuada de distribuir el trabajo entre los procesadores.

En este capítulo se analiza la performance de arquitecturas distribuidas heterogéneas, frente a una clase de aplicación en la que el trabajo depende de los datos y no puede conocerse "a priori". Se ha elegido el conocido problema de las *N*-reinas para analizar en detalle el rendimiento, considerando diferentes técnicas de asignación del trabajo a los procesadores.

Esta aplicación pertenece a una clase de problemas en los que el tiempo de comunicación entre procesos *Tc* no es significativo frente al tiempo de procesamiento local *Tp* (*Tp* >> *Tc*). Esta característica permite identificar más claramente las diferencias entre las soluciones de esquemas de balance de carga estático y dinámico, sin superponer un overhead importante de comunicaciones ajenas a la distribución.

El problema de *N*-reinas suele verse como puramente matemático y recreativo, aunque posee varias aplicaciones en el mundo real:

- Control de tráfico aéreo.
- Compresión de datos.
- Balance de carga y ruteo de mensajes o datos en un multiprocesador.
- Prevención de deadlocks.
- Administración de almacenamientos de memorias compartidas.
- Procesamiento de imágenes.
- Detección de movimientos en una red distribuida.

5.1. Descripción de la aplicación

El problema de las *8-reinas* es un clásico problema combinatorio que consiste en distribuir 8 reinas en un tablero de ajedrez clásico (8x8) de tal manera que no se ataquen entre ellas. Esta pieza de ajedrez tiene la capacidad de moverse en cualquier dirección (vertical,

horizontal y diagonal) la cantidad de casillas que desee, por lo que no es válido ubicar dos reinas en la misma fila, columna o diagonal.

El problema de las *N*-reinas es una generalización que consiste en ubicar *N* reinas en un tablero de *NxN* (para $N > 0$) [Bruen & Dixon, 1975][De Giusti, et al., 2003][Somers , 2002]. En la Figura 5.1 se muestran dos ejemplos de tableros para $N = 5$, uno válido (a) y otro inválido (b).

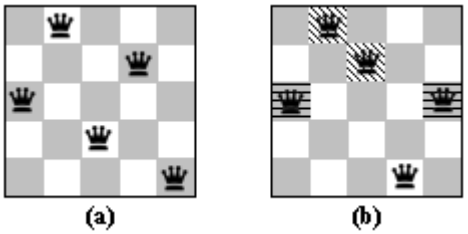


Figura 5.1. Ejemplo de tablero válido (a) e inválido (b).

Una solución inicial al problema de las N-reinas, mediante un algoritmo secuencial elemental (por fuerza bruta), consiste en probar todas las combinaciones posibles de ubicación de las reinas en el tablero y quedarse con aquéllas que son válidas.

La solución antes mencionada mejora su rendimiento interrumpiendo la búsqueda para una combinación en el momento en que se determina que la misma no es válida. Teniendo en cuenta que una combinación válida puede generar hasta 8 soluciones diferentes, las cuales son rotaciones/simetrías de la misma, se puede reducir la cantidad de distribuciones a evaluar [Takahashi, 2003].

En la Figura 5.2 se muestra un ejemplo de una combinación que produce 8 soluciones diferentes en un tablero de 5x5.

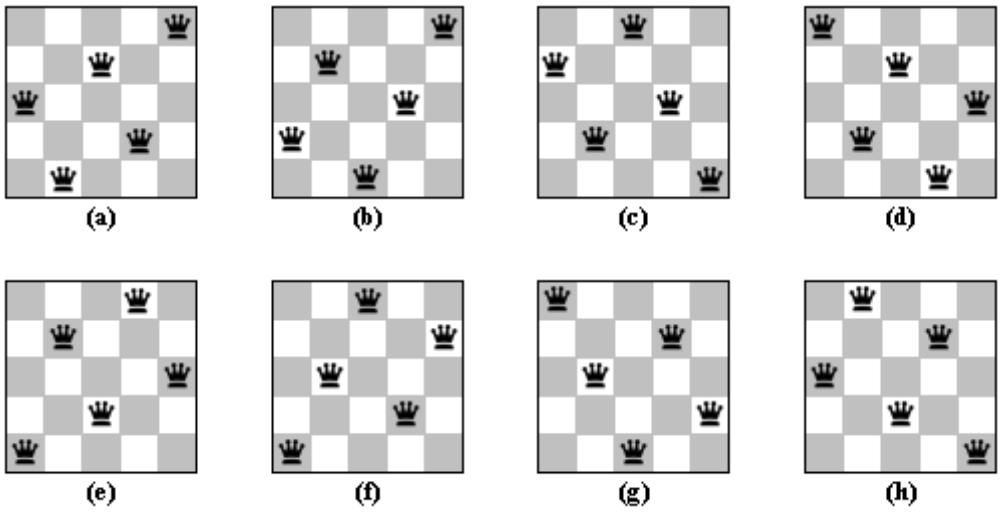


Figura 5.2. (a) y (b) combinación original y su simétrica, (c) y (d) rotación 90° y su simétrica, (e) y (f) rotación 180° y su simétrica, (g) y (h) rotación 270° y su simétrica.

Analizando el problema se puede ver que el costo en tiempo se incrementa en forma exponencial a medida que aumenta el tamaño del tablero (*N*), lo que hace complejo trabajar con tableros donde *N* tiende a valores altos.

Resulta claro que por sus características el problema de las N-reinas es de interés para analizar la performance de una arquitectura distribuida heterogénea: es escalable, requiere balanceo en función de los datos, tiene diferentes soluciones secuenciales/paralelas, y pueden diferenciarse claramente cómputo y comunicaciones [Dongarra, et al., 2003].

5.2. *Solución secuencial*

A continuación se realiza una breve descripción del mejor algoritmo secuencial para resolver el problema en un tablero de *N*x*N* [Bernhardsson, 1991][Takahashi, 2003].

El algoritmo realiza *N*/2 iteraciones (parte entera de división), y en cada una de ellas ubica la reina en una posición diferente de la primera fila. Las *N*/2 posiciones restantes no son analizadas debido a que son soluciones simétricas de la rotación a 90º de los casos evaluados.

```
main ()
{ cantSol:=0
  for  (pos= 1..N/2)
     ubicarReina (1,pos,tablero)
     detPosVálida (posVálida,tablero,2)
      cantSol:=cantSol + detSol (2,posVálida,tablero)
}
```

(a)

```
function cantidadSoluciones (t)
{ if (rot(t,90)=t) or (rot(t,90)=sim(t)) then return (2)
  else if (rot(t,180)=t) or (rot(t,180)=sim(t)) then return (4)
        else return (8)
}

rot(t,g): retorna el tablero t rotado en g grados.
sim(t): retorna el tablero simétrico de t.
```

(c)

```
function detSol (fila, posVálida, tablero)
{ i:= posiciónVálida (posVálida)
  if (fila = N) and (i ≤ N ) then
     ubicarReina (fila,i,tablero)
     return (cantidadSoluciones (tablero))
  else
     total:=0
     while (i ≤ N)
        ubicarReina (fila,i,tablero)
        detPosVálida (nuevaPosVálida,tablero, fila+1).
        total:= total + detSol (fila+1,nuevaPosVáalida,tablero )
         i:= posiciónVálida (posVálida)
      return total
}

ubicarReina (f,p,t): ubica una reina en la posición p de la fila f
del tablero t.
detPosVálida (c,t,f): determina el conjunto c de posiciones
válidas para la fila f en el tablero t.
posiciónVálida (c): saca y retorna la primer posición válida dada
en  c, o retorna N si no hay ninguna.
```

(b)

Figura 5.3. Pseudocódigo de la solución secuencial.

A partir de la reina ubicada en la primera fila se determina el vector de posiciones válidas para la siguiente fila, y para cada una de ellas se calculan las soluciones que las mismas generan (Figura 5.3(a)). Para obtener la cantidad de soluciones a partir de la fila *i* (tal que toda fila *j* con $j \leq i$ tiene ubicada su reina), se establece el vector de posiciones válidas para la fila *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 hay más posiciones válidas en una cierta fila (Figura 5.3(b)). Al llegar a ubicar una reina en la última fila se calcula la cantidad de

soluciones diferentes que generan dicha combinación y su simétrica al ser rotadas 90º, 180º y 270º (Figura 5.3(c)).

5.3. Solución paralela

Para la solución paralela, se ubica la reina de una o más filas y se obtienen todas las soluciones para esa disposición inicial. Cada procesador se encarga de resolver el problema para un subconjunto de éstas, de manera tal que el sistema completo trabaje con todas las posibles combinaciones (cada una será una *tarea* en la descomposición del problema) de esas filas. Para esto deben tenerse en cuenta dos aspectos:

- La forma de construir las combinaciones.
- Cómo distribuir las combinaciones entre las máquinas.

5.3.1. Construcción de las combinaciones

La idea inicial es que para formar las combinaciones sólo se tenga en cuenta la primera fila del tablero (*N/2* posibilidades como se expresa en la sección 5.2). Cada proceso se encarga de encontrar todas las soluciones a partir de la reina ubicada en una posición de dicha fila. En la Figura 5.4 se muestra un ejemplo de cuáles son las tareas en un tablero de 5x5.

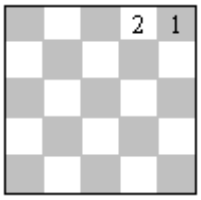


Figura 5.4. Distribución de la primera fila.

La cantidad de combinaciones a evaluar por el algoritmo se incrementa a medida que la reina ubicada en la primera fila se aproxima al centro del tablero [De Giusti, et al., 2003]. Como se sabe, cuanto más balanceado sea el trabajo a realizar entre los distintos procesadores el tiempo del algoritmo se reduce, por lo tanto la solución anterior se descarta [Grama, et al., 2003][Watts & Taylor, 1998].

Al utilizar una arquitectura heterogénea la cantidad de trabajo (combinaciones o tareas) que debe resolver cada procesador varía según la relación que existe en cuanto a la potencia de cálculo. Para lograr distribuir las tareas en forma equilibrada, es conveniente usar 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 [Zhang & Yan, 1995]. Esto se puede lograr al usar más filas 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 heterogéneos, siendo *N* el tamaño del

tablero. En la Figura 5.5 se muestra un ejemplo en un tablero de 5x5, usando tres filas para formar las combinaciones iniciales (algunas de ellas inválidas).

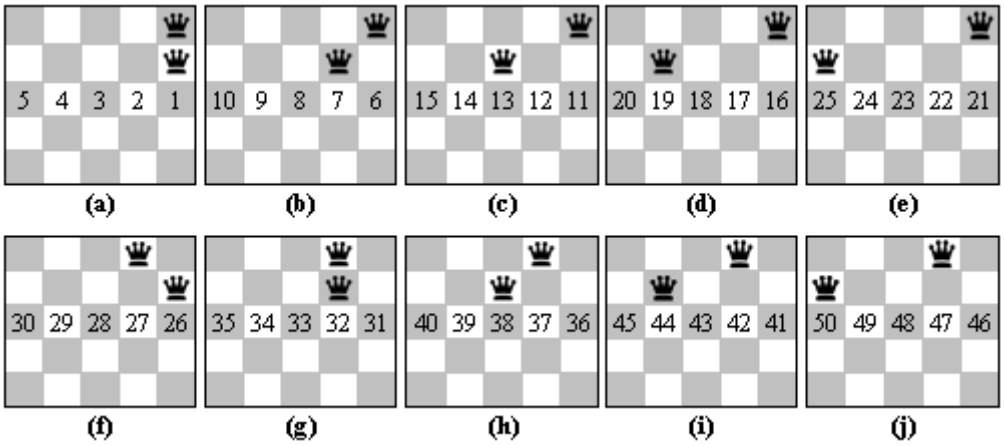


Figura 5.5. Combinaciones formadas a partir de las 3 primeras filas.

5.3.2. Distribución de las combinaciones

Para esta solución, en la cual se realiza una descomposición de grano fino en tareas relativamente independientes (sólo deben interactuar para unir los resultados), es adecuado utilizar el paradigma de programación paralela *Master/Worker*.

Existen diferentes maneras en que el *master* reparte el trabajo entre los *workers*, de acuerdo a si se tiene en cuenta la heterogeneidad de la arquitectura, y si se hace en forma estática o dinámica. En este trabajo se han implementado para comparar diferentes métodos de distribución: *Estática Directa (DEd)*, *Estática Predictiva (DEp)*, *Dinámica por Demanda (DDd)*.

En el resto de este capítulo se realiza una breve descripción de los tres métodos de distribución de trabajo mencionados.

5.3.2.1. Distribución Estática Directa (DEd)

En este tipo de distribución, las W combinaciones que se obtienen como se indica en la sección 5.3.1, se reparten de forma cíclica entre los p procesadores *workers* que forman la arquitectura. De esta manera, cada procesador p_i resuelve W / p combinaciones.

En este caso no se tiene en cuenta la posible heterogeneidad entre las máquinas, y no se requiere un proceso *master* que realice la distribución, debido a que los mismos procesos pueden determinar qué combinaciones deben resolver. En este caso la función del *master* se reduce a recopilar los resultados finales de los otros procesos, por lo tanto (para aprovechar su tiempo ocioso) actúa como un *worker* más, con la única diferencia que al final recibe los resultados de los otros procesos para acumularlos y obtener así la cantidad de soluciones total.

Ejemplo 5.1. Si se cuenta con una arquitectura compuesta por 8 procesadores heterogéneos $p_0 ... p_7$ donde la potencia de cómputo relativa de cada uno es:

$$pcr_0 = pcr_1 = pcr_2 = 1$$
$$pcr_3 = pcr_4 = pcr_5 = 0{,}66$$
$$pcr_6 = pcr_7 = 0{,}33$$

Y se trabaja con una instancia del problema tal que:

$$\left.\begin{array}{l} N = 6 \\ f = 3 \end{array}\right\} \; W = 108$$

Con la Distribución Estática Directa (*DEd*) las combinaciones se asignan de la siguiente manera:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	105	106	107	108
p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	...	p_0	p_1	p_2	p_3

5.3.2.2. Distribución Estática Predictiva (DEp)

En este caso, la distribución se realiza teniendo en cuenta la posible heterogeneidad de la arquitectura, basándose en la potencia de cómputo de los procesadores.

Al igual que la técnica anterior (*DEd*), no es necesario utilizar un proceso *master* que realice la distribución, aunque es más compleja la manera en que los *workers* determinan las combinaciones que deben resolver. El *master* actúa de la misma manera que en la distribución *DEd*.

Para determinar cuáles son las combinaciones que debe resolver cada procesador p_i se realiza los siguientes pasos:

- Dadas W combinaciones a realizar por medio de p procesadores, se obtiene la cantidad relativa de combinaciones para cada uno (cr_i , $\forall\, i = 0..p\text{-}1$) por medio de la Ecuación 5.1.

$$cr_i = \frac{pcr_i}{pcr_{MaquinaMenosPotente}} \tag{5.1}$$

- Se determina la cantidad B de combinaciones consecutivas que forman un bloque por medio de la Ecuación 5.2. El uso de bloques permite una distribución más equilibrada de las combinaciones.

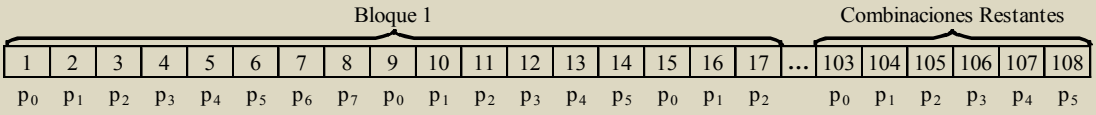
$$B = \sum_{i=0}^{p-1} cr_i \tag{5.2}$$

- En cada bloque, se distribuyen las combinaciones a realizar considerando que se asignan en forma cíclica (este paso puede tener variantes en la forma de distribuir las tareas de cada bloque entre los procesos) a cada procesador p_i que no haya realizado cr_i combinaciones en ese bloque.

Ejemplo 5.2. Para la arquitectura del Ejemplo 5.1, la cantidad relativa de combinaciones para cada procesador es:

$$\left.\begin{array}{l} cr_0 = cr_1 = cr_2 = \dfrac{1}{0{,}33} \cong 3 \\ cr_3 = cr_4 = cr_5 = \dfrac{0{,}66}{0{,}33} = 2 \\ cr_6 = cr_7 = \dfrac{0{,}33}{0{,}33} = 1 \end{array}\right\} \quad B = 17$$

En la instancia del problema del Ejemplo 5.1, con la Distribución Estática Predictiva (*DEp*) las combinaciones se asignan de la siguiente manera:



En la Figura 5.6 se muestra el pseudocódigo del algoritmo para ambas distribuciones estáticas (*DEd* y *DEp*); la diferencia radica en la forma en que los procesos determinan cuáles son las combinaciones que deben resolver. En este pseudocódigo se usan tres filas del tablero para formar las combinaciones iniciales a distribuir.

```
main () //Proceso p_id , id = 0..p-1
{ cantSol := 0
  while (p_id tenga combinaciones)
        determina la ubicación en la fila 1 (q1)
        determina la ubicación en la fila 2 (q2)
        determina la ubicación en la fila 3 (q3)
        cantSol := cantSol + detSolParcial (q1,q2,q3,tablero)
  if (id > 0)
       send (cantSol, 0)
  else
       for (i=1..p-1)
            recv (cantOtroProc, i);
            cantSol := cantSol + cantOtroProc
}
```

(a)

```
function detSolParcial (posFila1, posFila2, posFila3, tablero)
{ ubicarReina (1, posFila1, tablero)
  ubicarReina (2, posFila2, tablero)
  ubicarReina (3, posFila3, tablero)
  detPosVálida (posVálida, tablero, 4).
  return detSol (4, posVálida, tablero)
}
```

ubicarReina (f,p,t): ubica una reina en la posición *p* de la fila *f* del tablero *t*.
detPosVálida (c,t,f): determina el conjunto *c* de posiciones válidas para la fila *f* en el tablero *t*.

(b)

Figura 5.6. Pseudocódigo para *DEd* y *DEp*. La función *detSol* es la descripta en la Figura 5.3 (b).

5.3.2.3. Distribución Dinámica por Demanda (DDd)

En este caso se requiere un proceso *master* que -a diferencia de las distribuciones estáticas- sólo se encarga de distribuir las combinaciones y recibir los resultados, es decir que no actúa también como *worker*. En esta solución la cantidad total de procesos es $p+1$.

Las W combinaciones son distribuidas por el *master* entre los p procesos *workers* en dos etapas: primero un porcentaje Li de ellas se reparte con una *Distribución Estática Predictiva*; y luego las combinaciones restantes se distribuyen bajo demanda a medida que los *workers* solicitan más trabajo. Para esto el proceso *master* realiza los siguientes pasos:

- Reparte las combinaciones iniciales:
  - Calcula la cantidad de combinaciones CI a repartir inicialmente por medio de la Ecuación 5.3.

$$CI = \frac{W \times Li}{100} \tag{5.3}$$

  - Obtiene la cantidad de combinaciones iniciales ci_i que le corresponden al proceso *worker* p_i según pcr_i como se indica en las Ecuaciones 5.4 y 5.5.

$$pct_{workers} = \sum_{i=1}^{p} pcr_i \tag{5.4}$$

$$ci_i = \frac{CI \times pcr_i}{pct_{workers}} \tag{5.5}$$

  - Distribuye las CI combinaciones iniciales asignando ci_i combinaciones consecutivas a p_i, para $i = 1..p$.

- Actualiza el valor de *CI* por medio de la Ecuación 5.6.

$$CI = \sum_{i=1}^{p} ci_i \qquad (5.6)$$

- Reparte las combinaciones restantes:
  - Inicializa la cantidad de combinaciones restantes: $CR = W - CI$.
  - Inicializa la cantidad de combinaciones para cada solicitud: $CB = CB_{inicial}$.
  - Mientras $CR > 0$
    - Recibe una solicitud de trabajo de un proceso *worker* p_i que está libre (ha terminado el trabajo que se le ha asignado).
    - Envía *CB* (*CR* si $CR < CB$) combinaciones consecutivas al procesador p_i que realizó la solicitud.
    - Actualiza el valor de *CR*: $CR = CR - CB$.
    - Si $CB > 1$, actualiza su valor: $CB = CB - 1$.
  - Informa a todos los procesos *worker* que no hay más trabajo para realizar.

En la Figura 5.7 se presenta el pseudocódigo del algoritmo de la *Distribución Dinámica por Demanda* (*DDd*). En este pseudocódigo se usan tres filas del tablero para formar las combinaciones iniciales a distribuir.

```
main  ()  //Proceso master p0
{ cantSol := 0
  for (i= 1..p) //Reparto inicial
        determina la primera combinación (pri)
        determina la última combinación (ult)
        send (pri, ult, i)

 while (hay combinaciones) //Reparto restante
       recv (pedido, x)
        determina la primera combinación (pri)
        determina la última combinación (ult)
        send (pri, ult, x)
   for (i= 1..p)
        send (FIN, FIN, i)
   for (i= 1..p) //Recepción de los resultados
       recv (cantOtroProc, i)
        cantSol := cantSol + cantOtroProc
}
```

(a)

```
main ()  //Proceso worker pid , id = 1..p
{ cantSol := 0
  recv (pri, ult, 0)
  while (pri <> ult)
      for (combinacion = pri..ult)
          determina la ubicación en la fila 1 (q1)
          determina la ubicación en la fila 2 (q2)
          determina la ubicación en la fila 3 (q3)
          cantSol := cantSol + detSolParcial (q1,q2,q3,tablero)
      send (pedido, 0)
      recv (pri, ult, 0)
  send (cantSol, 0)
}
```

(b)

Figura 5.7. Pseudocódigo para *DDd*. La función *detSolParcial* es la descripta en la Figura 5.3 (b).

Para lograr el mejor rendimiento de la arquitectura es importante definir en forma correcta los valores de *Li* y de $CB_{inicial}$ ya que:

- A medida que *Li* se acerca al 100 %, la distribución se aproxima a la realizada en *DEp*. Esto produce un mayor desbalance de trabajo.

- A medida que *Li* se acerca al 0 %, la distribución se torna totalmente dinámica. Esto logra un mejor balance de trabajo, pero sufre una pérdida de rendimiento por el incremento de comunicaciones y el tiempo ocioso de los procesadores mientras esperan recibir más trabajo.
- Al igual que el parámetro *Li,* si $CB_{inicial}$ crece, se reduce la cantidad de interacciones entre el *master* y los *workers*, pero se incrementa el desbalance; por otra parte, si decrece, aumentan las interacciones pero se reduce el desbalance.

Ejemplo 5.3. Si se cuenta con una arquitectura compuesta por 9 procesadores heterogéneos $p_0..p_8$ donde la potencia de cómputo relativa de cada uno es:

$$pcr_0 = 0{,}25 \quad \{Master\}$$
$$pcr_1 = pcr_2 = pcr_3 = 1$$
$$pcr_4 = pcr_5 = pcr_6 = 0{,}66$$
$$pcr_7 = pcr_8 = 0{,}33$$

Se resuelve la instancia del problema del ejemplo 5.1, con *Li = 80* % y $CB_{inicial} = 5$.

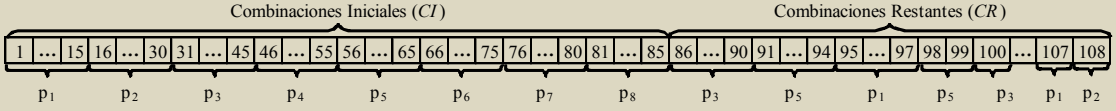
- Inicialmente $CI \cong 86$ $\qquad pct_{workers} = 5.64$

- La cantidad de combinaciones iniciales para cada proceso son:

$$\left.\begin{array}{l} ci_1 = ci_2 = ci_3 = \dfrac{86 \times 1}{5{,}64} = 15{,}25 \cong 15 \\ ci_4 = ci_5 = ci_6 = \dfrac{86 \times 0{,}66}{5{,}64} = 10{,}06 \cong 10 \\ ci_7 = ci_8 = \dfrac{86 \times 0{,}33}{5{,}64} = 5{,}20 \cong 5 \end{array}\right\} \; Total = 85$$

- Se actualiza *CI = 85;* $\qquad CR = 108 - 85 = 23;$ $\qquad CB = 5$

Con la Distribución Dinámica por Demanda (*DDd*) las combinaciones se asignan de la siguiente manera:



Lista de requerimientos

p3 - p5 - p1 - p5 - p3 - ... - p1 - p2

Capítulo 6 – Experimentación

En esta sección se describen las pruebas y resultados obtenidos para analizar el comportamiento de las diferentes implementaciones presentadas. En primera instancia se comparan las tres soluciones (DEd, DEp y DDd) utilizando el cluster completo para determinar cuál de las distribuciones se adapta mejor al problema y la arquitectura.

Una vez determinada la mejor solución al problema (mejor distribución de trabajo) se analiza el comportamiento de la misma usando diferentes tamaños de la arquitectura (distintas combinaciones de máquinas de la arquitectura completa).

6.1. *Hardware utilizado*

La experimentación se realizó sobre una arquitectura de cluster heterogéneo compuesta por cuatro grupos de máquinas:

- *Cluster 1*. Compuesto por 17 Pentium IV 2.4 Ghz, de 1 Gb de memoria.
- *Cluster 2*. Compuesto por 10 Celeron 2 Ghz, de 128 Mb de memoria.
- *Cluster 3*. Compuesto por 8 Duron 800Mhz con 256 Mb de memoria.
- *Cluster 4*. Compuesto por 8 Pentium III 700 Mhz, 256 Mb de memoria.

La comunicación dentro de cada grupo se realiza por medio de una red Ethernet de 100 Mbit, y se utiliza un switch para la comunicación entre ellos.

6.2. *Comparación de las tres soluciones*

El lenguaje utilizado para las implementaciones es C junto con la librería MPI (*LamMPI*) para manejar las comunicaciones entre procesos [Moura e Silva & Buyya, 1999].

Para llevar a cabo las pruebas, primero se debe determinar la potencia de cómputo relativa de cada tipo de procesador para esta clase de aplicación en particular. Este valor se obtiene ejecutando el algoritmo secuencial (con tableros relativamente chicos, $N < 20$) en una máquina de cada tipo. En la Tabla 6.1 se muestran los resultados de estas pruebas, y en la Tabla 6.2 la potencia de cómputo relativa de cada tipo de máquina respecto a la más potente (las pertenecientes al *Cluster 1*).

Tamaño del Tablero	Cluster 1	Cluster 2	Cluster 3	Cluster 4
17 x 17	39,71	47,35	82,68	110,35
18 x 18	288,067	341,88	600,84	800,98
19 x 19	2218,45	2633,02	4619,25	6153,77

Tabla 6.1. Tiempos de ejecución secuencial (en segundos) en una máquina de cada cluster.

Tipo de máquina	*pcr*
Cluster 1	1
Cluster 2	0,84
Cluster 3	0,48
Cluster 4	0,36

Tabla 6.2. Potencia de cómputo relativa de cada tipo de procesador.

Las pruebas se realizaron utilizando las 43 máquinas del hardware mencionado en la Sección 6.1, siendo *pct = 32,12*.

En todas las pruebas se utilizan las cuatro primeras filas del tablero para formar las combinaciones iniciales (tareas en que se descompone la aplicación). Se probaron tableros de diferente tamaño (*N* = 17; 18; 19; 20 y 21). En el caso de la distribución dinámica, se probó con diferentes porcentajes de reparto inicial (*Li* = 0, 5, 10, 15, 20 y 25), y con el valor $CB_{inicial}=50$ en todos los casos.

En la Tabla 6.3 se muestra el tiempo paralelo (medido en segundos) de las pruebas realizadas y el tiempo secuencial para cada tamaño de tablero, y en las Tablas 6.4 y 6.5 se muestra el speedup y la eficiencia, respectivamente.

Tamaño del Tablero	Secuencial	*DEd*	*DEp*	*DDd*					
				0%	*5%*	*10%*	*15%*	*20%*	*25%*
17 x 17	39,71	4,70	2,81	1,65	1,61	1,59	1,58	1,56	1,70
18 x 18	288,07	27,91	20,94	9,58	9,55	9,54	9,52	9,67	12,00
19 x 19	2218,45	228,16	152,95	72,06	72,53	72,11	71,80	73,66	77,50
20 x 20	17757,66	1795,20	1399,46	575,57	574,81	574,41	573,80	572,99	713,12
21 x 21	149393,74	13957,76	9554,65	4822,85	4823,02	4820,22	4819,90	4819,99	5513,25

Tabla 6.3. Tiempo paralelo y secuencial de las pruebas realizadas en esta etapa.

Tamaño del Tablero	*DEd*	*DEp*	*DDd*					
			0%	*5%*	*10%*	*15%*	*20%*	*25%*
17 x 17	8,45	14,13	24,13	24,66	24,96	25,13	25,39	23,30
18 x 18	10,32	13,76	30,07	30,15	30,20	30,25	29,79	24,00
19 x 19	9,72	14,50	30,79	30,59	30,76	30,90	30,12	28,63
20 x 20	9,89	12,69	30,85	30,89	30,91	30,95	30,99	24,90
21 x 21	10,70	15,64	30,98	30,98	30,99	31,00	30,99	27,10

Tabla 6.4. Speedup alcanzado en las pruebas realizadas en esta etapa.

Tamaño del Tablero	DEd	DEp	DDd					
			0%	5%	10%	15%	20%	25%
17 x 17	0,26	0,44	0,75	0,77	0,78	0,78	0,79	0,73
18 x 18	0,32	0,43	0,94	0,94	0,94	0,94	0,93	0,75
19 x 19	0,30	0,45	0,96	0,95	0,96	0,96	0,94	0,89
20 x 20	0,31	0,40	0,96	0,96	0,96	0,96	0,96	0,78
21 x 21	0,33	0,49	0,96	0,96	0,96	0,96	0,96	0,84

Tabla 6.5. Eficiencia de las pruebas realizadas en esta etapa.

En la Figura 6.1 se muestra en forma gráfica la eficiencia lograda para los diferentes tamaños de tablero con las distintas soluciones. Por razones de legibilidad, para la solución *DDd* se utiliza la eficiencia lograda con *Li = 15*.

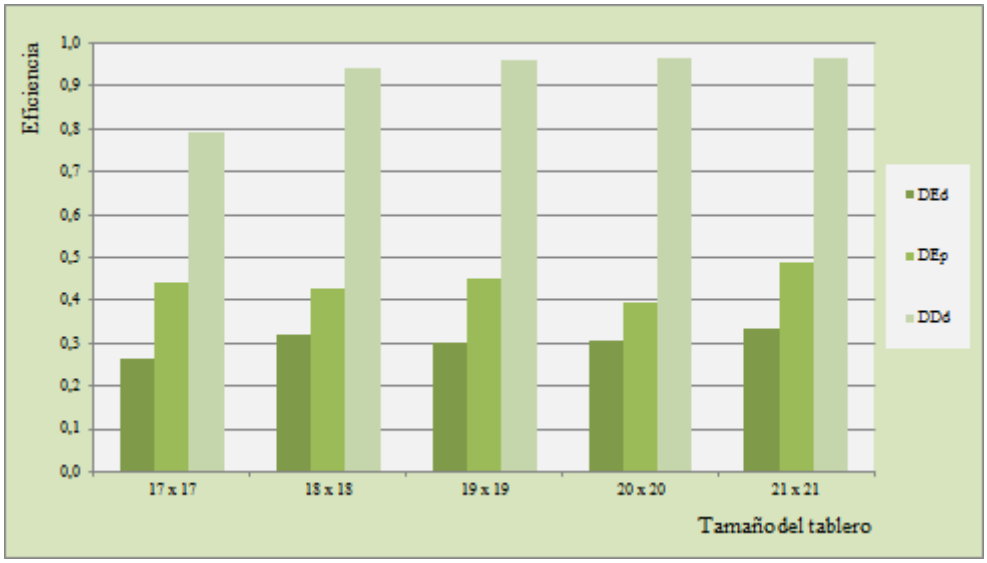


Figura 6.1. Eficiencia lograda para los distintos tamaño de tablero y soluciones (en el caso de *DDd*, con *Li=15*).

En este gráfico se puede ver claramente la superioridad (en cuanto a la eficiencia lograda) del algoritmo que distribuye el trabajo en forma dinámica respecto a los que lo hacen en forma estática, a pesar de requerir más comunicaciones y generar más ocio para interactuar entre el proceso *master* y los *workers*. Esto se debe al desbalance de carga producido al distribuir el trabajo estáticamente, como se puede observar en la Tabla 6.6, donde se muestra esta medida en las pruebas realizadas.

Tamaño del Tablero	DEd	DEp	DDd					
			0%	5%	10%	15%	20%	25%
17 x 17	236,09	122,37	9,23	7,57	8,10	7,60	9,98	10,91
18 x 18	148,30	115,20	2,58	2,06	2,08	2,23	3,02	3,77
19 x 19	160,66	107,62	0,40	0,39	0,36	0,45	0,60	0,66
20 x 20	162,02	128,21	0,05	0,09	0,07	0,08	0,11	0,11
21 x 21	145,60	92,91	0,03	0,03	0,03	0,03	0,03	0,05

Tabla 6.6. Desbalance de carga obtenido en las pruebas realizadas.

El desbalance generado en las distribuciones estáticas supera en varios órdenes de magnitud al del algoritmo con distribución dinámica. Entre los algoritmos *DEd* y *DEp* se puede observar una mejora importante al tener en cuenta la potencia de cómputo de las máquinas, pero no alcanza para aproximarse al *DDd*.

6.3. *Análisis de la escalabilidad de la solución DDd*

Una vez determinada la solución que mejor eficiencia logra (la que realiza la Distribución Dinámica por Demanda), se busca determinar el comportamiento de la misma para diferentes tamaños de arquitectura, analizando de esta manera la escalabilidad.

En este caso, se definen diferentes subconjuntos de máquinas del cluster completo, los cuales se detallan en la Tabla 6.7. En cada una de estas configuraciones se prueba el algoritmo *DDd* para tableros de distinto tamaño (N = 17; 18; 19; 20 y 21) y variando el porcentaje de reparto inicial (Li = 0, 5, 10, 15, 20 y 25).

pct	Cluster 1	Cluster 2	Cluster 3	Cluster 4
6,36	3	3	1	1
11,36	5	4	4	3
16,36	7	5	7	5
21,36	9	8	8	5
32,12	17	10	8	8

Tabla 6.7. Tamaños de la arquitectura usados, indicando la cantidad de máquinas de cada tipo.

En el Anexo I se encuentran detallados los resultados de todas las pruebas realizadas. En la Tabla 6.8 se exhibe un cuadro resumen con los tiempos de ejecución para los distintos tamaños de tablero en cada una de las arquitecturas usadas (detallados en la tabla 6.7). En las Tablas 6.9 y 6.10 se indica el speedup y la eficiencia respectivamente, obtenidas en las pruebas. Lo mismo se grafica en las Figuras 6.2 y 6.3. Para cada combinación *tamaño–arquitectura* se muestra el mejor tiempo logrado de los obtenidos con los diferentes valores de *Li*.

Tamaño del Tablero	Secuencial	*pct* = 6,36	*pct* = 11,36	*pct* = 16,36	*pct* = 21,36	*pct* = 32,12
17 x 17	39,71	8,83	4,51	3,09	2,39	1,56
18 x 18	288,07	56,73	29,25	19,74	14,95	9,52
19 x 19	2218,45	426,18	219,78	148,10	111,64	71,80
20 x 20	17757,66	3394,00	1753,61	1182,02	892,08	572,99
21 x 21	149393,74	28613,66	14754,08	9934,55	7498,34	4819,90

Tabla 6.8. Tiempo (en segundos) del algoritmo secuencial y paralelo (con diferentes *pct*) con los distintos tamaños de tablero.

Se puede observar el crecimiento exponencial del tiempo de respuesta al aumentar el tamaño del tablero, y la reducción proporcional al aumentar la potencia de cómputo de la arquitectura utilizada.

Tamaño del Tablero	*pct* = 6,36	*pct* = 11,36	*pct* = 16,36	*pct* = 21,36	*pct* = 32,12
17 x 17	4,50	8,81	12,84	16,59	25,38
18 x 18	5,08	9,85	14,59	19,26	30,25
19 x 19	5,21	10,09	14,98	19,87	30,90
20 x 20	5,23	10,13	15,02	19,91	30,99
21 x 21	5,22	10,13	15,04	19,92	31,00

Tabla 6.9. Speedup obtenido con los distintos tamaños de arquitectura y tablero.

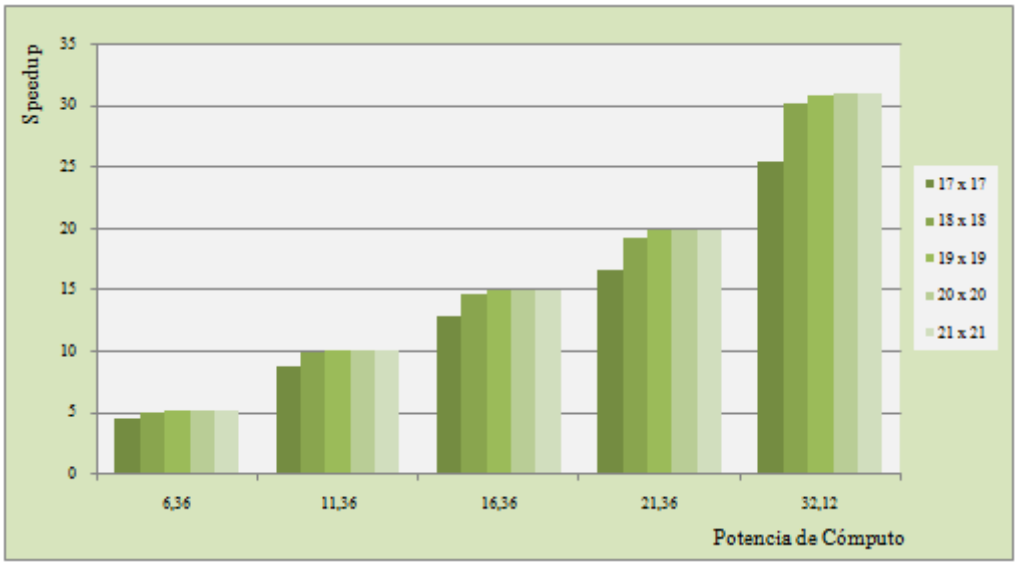


Figura 6.2. Speedup obtenido con los distintos tamaños de arquitectura y tablero.

Como es de esperar, el speedup aumenta de forma proporcional a la potencia de cómputo al cambiar la arquitectura sobre la que se realiza la prueba. Por otro lado, al aumentar el tamaño del tablero en una misma arquitectura, se observa un leve incremento del speedup, tendiendo a estabilizarse a partir de *N=19*.

Tamaño del Tablero	*pct* = 6,36	*pct* = 11,36	*pct* = 16,36	*pct* = 21,36	*pct* = 32,12
17 x 17	0,71	0,78	0,79	0,78	0,79
18 x 18	0,80	0,87	0,89	0,90	0,94
19 x 19	0,82	0,89	0,92	0,93	0,96
20 x 20	0,82	0,89	0,92	0,93	0,96
21 x 21	0,82	0,89	0,92	0,93	0,96

Tabla 6.10. Eficiencia obtenida con los distintos tamaños de arquitectura y tablero.

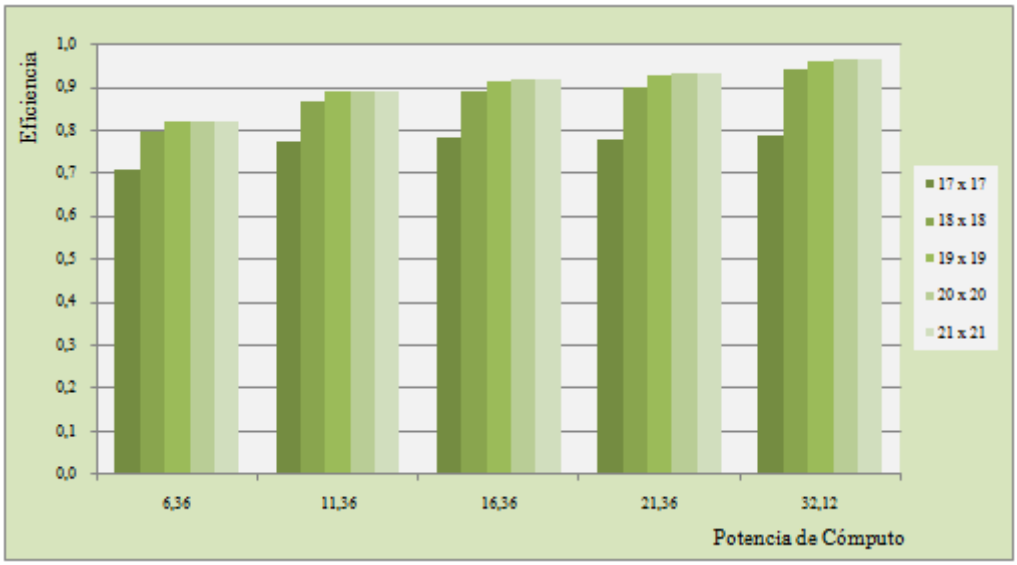


Figura 6.3. Eficiencia obtenida con los distintos tamaños de arquitectura y tablero.

Desde el punto de vista de la eficiencia se observa una incremento a medida que se aumenta el tamaño del problema, tendiendo a estabilizarse a partir de *N=19* como ocurre con el speedup. Esto se debe a que el *Componente Paralelo* del algoritmo crece a un ritmo mayor que el *Componente Serie* al aumentar el problema (tamaño del tablero). Esto suele caracterizar a los problemas escalables.

Por otro lado, también se logra un incremento en la eficiencia -menos pronunciado- al incrementar la potencia de cómputo de la arquitectura utilizada. Esta mejora tenderá a estabilizarse y posteriormente decaer cuando la arquitectura utilizada sea lo suficientemente grande como para generar excesivas colisiones entre los *worker* (al realizar los pedidos de trabajo) e incrementar el desbalance de carga entre ellos.

Capítulo 7 – Conclusiones y Trabajos Futuros

El crecimiento del procesamiento paralelo como una técnica para aumentar el aprovechamiento de sistemas de cómputo es innegable, principalmente debido a la aparición de arquitecturas distribuidas que son accesibles a cualquier clase de usuario. En particular, las redes de computadoras constituyen una plataforma de cómputo paralelo muy utilizada por sus ventajas en términos de la relación costo/rendimiento.

Este crecimiento lleva a un incremento de la investigación y desarrollo de éste 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 este Trabajo Final es comparar el efecto de la distribución de trabajo estática y dinámica sobre arquitecturas de cluster heterogéneo, analizando al mismo tiempo el speedup paralelo teórico y el obtenido experimentalmente para un determinado tipo de problema.

En particular, se ha elegido una aplicación clásica (Parallel N-Queens) con un algoritmo de solución paralela en la que predomina el procesamiento sobre el tamaño de los datos, de modo de profundizar en los aspectos del balance de carga (estático o dinámico) sin una distorsión de los resultados producida por aspectos relacionados al uso de la memoria y/o al tamaño de los mensajes a comunicar.

Para la experimentación se ha utilizado una combinación de 4 clusters interconectados, donde las máquinas dentro de cada grupo poseen procesadores homogéneos, pero diferentes entre clusters. De este modo el conjunto puede verse como un cluster heterogéneo de 43 procesadores.

El problema se ha resuelto utilizando el paradigma master/worker donde el procesamiento se descompone en tareas irregulares que atentan contra el balance de carga entre los procesadores. Por esta razón se han analizado tres estrategias de distribución de trabajo calculando en cada caso el desbalance de carga y el rendimiento obtenido, comparando los resultados para determinar la que tiene mejor comportamiento, y finalmente estudiar la escalabilidad para esa solución.

La solución paralela pura (sin tener en cuenta la distribución del trabajo) para el tipo de problemas donde $Tp >> Tc$, en particular el de N-Reinas requiere mínima comunicación entre máquinas, lo que hace esencial la elección de la distribución de datos entre los procesadores,

para alcanzar un speedup cercano al óptimo (es decir un buen rendimiento). De las tres opciones estudiadas tenemos que:

- En la *Distribución Estática Directa (DEd)*:
  - el overhead producido por las interacciones entre procesos es mínima ya que cada worker sólo tiene una comunicación (para devolver la cantidad de soluciones encontradas en su parte de trabajo).
  - se puede aprovechar al proceso master para que realice trabajo como un worker más, y al final recolecte el resultado de todos los demás.
  - no requiere tiempo extra para calcular cuántas y cuáles combinaciones debe resolver cada worker.
- En la *Distribución Estática Predictiva (DEp)*:
  - el overhead producido por las interacciones entre procesos es mínima ya que cada worker sólo tiene una comunicación (para devolver la cantidad de soluciones encontradas en su parte de trabajo).
  - se puede aprovechar al proceso master para que realice trabajo como un worker más, y al final recolecte el resultado de todos los demás.
  - requiere tiempo extra para calcular cuantas y cuales combinaciones debe resolver cada worker.
- En la *Distribución Dinámica bajo Demanda (DDd)*:
  - tiene un overhead importante producido por las interacciones entre procesos dado que cada vez que un worker requiere más trabajo debe pedirlo al master, lo que implica (en cada pedido de trabajo) dos comunicaciones. Obviamente este overhead varía dependiendo del porcentaje de reparto inicial *Li* y el tamaño del tablero *N***.**
  - no se puede aprovechar al proceso master para que realice trabajo como un worker más porque en todo momento debe estar resolviendo demandas de trabajo.
  - en el master se produce un cuello de botella al haber muchas demandas simultáneas de trabajo, lo cual se incrementa al aumentar la cantidad de procesos worker.

De los resultados obtenidos en la primera fase de pruebas (comparación del rendimiento de las tres soluciones) se puede observar que:

- Naturalmente los algoritmos que tienen en cuenta la potencia de cálculo de cada máquina para la distribución del trabajo se comportan mejor que la distribución Estática Directa. Esta mejora se expresa claramente en el balance de carga y speedup.

- Entre los algoritmos que tienen en cuenta la potencia de cálculo, se puede observar que aquel que distribuye en forma dinámica logra repartir el trabajo de manera más balanceada entre todas las máquinas, sin afectar demasiado el tiempo final de la ejecución.
- En la distribución dinámica el speedup obtenido es muy cercano al óptimo de acuerdo a la arquitectura paralela que se utiliza en este caso, lo cual es más notorio a medida que aumenta el tamaño del problema (en este caso del tablero).

De esta manera se puede concluir que la solución paralela que realiza la Distribución Dinámica por Demanda (*DDd*) logra el mejor rendimiento. Esto se debe a que la importante reducción en el desbalance de carga produce una disminución en el tiempo final de ejecución mucho mayor al overhead que genera: el incremento de las interacciones, la centralización de solicitudes (cuello de botella) producida en el master y el no poder usar al master para que realice el trabajo de un worker (tener un proceso menos que trabaje).

La escalabilidad de un sistema paralelo es otro aspecto 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.

Una vez determinada la superioridad de la solución *DDd* respecto a las que realizan la distribución en forma estática, se analizó la escalabilidad de la misma utilizando diferentes tamaños de arquitectura con los distintos tamaños de tablero.

Desde el punto de vista de la eficiencia se observa una incremento a medida que se aumenta el tamaño del problema, tendiendo a estabilizarse a partir de *N=19* como ocurre con el speedup. Esto se debe a que el *Componente Paralelo* del algoritmo crece a un ritmo mayor que el *Componente Serie* al aumentar el problema (tamaño del tablero).

Por otro lado, también se logra un incremento en la eficiencia -menos pronunciado- al incrementar la potencia de cómputo de la arquitectura utilizada. Este resultado se debe a que:

- El *master* sólo se encarga de atender las demandas de trabajo, por lo que la mayor parte del tiempo el procesador está ocioso.
- Los *workers* están ociosos mientras esperan que el *master* atienda la petición de trabajo, o cuando no hay más trabajo por hacer y están esperando que el resto termine. Al realizar una distribución del trabajo balanceada, el tiempo ocioso de cada *worker* es chico y proporcional a la cantidad de pedidos de trabajo realizados.

- Al incrementar la potencia de cómputo de la arquitectura, se reduce el tiempo final de la aplicación, disminuyendo el tiempo ocioso del *master*. Por otro lado, suponiendo un sistema ideal en el cual no hay colisiones por pedidos simultáneos de trabajo, la proporción de tiempo perdido por el conjunto de *workers* tiende a mantenerse.

Esta mejora tenderá a estabilizarse y posteriormente decaer cuando la arquitectura utilizada sea lo suficientemente grande como para generar excesivas colisiones entre los *worker* (al realizar los pedidos de trabajo) e incrementar el desbalance de carga entre ellos.

Como trabajo futuro se pretende analizar el comportamiento de este tipo de problemas/soluciones en arquitecturas de mayor tamaño, en particular en un cluster heterogéneo de multicores.

Al contar en esta arquitectura con mayor cantidad de elementos de procesamiento (cores totales), el cuello de botella generado en el master será más notorio, por lo que se debe realizar una solución con dos niveles de master (uno general y uno local en cada nodo) para comparar con la actual y analizar a partir de qué momento (tamaño de problema y cantidad de procesos worker) hay que utilizarla.

La solución con dos niveles de master permite al mismo tiempo comparar una solución pura de pasaje de mensajes con una híbrida. En esta última, la comunicación entre el master general y cada nodo del cluster es por pasaje de mensajes, mientras que dentro del nodo los procesos siguen el modelo *bag of task* a través dela memoria compartida del multicore. De esta manera se evita perder un core por nodo para ejecutar el master local.

Anexo I- Resultados Completos

En ete anexo se detallan los resultados de las pruebas realizadas para la *DDd*. Se agregan los resultados de: *desbalance de cómputo*, *tiempo de ejecución*, *speedup* y *eficiencia.*

AI.1. Desbalance de cómputo

La Tabla AI.1 muestra el desbalance de cómputo de las pruebas realizadas. Las Figuras AI.1, AI.2, AI.3, AI.4 y AI.5 muestran en forma gráfica estos mismos datos separados por tamaño de tablero (*N*).

Tamaño	Li	*pct* = 6,36	*pct* = 11,36	*pct* = 16,36	*pct* = 21,36	*pct* = 32,12
17 x 17	0%	3,182	7,147	6,074	6,958	9,226
	5%	3,573	8,662	3,662	6,886	7,570
	10%	6,971	3,578	4,713	5,046	8,100
	15%	6,988	3,548	8,565	8,837	7,602
	20%	6,114	4,970	4,810	5,660	9,979
	25%	5,249	4,872	5,843	7,545	10,914
18 x 18	0%	1,435	0,947	0,813	1,943	2,582
	5%	0,680	1,034	0,799	1,074	2,059
	10%	0,905	1,213	1,292	1,236	2,083
	15%	1,111	1,023	1,043	1,689	2,231
	20%	0,865	1,132	1,222	1,063	3,022
	25%	1,249	1,319	1,624	1,528	3,765
19 x 19	0%	0,215	0,271	0,250	0,303	0,399
	5%	0,179	0,284	0,261	0,238	0,394
	10%	0,335	0,366	0,313	0,292	0,363
	15%	0,191	0,214	0,313	0,249	0,447
	20%	0,205	0,278	0,295	0,366	0,598
	25%	0,262	0,299	0,262	0,445	0,662
20 x 20	0%	0,065	0,062	0,044	0,072	0,054
	5%	0,055	0,080	0,046	0,071	0,088
	10%	0,066	0,080	0,048	0,091	0,073
	15%	0,071	0,069	0,058	0,082	0,075
	20%	0,081	0,093	0,068	0,087	0,105
	25%	0,067	0,085	0,060	0,096	0,110
21 x 21	0%	0,034	0,030	0,027	0,031	0,029
	5%	0,049	0,050	0,027	0,030	0,028
	10%	0,029	0,034	0,028	0,030	0,028
	15%	0,029	0,027	0,030	0,028	0,030
	20%	0,038	0,027	0,031	0,035	0,027
	25%	0,028	0,029	0,063	0,053	0,045

Tabla AI.1. Desbalance de carga de todas las pruebas.

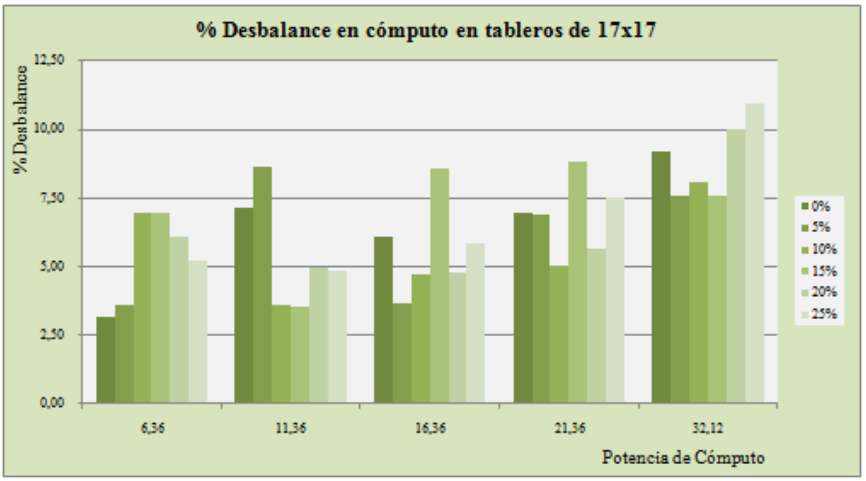


Figura AI.1. Desbalance de carga de las pruebas con tablero de 17x17.

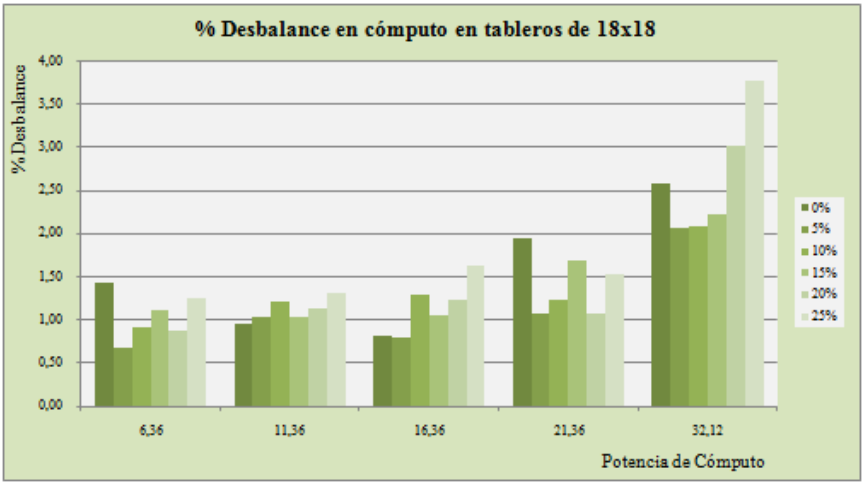


Figura AI.2. Desbalance de carga de las pruebas con tablero de 18x18.

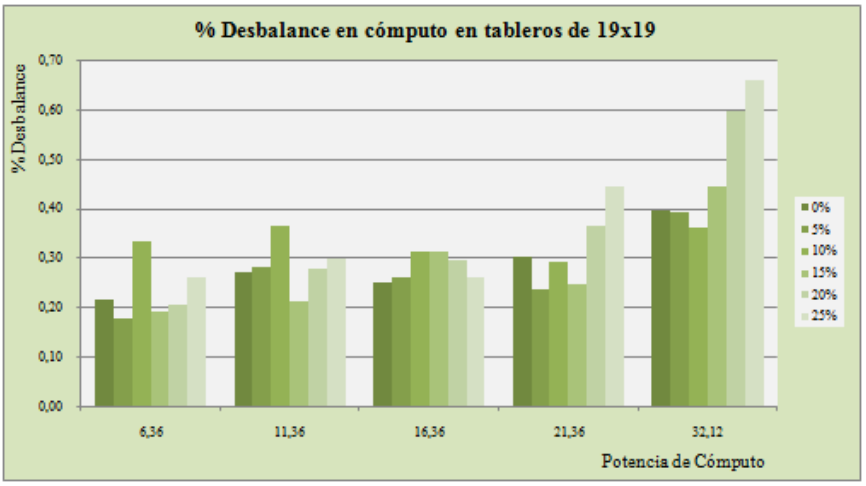


Figura AI.3. Desbalance de carga de las pruebas con tablero de 19x19.

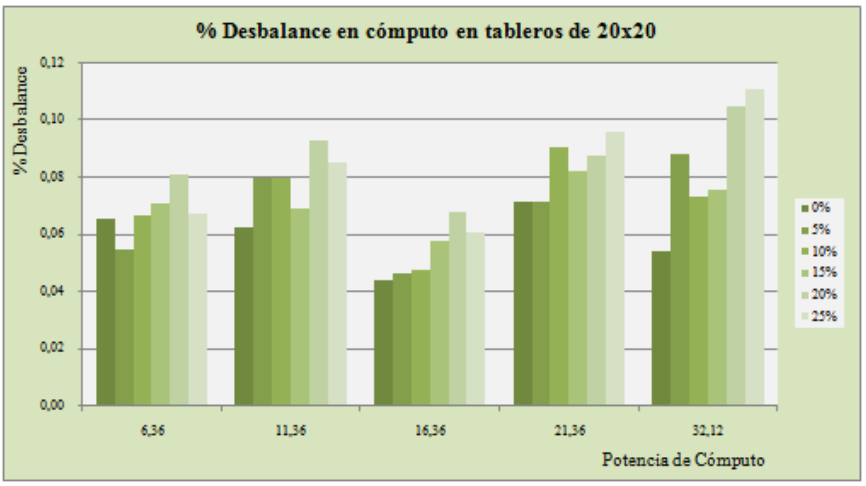


Figura AI.4. Desbalance de carga de las pruebas con tablero de 20x20.

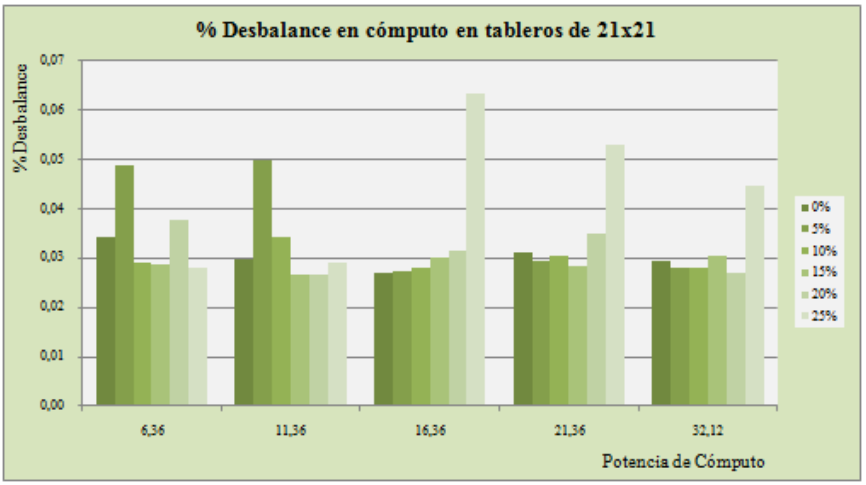


Figura AI.5. Desbalance de carga de las pruebas con tablero de 21x21.

AI.2. Tiempo de Ejecución

La Tabla AI.2 muestra el tiempo de ejecución final de las pruebas realizadas con los siguientes parámetros:

- N = 17; 18; 19; 20 y 21.
- *pct* = 6,36; 11,36; 16,36; 21,36 y 32,36.
- Li= 0; 5; 10; 15; 20; y 25.

Las Figuras AI.6, AI.7, AI.8, AI.9 y AI.10 muestran en forma gráfica estos mismos datos separados por tamaño de tablero (N).

Tamaño	Li	*pct* = 6,36	*pct* = 11,36	*pct* = 16,36	*pct* = 21,36	*pct* = 32,12
17	0%	9,161	4,690	3,263	2,513	1,646
	5%	8,984	4,735	3,193	2,642	1,607
	10%	8,992	4,596	3,316	2,454	1,591
	15%	9,002	4,613	3,147	2,436	1,580
	20%	8,840	4,529	3,128	2,417	1,564
	25%	8,827	4,506	3,092	2,394	1,704
18	0%	57,107	29,507	20,000	15,129	9,581
	5%	57,204	29,482	19,859	15,026	9,553
	10%	56,944	29,375	19,901	15,003	9,540
	15%	57,864	29,314	19,923	15,109	9,524
	20%	56,806	29,451	20,128	14,954	9,669
	25%	56,729	29,247	19,738	14,960	12,003
19	0%	426,497	219,861	150,808	113,199	72,055
	5%	429,663	221,065	149,845	111,681	72,529
	10%	427,589	219,877	149,090	111,954	72,112
	15%	433,508	220,598	149,591	111,714	71,803
	20%	426,501	222,122	149,555	112,323	73,660
	25%	426,182	219,785	148,103	111,639	77,495
20	0%	3405,954	1774,845	1183,118	892,079	575,573
	5%	3401,141	1755,911	1192,632	897,887	574,810
	10%	3406,095	1753,954	1193,879	893,246	574,405
	15%	3394,004	1787,083	1182,852	895,313	573,798
	20%	3451,736	1753,614	1182,018	898,215	572,992
	25%	3395,063	1762,427	1190,493	904,204	713,124
21	0%	28849,749	15011,524	9934,545	7513,581	4822,848
	5%	30508,856	14754,079	9935,288	7514,579	4823,020
	10%	28686,673	14756,943	10026,894	7498,342	4820,224
	15%	28689,472	14768,837	10252,750	7807,282	4819,902
	20%	28613,663	14791,350	10051,752	7500,784	4819,994
	25%	29091,267	14765,091	10341,259	7505,213	5513,249

Tabla AI.2. Tiempo de ejecución de todas las pruebas.

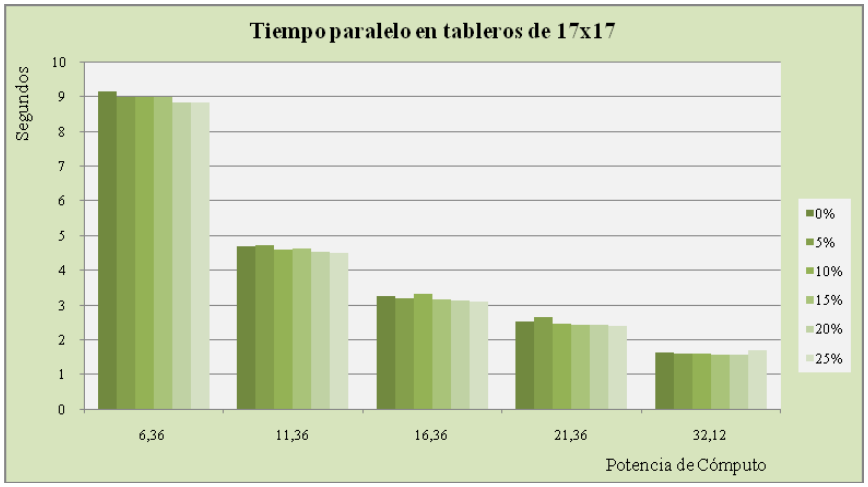


Figura AI.6. Tiempo de ejecución de las pruebas con tablero de 17x17.

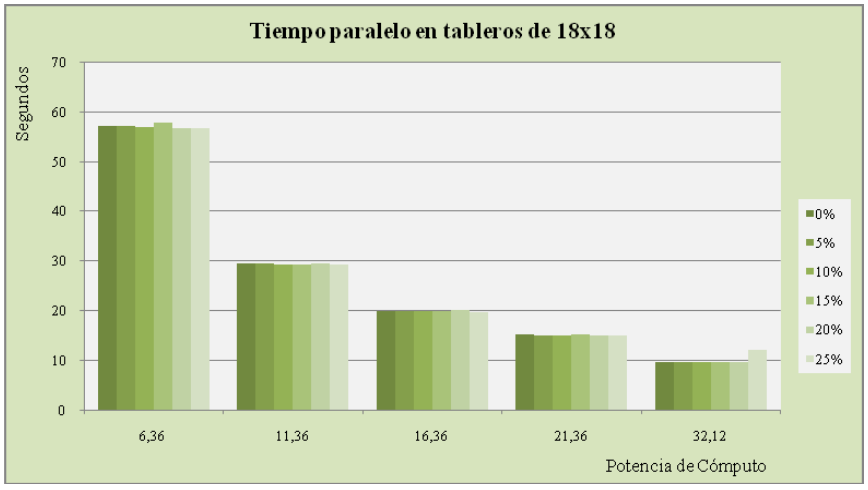


Figura AI.7. Tiempo de ejecución de las pruebas con tablero de 18x18.

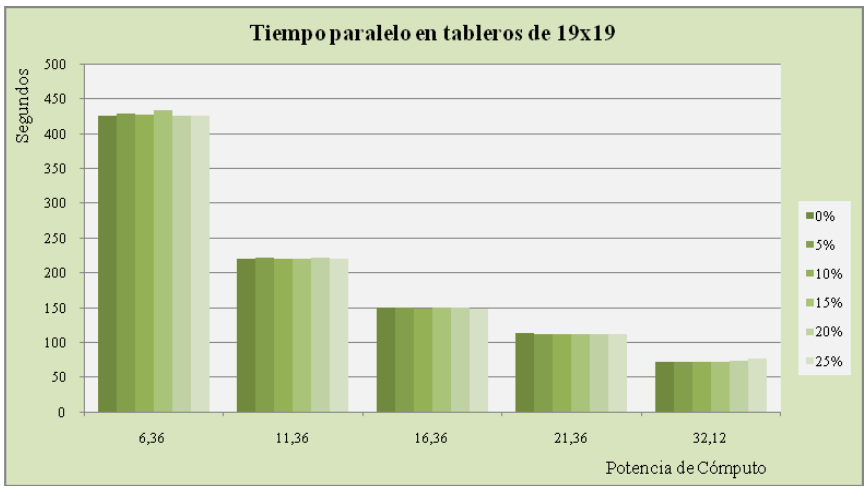


Figura AI.8. Tiempo de ejecución de las pruebas con tablero de 19x19.

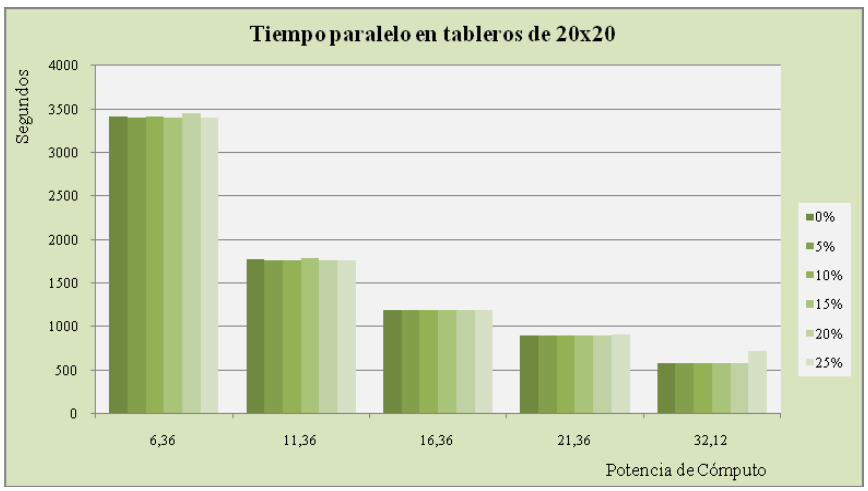


Figura AI.9. Tiempo de ejecución de las pruebas con tablero de 20x20.

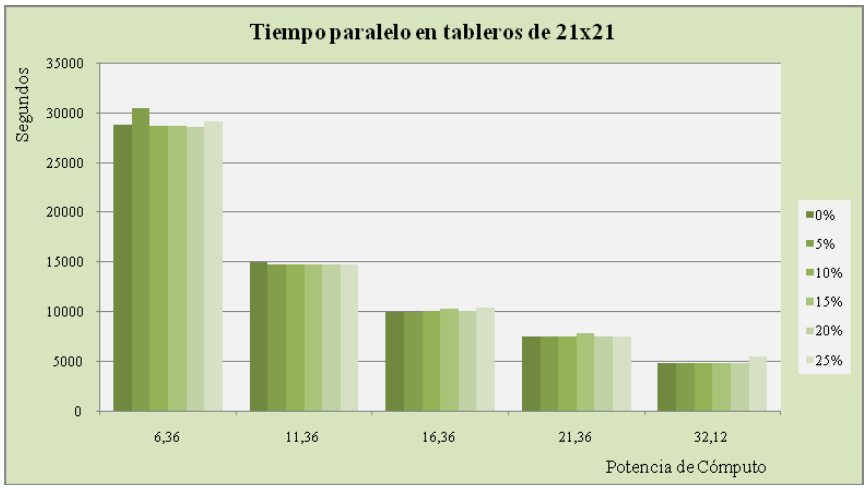


Figura AI.10. Tiempo de ejecución de las pruebas con tablero de 21x21.

AI.3. Speedup

La Tabla AI.3 muestra el speedup de las pruebas realizadas.

Tamaño	Li	*pct* = 6,36	*pct* = 11,36	*pct* = 16,36	*pct* = 21,36	*pct* = 32,12
17	0%	4,335	8,467	12,171	15,804	24,132
	5%	4,420	8,387	12,435	15,031	24,715
	10%	4,416	8,641	11,975	16,183	24,960
	15%	4,411	8,608	12,619	16,299	25,137
	20%	4,492	8,768	12,696	16,432	25,385
	25%	4,499	8,814	12,844	16,587	23,299
18	0%	5,044	9,763	14,404	19,041	30,065
	5%	5,036	9,771	14,506	19,172	30,154
	10%	5,059	9,807	14,475	19,200	30,195
	15%	4,978	9,827	14,459	19,066	30,248
	20%	5,071	9,781	14,312	19,264	29,793
	25%	5,078	9,849	14,594	19,256	23,999
19	0%	5,202	10,090	14,710	19,598	30,788
	5%	5,163	10,035	14,805	19,864	30,587
	10%	5,188	10,089	14,880	19,816	30,764
	15%	5,117	10,057	14,830	19,858	30,896
	20%	5,202	9,988	14,834	19,751	30,118
	25%	5,205	10,094	14,979	19,872	28,627
20	0%	5,214	10,005	15,009	19,906	30,852
	5%	5,221	10,113	14,889	19,777	30,893
	10%	5,213	10,124	14,874	19,880	30,915
	15%	5,232	9,937	15,013	19,834	30,948
	20%	5,145	10,126	15,023	19,770	30,991
	25%	5,230	10,076	14,916	19,639	24,901
21	0%	5,178	9,952	15,038	19,883	30,976
	5%	4,897	10,126	15,037	19,881	30,975
	10%	5,208	10,124	14,899	19,924	30,993
	15%	5,207	10,115	14,571	19,135	30,995
	20%	5,221	10,100	14,862	19,917	30,995
	25%	5,135	10,118	14,446	19,905	27,097

Tabla AI.3. Speedup de todas las pruebas.

Las Figuras AI.11, AI.12, AI.13, AI.14 y AI.15 muestran en forma gráfica el speedup de todas las pruebas por tamaño de tablero (*N*).

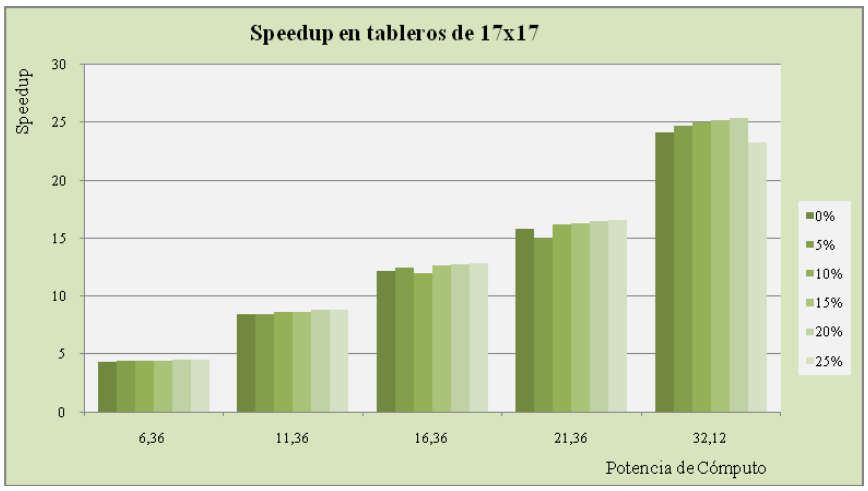


Figura AI.11. Speedup de las pruebas con tablero de 17x17.

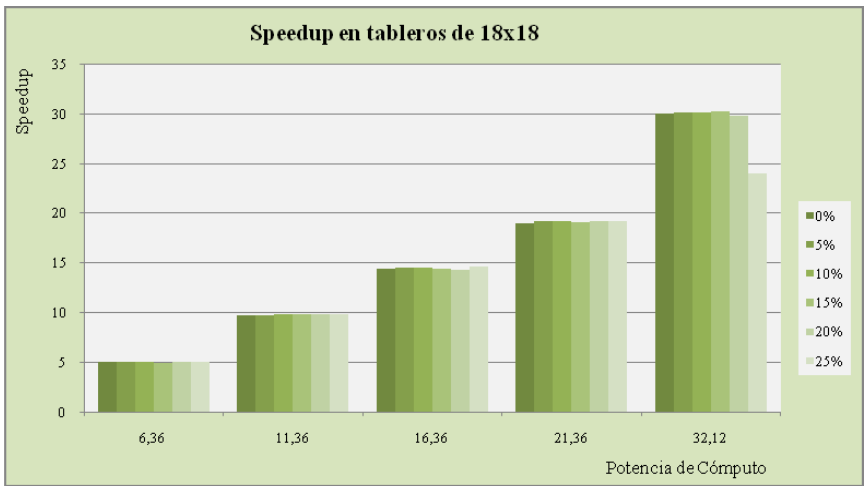


Figura AI.12. Speedup de las pruebas con tablero de 18x18.

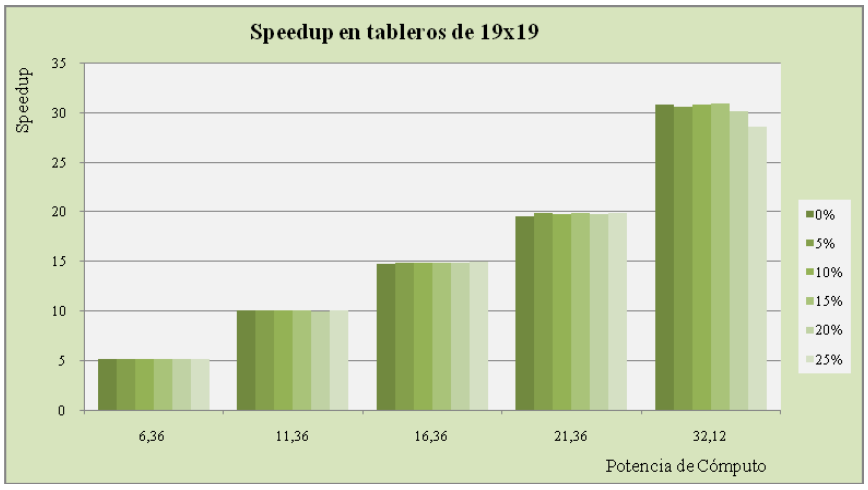


Figura AI.13. Speedup de las pruebas con tablero de 19x19.

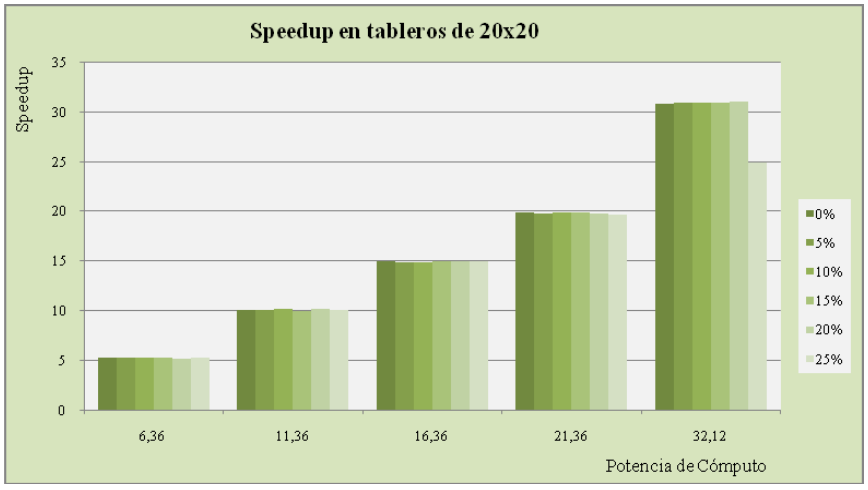


Figura AI.14. Speedup de las pruebas con tablero de 20x20.

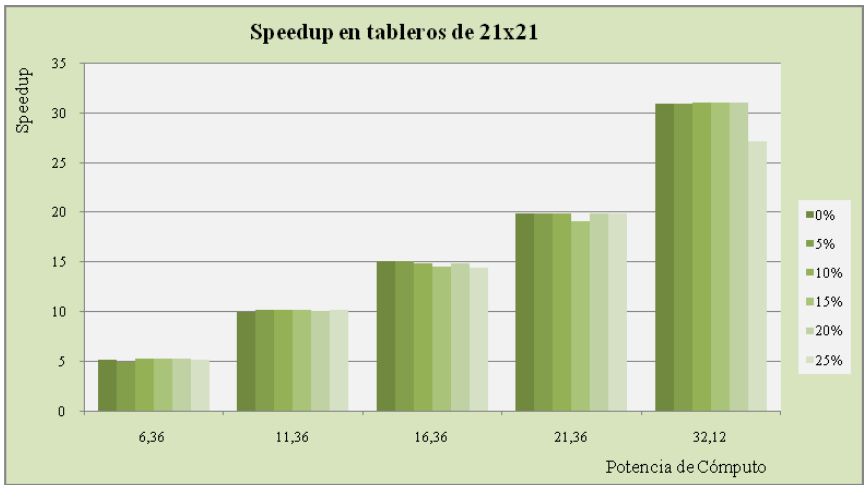


Figura AI.15. Speedup de las pruebas con tablero de 21x21.

AI.4. Eficiencia

La Tabla AI.4 muestra la eficiencia de las pruebas realizadas con los siguientes parámetros:

- *N* = 17; 18; 19; 20 y 21.
- *pct* = 6,36; 11,36; 16,36; 21,36 y 32,36.
- *Li*= 0; 5; 10; 15; 20; y 25.

Las Figuras AI.16, AI.17, AI.18, AI.19 y AI.20 muestran en forma gráfica estos mismos datos separados por tamaño de tablero (*N*).

Tamaño	Li	*pct* = 6,36	*pct* = 11,36	*pct* = 16,36	*pct* = 21,36	*pct* = 32,12
17	0%	0,682	0,745	0,744	0,740	0,751
	5%	0,695	0,738	0,760	0,704	0,769
	10%	0,694	0,761	0,732	0,758	0,777
	15%	0,694	0,758	0,771	0,763	0,783
	20%	0,706	0,772	0,776	0,769	0,790
	25%	0,707	0,776	0,785	0,777	0,725
18	0%	0,793	0,859	0,880	0,891	0,936
	5%	0,792	0,860	0,887	0,898	0,939
	10%	0,795	0,863	0,885	0,899	0,940
	15%	0,783	0,865	0,884	0,893	0,942
	20%	0,797	0,861	0,875	0,902	0,928
	25%	0,798	0,867	0,892	0,901	0,747
19	0%	0,818	0,888	0,899	0,917	0,959
	5%	0,812	0,883	0,905	0,930	0,952
	10%	0,816	0,888	0,910	0,928	0,958
	15%	0,805	0,885	0,906	0,930	0,962
	20%	0,818	0,879	0,907	0,925	0,938
	25%	0,818	0,889	0,916	0,930	0,891
20	0%	0,820	0,881	0,917	0,932	0,961
	5%	0,821	0,890	0,910	0,926	0,962
	10%	0,820	0,891	0,909	0,931	0,962
	15%	0,823	0,875	0,918	0,929	0,963
	20%	0,809	0,891	0,918	0,926	0,965
	25%	0,822	0,887	0,912	0,919	0,775
21	0%	0,814	0,876	0,919	0,931	0,964
	5%	0,770	0,891	0,919	0,931	0,964
	10%	0,819	0,891	0,911	0,933	0,965
	15%	0,819	0,890	0,891	0,896	0,965
	20%	0,821	0,889	0,908	0,932	0,965
	25%	0,807	0,891	0,883	0,932	0,844

Tabla AI.4. Eficiencia de todas las pruebas.

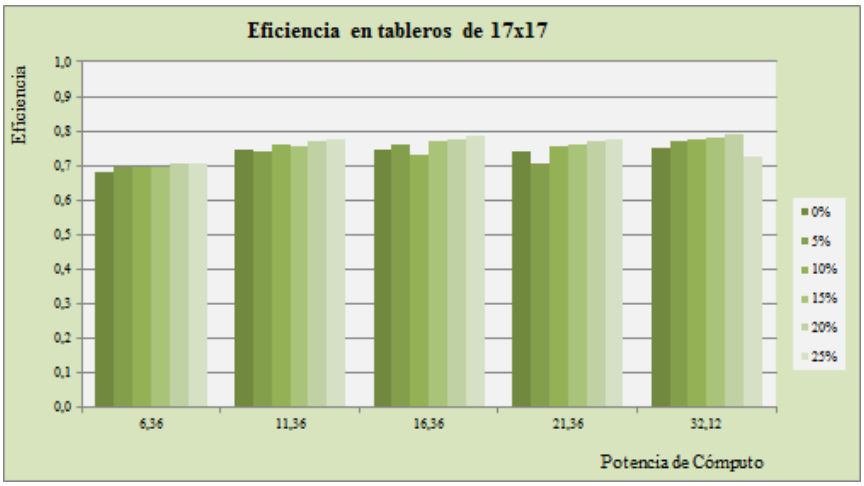


Figura AI.16. Eficiencia de las pruebas con tablero de 17x17.

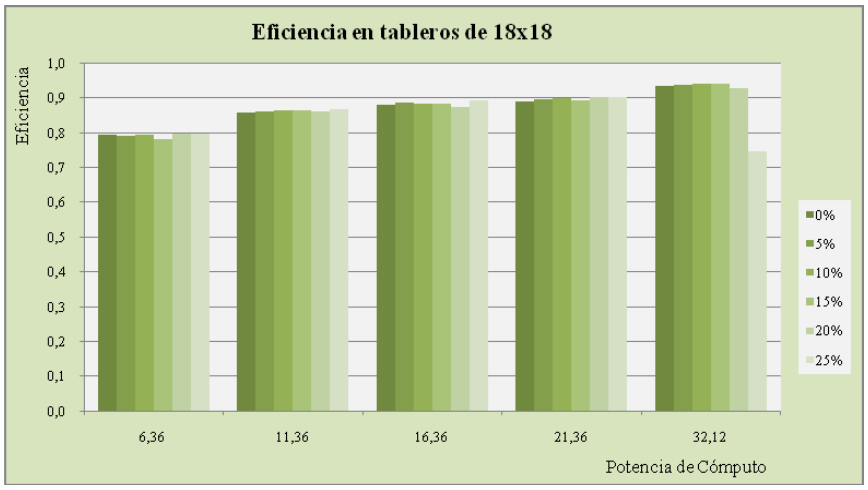


Figura AI.17. Eficiencia de las pruebas con tablero de 18x18.

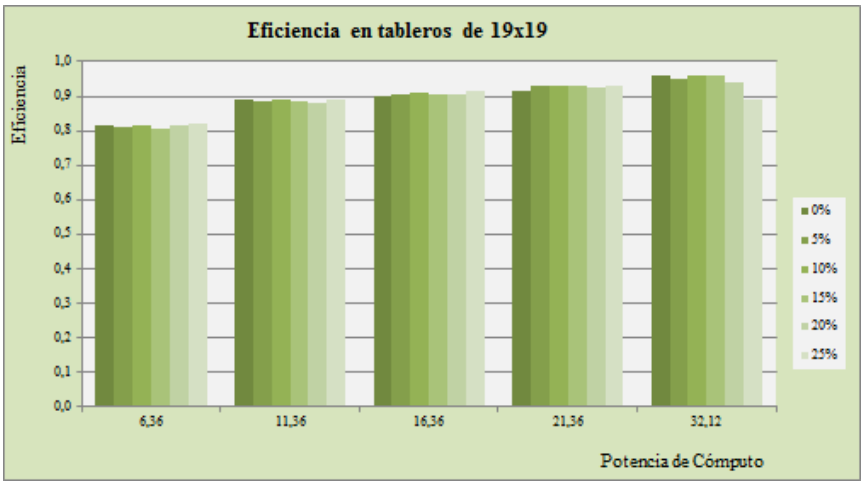


Figura AI.18. Eficiencia de las pruebas con tablero de 19x19.

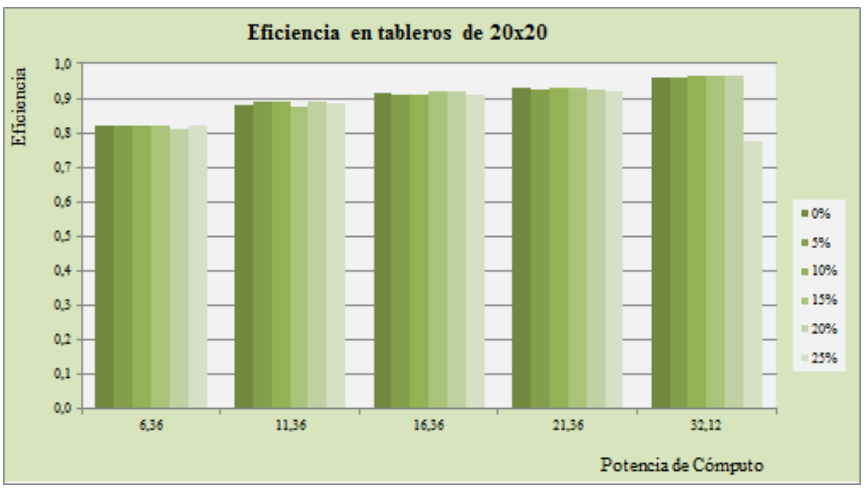


Figura AI.19. Eficiencia de las pruebas con tablero de 20x20.

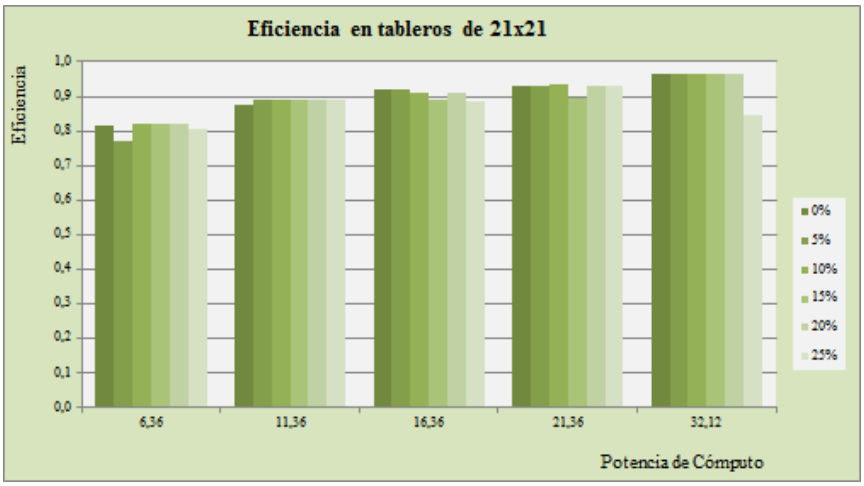


Figura AI.20. Eficiencia de las pruebas con tablero de 21x21.

Bibliografía

Al-Jaroodi, J., Mohamed, N., Jiang, H., & Swanson, D. (2003). Modeling Parallel Applications Performance on Heterogeneous System. *Parallel and Distributed Processing Symposium, 2003. Proceedings. International* (p. 7). Francia: IEEE Computer Society.

Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming.* EEUU: Addison Wesley Longman.

Baiardi, F., Chiti, S., Mori, P., & Ricci, L. (2001). Integrating Load Balancing and Locality in the Parallelization of Irregular Problems. *Future Generation Computer Systems, 17*(8), 969-975.

Ben-Ari, M. (2006). *Principles of Concurrent and Distributed Programming. Second edition.* Addison Wesley.

Bernhardsson, B. (1991). Explicit Solution to the N-queens Problems for all N. *ACM SIGART Bulletin, 2*(2), 7.

Bohn, C. A., & Lamont, G. B. (2002). Load Balancing for Heterogeneous Clusters of PCs. *Future Generation Computer Systems, 18*(3), 389-400.

Bruen, A., & Dixon, R. (1975). Then N-queens Problem. *Discrete Mathematics, 12*(4), 393-395.

Bubak, M., Funika, W., & Moscinski, J. (1997). Performance Analysis of Parallel Applications under Message Passing Environments. *The 2nd International Conference on Parallel Processing and Applied Mathematics* (pp. 414-422). Polonia: Technical University of Czestochowa.

De Giusti, L. C. (2011). *Mapping sobre arquitecturas heterogéneas - Tesis Doctoral en Ciencias Informáticas, Premio Dr. Raúl Gallard.* La Plata: Edulp.

De Giusti, L. C., Novarini, P., Naiouf, M., & De Giusti, A. E. (2003). Parallelization of the N-queens problem. Load unbalance analysis. *Proceeding del IX Congreso Argentino de Ciencias de la Computación (CACIC'03)*, (pp. 397-405). La Plata.

Dongarra, J., Foster, I., Fox, G. C., Gropp, W., Kennedy, K., Torczon, L., & White, A. (2003). *The Sourcebook of Parallel Computing.* EEUU: Morgan Kaufmann Publishers.

Grama, A., Gupta, A., Karypis, G., & Kumar, V. (2003). *An Introduction to Parallel Computing. Design and Analysis of Algorithms (2da Edition).* Inglaterra: Addison Wesley.

Hwang, K., & Xu, Z. (1998). *Scalable parallel computing: technology, architecture, programming.* McGraw-Hill.

Jordan, H. F., & Alaghband, G. (2002). *Fundamentals of Parallel Computing.* Prentice Hall.

Juhasz, Z., Kacsuk, P., & Kranzlmuller, D. (Eds.). (2004). *Distributed and Parallel Systems: Cluster and Grid Computing.* New York: Springer-Verlag.

Kumar, V., Gopalakrishnan, P. S., & Kanal, L. N. (2012). *Parallel algorithms for machine intelligence and vision.* Springer-Verlag.

Leopold, C. (2001). *Parallel and Distributed Computing. A survey of Models, Paradigms, and Approaches.* EEUU: Wiley.

McCool, M. D., Robison, A. D., & Reinders, J. (2012). *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann.

Moura e Silva, L., & Buyya, R. (1999). Parallel programming models and paradigms. In R. Buyya (Ed.), *High Performance Cluster Computing, volume 2, Programming and Applications* (pp. 4-27). Upper Saddle River: Prentice Hall PTR.

Naiouf, M. (2004). *Procesamiento Paralelo. Balance de Carga Dinámico en Algoritmos de Sorting.* La Plata: Tesis Doctoral. Universidad Nacional de La Plata.

Naiouf, M., De Giusti, L. C., Chichizola, F., & De Giusti, A. E. (2006). Dynamic Load Balancing on Non-homogeneous Clusters. In G. Min, B. Di Martino, L. T. Yang, M. Guo, & G. Rünger (Eds.), *Lecture Notes in Computer Science (Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops)* (Vol. 4331, pp. 68-73). Springer Berlin Heidelberg.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., & Dongarra, J. (1996). *MPI: The Complete Reference.* Cambridge: The MIT Press.

Somers, J. (2002). *The N Queens Problem a study in optimization*. Retrieved from www.jsomers.com/nqueen_demo/nqueens.html.

Takahashi, K. (2003). *N-queens problem (number of solutions)*. Retrieved from http://www.ic-net.or.jp/home/takaken/e/queen/

Tinetti, F. G. (2004). *Tesis Doctoral: Cómputo Paralelo en Redes Locales de Computadoras.* España: Universidad Autónoma de Barcelona.

Vaughan, F. A., Grove, D. A., & Coddington, P. D. (2003). Communication Performance Issues for Two Cluster Computers. *Proceedings of the 26th Australasian computer science conference. 16*, pp. 171-180. Australia: Australian Computer Society, Inc

Watts, J., & Taylor, S. (1998). A Practical Approach to Dynamic Load Balancing. *IEEE Transactions on Parallel and Distributed Systems, 9*(3), 235-248.

Wilkinson, B., & Allen, M. (2004). *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers (2da edition).* EEUU: Prentice Hall.

Zhang, X., & Yan, Y. (1995). Modeling and Characterizing Parallel Computing Performance on Heterogeneous Networks of Workstations. *Seventh IEEE Symposium on Parallel and Distributed Processing.* (pp. 25-34). IEEE Computer Society.