Algoritmos en paralelo para una variante del problema *bin packing*

Proyecto : 5402-1440-2601

MSc. Geovanni Figueroa M.
MSc. Ernesto Carrera R.
Instituto Tecnológico de Costa Rica
Vicerrectoría de Investigación y Extensión
Dirección de Proyectos

23 de junio de 2011

Índice general

Índice de figuras

Índice de cuadros

Resumen

En este informe se resumen los resultados obtenidos en la investigación realizada sobre una variante del problema *bin packing*. El objetivo fue diseñar e implementar algoritmos determinísticos y heurísticos en paralelo para resolver y aproximar la solución a dicho problema. Se presenta el problema; se hace un análisis de la complejidad del mismo; se mencionan algunos de los modelos existentes para la programación en paralelo así como algunas bibliotecas que permiten el desarrollo de algoritmos con estos modelos; se introducen las métricas usuales que permiten medir el desempeño de un algoritmo en paralelo; y se resumen los experimentos realizados. En los diseños de los algoritmos se utilizó el modelo exploratorio, y su implementación se realizó utilizando la biblioteca OpenMP en C. Los resultados obtenidos en instancias de prueba mostraron mejoras en el tiempo de ejecución de hasta 10x con respecto a las implementaciones secuenciales de los algoritmos respectivos. Estos resultados permiten concluir que el diseño propuesto y la implementación respectiva, resuelven de manera satisfactoria el problema planteado.

palabras clave: optimización combinatoria, optimización discreta, problema bin packing, heurísticas, algoritmos en paralelo.

Capítulo 1

Descripción del problema

1.1. Introducción

La optimización combinatoria es una rama de la optimización en general. Se encarga básicamente de problemas de optimización donde la región de soluciones factibles es discreta o puede ser reducida a una región discreta, y tiene como objetivo encontrar la mejor solución. Esta reducción, de manera sorprendente, convierte problemas que en el caso general pueden ser resueltos con relativa facilidad, en problemas sumamente complejos en su tiempo de solución.

Involucra las matemáticas aplicadas y ciencias de la computación, y está relacionada con investigación de operaciones, teoría de algoritmos y teoría de la complejidad computacional, y abarca áreas como la inteligencia artificial, las matemáticas y la ingeniería de software.

Dentro de la optimización combinatoria, las líneas de investigación han ido, tanto por definir técnicas generales de solución (*branch and bound*, optimización lineal entera, *branch and price* y generación de columnas), como por resolver de manera óptima, o encontrar heurísticas eficientes, para problemas específicos.

Debido a la inconveniencia de encontrar soluciones óptimas a problemas complejos, es que se suelen utilizar heurísticos que aseguren, al menos, una "buena"solución. Es decir, una solución que, aunque no es óptima, es aceptable para el que presenta el problema. Dentro de dichos heurísticos, últimamente ha tomado gran fuerza la utilización de meta–algoritmos evolutivos, los cuales están inspirados en características de la evolución biológica: reproducción, mutación, recombinación y selección, e incluyen técnicas como algoritmos genéticos y simulated annealing entre otros.

La computación paralela es una técnica de programación que ha tenido un fuerte empuje en los últimos años, debido a que el desarrollo de microprocesadores ha alcanzado un límite en lo que se refiere al número máximo de operaciones que pueden ejecutarse por segundo. Esto ha motivado el desarrollo de las tecnologías de procesadores paralelos (multinúcleo) y procesos distribuidos.

La programación tradicional está orientada hacia la computación secuencial; en ésta, el flujo de instrucciones se ejecuta secuencialmente, una a una. En la computación paralela se emplean elementos de procesamiento simultáneo; esto se logra dividiendo el problema en subprocesos independientes de tal manera que cada elemento de procesamiento pueda ejecutarse a la vez. Estos elementos de procesamiento pueden ser diversos e incluir recursos tales como un único ordenador con muchos procesadores, varios ordenadores en red, hardware especializado o una combinación de los mismos. Esto hace que esta técnica sea una herramienta útil para resolver problemas computacionalmente complejos; sin embargo, ello aumenta la dificultad del diseño de un algoritmo en

paralelo, debido a que la comunicación y sincronización entre los diferentes subprocesos debe realizarse adecuadamente para conseguir un buen rendimiento.

En el contexto de los métodos heurísticos, el paralelismo no sólo significa resolver los problemas de forma más rápida, sino que además se obtienen modelos de búsqueda más eficientes: un algoritmo heurístico paralelo puede ser más efectivo que uno secuencial, aun ejecutándose en un sólo procesador [1].

El problema estudiado tiene un espacio factible muy grande [5]. Incluso para instancias pequeñas del problema la solución óptima no se puede hallar en un tiempo razonable, y para heurísticos, mientras mayor sea la cantidad de trabajo realizado, más cercana estará la aproximación de la solución óptima.

Este proyecto tiene como objetivo diseñar e implementar algoritmos en paralelo que permitan reducir el tiempo de ejecución, en comparación con los algoritmos actuales, tanto heurísticos como algoritmos que resuelven el problema de manera óptima.

Como resultado de la investigación en el proyecto *Optimización de Lipschitz, optimización combinatoria y algoritmos evolutivos* [6] se han propuesto algoritmos secuenciales tanto para aproximar como para resolver de forma óptima el problema. Por tratarse de un problema *NP*, el tiempo de cómputo de la solución, incluso para instancias pequeñas es excesivo, por esta razón el uso de algoritmos en paralelo es una alternativa natural de investigar, para encontrar buenas aproximaciones o soluciones óptimas.

1.2. Planteamiento del problema

En el trabajo de tesis de maestría del MSc. Luis Carrera [3], junto con su asesor el Dr. Feliú Sagols, se resolvió el siguiente problema de optimización combinatoria:

Problema P: Dados $\mathbf{d} = [d_1, d_2, \ldots, d_n]$, c y m, con $d_i \in \mathbb{N}$ para $i = 1, \ldots, n$, $m \in \mathbb{N}$, $c \in \mathbb{N}$, $m \cdot c \geq n$, determinar $\mathbf{b} = (b_1, \ldots, b_m)$ y $S_{n\times m} = [s_{i,j}]$, sujeto a:

$$b_j \in \mathbb{N} \qquad j = 1, \ldots, m \tag{1.1}$$

$$s_{i,j} = 0, \ldots, c \qquad i = 1, \ldots, n;\ j = 1, \ldots, m \tag{1.2}$$

$$\sum_{i=1}^{n} s_{i,j} = c \qquad j = 1, \ldots, m \tag{1.3}$$

$$\sum_{j=1}^{m} b_j s_{i,j} \geq d_i \qquad i = 1, \ldots, n \tag{1.4}$$

tal que se minimice $\displaystyle\sum_{j=1}^{m} b_j$.

1.3. Complejidad del problema

Con la idea de evidenciar la complejidad del problema considere la siguiente situación: suponga que una tienda de abarrotes recibe mercadería nueva, pero se percatan de que tienen almacenados 97 jugos de uva, 76 de manzana y 68 de naranja, y podrían caducar si no se venden pronto. Se

!htb

Cuadro 1.1: Matriz para representar una solución al problema.

$$\left(\begin{array}{c|cc|c} & x & y & x+y \\ \hline d_1 & a_{11} & a_{12} & r_1 \\ d_2 & a_{21} & a_{22} & r_1 \\ d_3 & a_{31} & a_{32} & r_1 \end{array}\right)$$

sugiere entonces ponerlos en oferta en 2 tipos de paquetes surtidos con 6 jugos, de manera que todos los paquetes de un mismo tipo sean iguales (es decir, que tengan los mismos sabores, y la misma cantidad de jugos de cada sabor).

¿Cómo se deben empacar los jugos, de manera que se pongan en oferta todos los jugos que estaban almacenados y se minimice el número de paquetes?

Observe que se tienen tres tipos diferentes de jugos (objetos, $n = 3$) y dos tipos diferentes de paquetes (recipientes, $m = 2$). Para modelar el problema suponga que:

- x es el número de paquetes necesarios del primero tipo (repeticiones del recipiente 1).
- y es el número de paquetes necesarios del segundo tipo (repeticiones del recipiente 2).
- a_{ij} es el número de jugos de tipo i (uva, manzana o naranja) que se deben poner en el recipiente de tipo j (número de copias del objeto i-ésimo en el recipiente j-ésimo).
- $d = (d_1, d_2, d_3) = (97, 76, 68)$ los requerimientos de cada tipo de jugo.
- $c = 6$ la capacidad de cada paquete (capacidad del recipiente).

Con esta notación se puede representar una instancia del problema por medio de la matriz que se muestra en el Cuadro 1.1, donde se han incluido los términos r_1, r_2 y r_3 los cuales representan el exceso o sobrante respecto a la cantidad deseada para cada uno de los objetos, y $x + y$ representa el valor de la solución al problema (total de paquetes necesarios).

Por otro lado, como el número de jugos por paquete debe ser 6, deben satisfacerse la siguiente condición:

$$\sum_{i=1}^{3} a_{ij} = 6 \qquad j = 1, 2 \tag{1.5}$$

Además, para agotar las existencias se debe cumplir que:

$$\begin{aligned} a_{11}x + a_{12}y &\geq 97 \\ a_{21}x + a_{22}y &\geq 76 \\ a_{31}x + a_{32}y &\geq 68 \end{aligned} \tag{1.6}$$

Y se quiere minimizar $f(x, y) = x + y$ (para utilizar la menor cantidad de jugos "extra").

Observe que el problema es complejo, pues no sólo se debe minimizar el costo, sino que además se debe buscar la forma de distribuir los jugos en los paquetes, esto hace que sea imposible aplicar directamente alguna técnica de programación lineal o entera, pues a priori no se conoce la distribución que se debe asignar a cada tipo de paquete. Por ejemplo, una solución se muestra en la

Cuadro 1.2: Una solución factible al problema ejemplo.

$$\left(\begin{array}{c|cc|c} & 20 & 24 & 44 \\ \hline 97 & 4 & 1 & 7 \\ 76 & 1 & 3 & 16 \\ 68 & 1 & 2 & 0 \end{array}\right)$$

Cuadro 1.2. afirma que si usamos 20 paquetes del tipo 1 y 24 paquetes del tipo 2 (ambos con capacidad para 6 jugos) el total de paquetes sería de 44. Además, en cada paquete de tipo 1 se deben colocar cuatro jugos de uva, uno de manzana y uno de naranja; mientras que en cada paquete de tipo 2 se deben colocar uno de uva, tres de manzana y dos de naranja. Observe que esta solución satisface las condiciones (1.5) y (1.7). Por otro lado, como deben ponerse en oferta todos los jugos existentes, pueden requerirse jugos extra, por ejemplo, en este caso se requieren 7 jugos de uva y 16 de manzana.

1.3.1. Espacio factible

Para tener una idea de la complejidad del problema vamos a explorar el espacio factible. Para esto calculemos todas las posibles matrices de distribución de jugos A^k que pueden presentarse:

$$A^k = \begin{pmatrix} a_{11}^k & a_{12}^k \\ a_{21}^k & a_{22}^k \\ a_{31}^k & a_{32}^k \end{pmatrix}$$

Observe que como $a_{1j}^k + a_{2j}^k + a_{3j}^k = 6$, donde a_{ij}^k es un entero entre 0 y 6 para toda $i = 1, 2, 3$, y $j = 1, 2$, se tienen 28 posibles tripletas con suma 6, como se muestra en el Cuadro 1.3. Aquí, se indican los valores de los a_{ij}^k presentes en la tripleta, mientras que los restantes son cero; por ejemplo, hay 6 tripletas que contienen los valores 1 y 5.

a_{ij}^k	total
6	3
1,5	6
2,4	6
3,3	3
1,1,4	3
1,2,3	6
2,2,2	1

Cuadro 1.3: Tripletas con suma 6.

Así, se tienen $28^2 = 784$ posibles matrices de distribución, alguna o algunas de las cuales producen una solución óptima, es decir, minimizan el número de paquetes.

En la Figura 1.1 se muestran las soluciones óptimas obtenidas al resolver la relajación lineal del

problema:

$$\begin{array}{ll} \text{mín} & x+y \\ & A\cdot(x,y)^t \geq d^t \\ & a_{ij} \in 0,1,\cdots,c \\ & x,y \in \mathbb{N} \end{array}$$

por medio de programación lineal, para cada matriz de distribución y para el vector de demandas $d = (97, 76, 68)$. Observe que detrás de cada uno de los puntos de la gráfica de la Figura 1.1 está una o varias matrices de distribución la cuales tienen a este punto como mejor solución. La recta $x + y = 41$ que se muestra en la Figura 1.1 corresponde a la solución óptima del problema. Observe que existen varias distribuciones que alcanzan el óptimo (puntos sobre la recta $x+y = 21$), por ejemplo las que se muestran en los Cuadros 1.4 y 1.5.

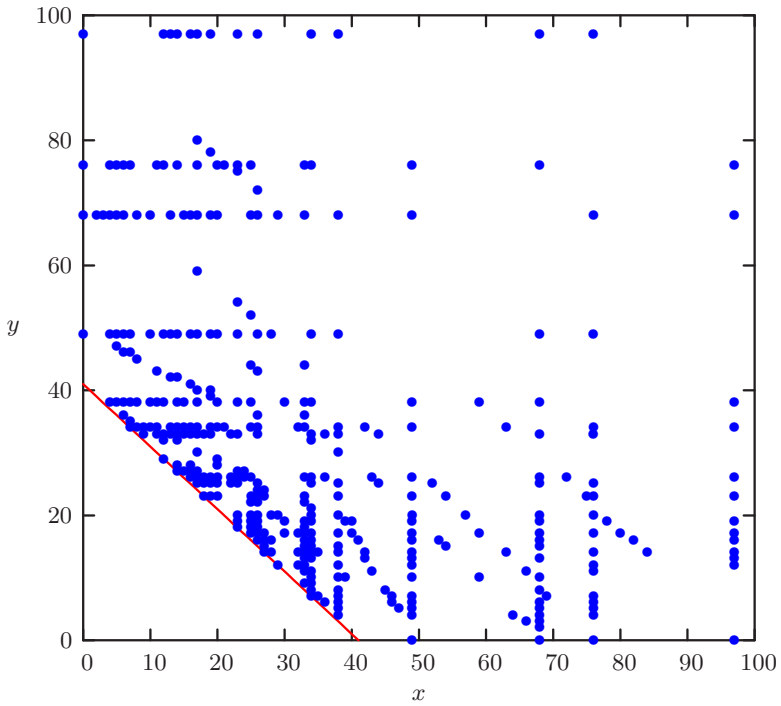


Figura 1.1: Soluciones para todas las posibles matrices de distribución.

Cuadro 1.4: Una distribución óptima al problema ejemplo.

	23	18	41
97	2	3	3
76	1	3	1
68	3	0	1

En general, para determinar el número posibles de matrices de distribución A^k, se determina el

Cuadro 1.5: Otra distribución óptima al problema ejemplo.

$$\left(\begin{array}{c|cc|c} & 34 & 7 & 41 \\ \hline 97 & 3 & 0 & 5 \\ 76 & 1 & 6 & 0 \\ 68 & 2 & 0 & 0 \end{array}\right)$$

número de distribuciones para cada columna, tal que

$$\sum_{i=1}^{n} a_{ij} = c$$

donde c es la capacidad de cualquiera de los recipientes. El número de distribuciones está dado por las combinaciones:

$$\left(\begin{array}{c} n+c-1 \\ n-1 \end{array}\right) = \frac{(n+c-1)!}{(n-1)!\ c!}$$

y en consecuencia el número total de matrices está dado por:

$$\left(\begin{array}{c} n+c-1 \\ c-1 \end{array}\right)^{m} \tag{1.7}$$

Observe que la cantidad de posibles matrices A^k aumenta drásticamente con respecto al aumento de cualquiera de los parámetros c, n ó m. Por otro lado, note que no toda matriz A^k es una solución factible, por ejemplo, aquellas matrices tales que

$$a_{i1} = a_{i2} = \cdots = a_{in} = 0$$

para algún $i \in \{1, 2, \cdots, m\}$, es decir, las matrices donde no se asignaron copias de la demanda i-ésima en ninguno de los recipientes. Esto hace que la ecuación (1.7) cuente algunas matrices de más.

1.4. Objetivos del proyecto

Los objetivos propuestos inicialmente para la investigación, los cuales se cumplieron por completo, fueron los siguientes:

1. *Objetivo general*

   Diseñar e implementar algoritmos en paralelo que aproximen y que resuelvan la variante propuesta al problema *bin packing*.

2. *Objetivos específicos*

   a) Investigar diferentes opciones disponibles para la programación de algoritmos en paralelo.

   b) Estudio de la plataforma escogida para la programación de algoritmos en paralelo.

c) Diseñar una paralelización para el heurístico que encuentra una solución aproximada al problema.

d) Implementar el algoritmo en paralelo para el diseño de paralelización del heurístico.

e) Diseñar una paralelización para el método que encuentra la solución óptima al problema.

f) Implementar el algoritmo en paralelo para el diseño de paralelización del óptimo.

g) Comparación de los tiempos de ejecución.

Capítulo 2

Programación paralela

La computación paralela es una técnica de programación que emplea elementos múltiples de procesamiento simultáneo para resolver un problema. Esto se logra dividiendo el problema en partes independientes de tal manera que cada elemento de procesamiento pueda ejecutar su parte de forma simultánea. Los elementos de procesamiento son diversos, van desde un único computador con muchos procesadores, redes de computadoras o hasta hardware especializado.

En los últimos años el interés por la computación paralela ha aumentado debido a la restricción física que impiden el escalado de frecuencia[1]. Esto ha hecho que se aproveche la mejor y mayor integración de los transistores para introducir otro tipo de mejoras, llegando así hasta los *cores* de hoy en día. Es decir, para ganar en rendimiento se ha pasado de escalar en frecuencia a escalar en paralelismo o procesamiento multinúcleo.

2.1. Diseño de algoritmos en paralelo

El diseño de algoritmos es fundamental al resolver problemas mediante el computador. Un algoritmo secuencial es esencialmente una receta o una sucesión de pasos usados para resolver un problema dado, usando un procesador. De forma similar un algoritmo en paralelo es una receta que dice como resolver un problema usando múltiples procesadores. Sin embargo, especificar un algoritmo en paralelo es más que simplemente describir los pasos a seguir, se deben considerar aspectos como la concurrencia y el diseñador debe indicar el conjunto de pasos que pueden ser ejecutados simultáneamente. Esto es esencial para lograr obtener ventajas del uso de un computador paralelo. Al diseñar un algortimo en paralelo se debe tomar en cuenta todos o algunos de los siguientes aspectos:

- Identificar porciones de trabajo que puedan ser ejecutadas simultáneamente.
- Asignar estas porciones de trabajo a múltiples procesadores que se ejecutan en paralelo.
- Distribuir las entradas, salidas y cálculo de datos intermedios asociados con el algoritmo.
- Administrar los accesos a datos compartidos por múltiples procesadores.
- Sincronizar los procesos en algunas etapas de la ejecución en paralelo.

[1]Cada vez es más difícil mejorar el rendimiento aumentando la frecuencia debido al aumento en el consumo de energía y consecuentemente de generación de calor.

2.1.1. Descomposición y dependencia de tareas

El proceso de dividir un problema en subproblemas más pequeños (tareas) los cuales potencialmente pueden ser ejecutados en paralelo se conoce como **descomposición**. El programador define en cuáles y en cuántas tareas se divide el problema principal; éstas pueden tener diferente tamaño, pero una vez definidas, deben considerarse como una unidad indivisible. La ejecución simultánea de múltiples tareas es la clave para reducir el tiempo requerido para resolver un problema entero.

En general hay tareas que pueden ser ejecutadas a la vez o en cualquier orden, son **independientes**, sin embrago, existen algunas que necesitan datos producidos por otras tareas por lo que puede ser necesario esperar a que estas finalizen, esto se conoce como **sincronización**.

2.1.2. Granularidad y concurrencia

El número y el tamaño de las tareas en las cuales un problema se descompone determina la **granularidad** de la descomposición. Si el problema se descompone en un gran número de tareas pequeñas se dice que la descomposición es **fina** y si se descompone en pocas tareas de gran tamaño se dice que la descomposición es **gruesa**.

Un concepto importante y relacionado con la granularidad es el número máximo de tareas que pueden ser ejecutadas simultáneamente por un programa en cualquier momento, esto se conoce como **grado máximo de concurrencia**. En la mayoría de los casos es menor que el número total de tareas debido a la dependencia entre tareas.

Un indicador más útil para medir el desempeño de un algoritmo en paralelo es el **grado promedio de concurrencia**, el cual es el número promedio de tareas que pueden ejecutarse simultáneamente durante la ejecución del programa.

Ambos, el grado máximo de concurrencia y el grado promedio de concurrencia usualmente aumentan a medida que la granularidad se hace más fina. Esto nos podría hacer pensar que el tiempo requerido para resolver un problema puede ser reducido simplemente incrementando la granularidad de la descomposición con el fin de realizar más y más tareas en paralelo, pero esto no es lo que sucede en la práctica, pues existe una cota inherente que establece que tan fina puede ser la descomposición de un problema.

Otro factor que limita la granularidad y el grado de concurrencia es la **interacción** entre las tareas que están en ejecución. A menudo las tareas comparten entradas, salidas o datos intermedios. Esta dependencia es usualmente el resultado de que una salida de una tarea es la entrada de otra tarea.

2.2. Técnicas de descomposición

Como ya mencionamos, uno de los pasos fundamentales para diseñar un algoritmo en paralelo es dividir los cálculos a realizar en un conjunto de tareas que se ejecutarán simultáneamente. Existen muchas técnicas de descomposición pero ninguna garantiza que el algoritmo resultante sea el mejor algoritmo paralelo para resolver un problema determinado. A pesar de esta deficiencia, utilizar una de las técnicas de descomposición o una combinación de ellas a menudo proporciona una descomposición eficaz.

2.2.1. Descomposición recursiva

Es un método para inducir la concurrencia que puede ser aplicado a problemas que pueden resolverse mediante la estrategia de *divide y vencerás*. En esta técnica, un problema se resuelve dividiéndolo en un conjunto de subproblemas independientes. Cada uno de estos subproblemas se resuelve de forma recursiva mediante la aplicación de una división similar en pequeños subproblemas, seguido de una combinación de sus resultados.

2.2.2. Descomposición de datos

La descomposición de datos es un técnica comúnmente utilizada para inducir la concurrencia en algoritmos que trabajan sobre grandes estructuras de datos. En este método, la descomposición de los cálculos se realiza en dos etapas. En la primera etapa, los datos sobre los que se realizarán los cálculos se particionan, y en la segunda etapa, estas particiones de datos se utilizan para inducir una división de los cálculos en tareas. Las operaciones que realizan estas tareas en diferentes particiones de datos suelen ser similares

La partición de datos se puede realizar de muchas formas; en general, hay que explorar y evaluar todas las formas posibles de dividir los datos y determinar cuál de ellas produce una descomposición natural y eficiente.

2.2.3. Descomposición exploratoria

Se utiliza para descomponer problemas en los cuales los cálculos subyacentes corresponden a una búsqueda sobre un espacio de soluciones. En la descomposición exploratoria se particiona el espacio de búsqueda en subespacios más pequeños y se explora simultáneamente en cada uno de estos subespacios, hasta que las soluciones deseadas se encuentran.

2.2.4. Descomposición híbrida

Las técnicas de descomposición anteriores no son exclusivas y a menudo se pueden combinar entre sí. Con frecuencia un cálculo se puede dividir en mútiples etapas a las cuales se les puede aplicar diferentes tipos de descomposición.

Hoy en día es muy difícil desarrollar un programa paralelo de forma automática. Por esto, se debe escoger el modelo apropiado, o una mezcla de ellos, para implementar un algoritmo dado.

2.3. Modelos de programación paralela

Un modelo de programación paralela es un conjunto de tecnologías hardware y software que permiten implementar algoritmos paralelos en la arquitectura adecuada. Desde el punto de vista del uso de la memoria existen tres modelos: los de memoria compartida (`OpenMP`), los de memoria distribuida (`MPI, PVM`) y los modelos híbridos.

2.3.1. MPI

`MPI` (Message Passing Interface) es un protocolo de comunicación entre computadoras. Es un estándar que define la sintaxis y semántica de las funciones contenidas en una biblioteca usada

para la comunicación entre los distintos nodos que ejecutan un programa en un sistema de memoria distribuida.

El paso de mensajes es una técnica empleada en programación concurrente que permite sincronizar procesos. Su principal característica es que no requiere de memoria compartida, por lo que se usa en la programación de sistemas distribuidos. El programador explícitamente divide el trabajo y los datos entre varios procesos y debe gestionar la comunicación entre ellos.

Los elementos principales que intervienen en el paso de mensajes son el proceso que envía, el que recibe y el mensaje. Dependiendo de si el proceso que envía el mensaje espera a que el mensaje sea recibido, se puede hablar de paso de mensajes sincrónico o asincrónico. En el paso de mensajes asincrónico, el proceso que envía no espera a que el mensaje sea recibido, y continúa su ejecución, siendo posible que vuelva a generar un nuevo mensaje y a enviarlo antes de que se haya recibido el anterior. Por este motivo es necesario sincronizar procesos: el proceso que envía mensajes sólo se bloquea o se detiene cuando finaliza su ejecución. En el paso de mensajes sincrónico, el proceso que envía el mensaje espera a que un proceso lo reciba para continuar su ejecución [2].

Las implementaciones de `MPI` son un conjunto de bibliotecas de rutinas que pueden ser utilizadas en programas escritos en los lenguajes de programación como: `C`, `C++` y `Fortran`. La ventaja de `MPI` sobre otras bibliotecas de paso de mensajes, es que los programas que utilizan la biblioteca son rápidos (pues la implementación de la biblioteca ha sido optimizada para el hardware en la cual se ejecuta) y portables (pues `MPI` ha sido implementado para casi todas las arquitecturas de memoria distribuida).

2.3.2. Posix Threads

`POSIX Threads`, o simplemente `Pthreads` es un estándar para la programación de aplicaciones multi-hilo. Existen implementaciones prácticamente para todos los sistemas operativos. En lenguajes de programación como `C/C++` y `C#` se implementa como una biblioteca que incluye tipos, funciones y constantes que permiten crear y manipular hilos, los cuales son usados para efectuar tareas simultáneas dentro de un mismo proceso. El uso de hilos es más efectivo en sistemas con varios procesadores o multi-núcleo, pues el flujo del proceso puede ser programado de forma tal que se beneficie del paralelismo o procesamiento distribuido.

2.3.3. OpenMP

`OpenMP` es una interfaz de programación de aplicaciones (API) usada para la programación multiproceso de memoria compartida. Está disponible para muchas arquitecturas, incluidas las plataformas de `UNIX`, `Microsoft Windows` y `Mac OS X`. Se compone de un conjunto de directivas de compilación, rutinas de biblioteca, y variables de entorno que afectan el comportamiento de un programa en tiempo de ejecución.

Esta aproximación es completamente flexible, si bien requiere un mayor trabajo del programador. Puede ser empleada en arquitecturas de memoria distribuida e incluso en redes de computadores, y también en arquitecturas de memoria compartida, aunque en ese caso los diferentes procesos utilizan partes de memoria independientes y separadas.

`OpenMP` es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas, tanto para computadoras de escritorio como supercomputadoras. Se fundamente en el modelo **fork-join** en donde una tarea

muy *pesada* se divide en k subtareas (hilos) de menor *peso* (fork), para luego *unir* estos resultados parciales (join) en una respuesta final.

Cuando se incluye una directiva de `OpenMP` el bloque de código se marca como paralelo y se lanzan hilos según la característica de la directiva y al final de ella se sincronizan los resultados para los diferentes hilos [11].

Se pueden construir aplicaciones con un modelo de programación paralela híbrida usando `MPI` y `OpenMP`, o a través de las extensiones de `OpenMP` para los sistemas de memoria distribuida. Estas aplicaciones se ejecutan en un `cluster` de computadoras [9].

De estos modelos de programación elegimos `OpenMP` por tratarse de un modelo de memoria compartida que mejor se adapta al problema de estudio y además porque tenemos acceso directo a un computador en que podemos realizar la implementación bajo este esquema.

2.4. Métricas para evaluar el desempeño de un sistema paralelo

Usualmente un algoritmo secuencial es evaluado en términos de su tiempo de ejecución, el cual se expresa como una función del tamaño de la entrada. Por otro lado, la estimación del tiempo de ejecución de un algoritmo paralelo es más complicado, pues no solo depende del tamaño de la entrada, sino que se deben considerar factores como el número de procesadores y parámetros de comunicación, tales como la manera en que estań interconectados los procesadores entre sí y con los módulos de memoria. Esto hace que se deba considerar como un sistema paralelo, compuesto por el algoritmo y la arquitectura en la cual se ejecutará [8].

Existen algunas métricas intuitivas para evaluar el desempeño de un sistema paralelo. Tal vez, la más simple es el tiempo reloj que tomó resolver una instancia de un problema dado en un sistema paralelo, pero esta medida tiene el inconveniente de que no puede ser extrapolada a otras instancias del problema o a otra configuración de equipo más grande. Por esta razón, son necesarias métricas más complejas que permiten cuantificar y extrapolar el desempeño de un sistema paralelo. Con este fin se han propuesto algunas métricas como las de `speedup` (aceleración) y la `eficiencia`. Estas métricas intentan responder a preguntas como:

1. ¿Qué algoritmo es más rápido entre varias alternativas?
2. ¿Qué tan beneficioso es resolver el problema en paralelo?
3. ¿Cómo se comporta el algoritmo al variar alguna de las características del equipo o del problema?

2.4.1. Speedup

Cuando evaluamos un sistema paralelo a menudo estamos interesados en conocer cuánto desempeño se logra al paralelizar una aplicación respecto a una implementación secuencial. El `speedup` es una medida que refleja el beneficio de resolver un problema en paralelo. Se define como el cociente del tiempo que toma resolver una instancia de un problema con un sólo procesador entre el tiempo que toma resolver la misma instancia del problema en paralelo con p procesadores idénticos, es decir,

$$S_p = \frac{T_1}{T_p}$$

donde p es el número de procesadores, T_1 es el tiempo de ejecución del algoritmo secuencial y T_p es el tiempo de ejecución del algoritmo paralelo usando p procesadores.

El `speedup` responde a la pregunta ¿qué tan beneficioso es resolver el problema de forma paralela? y típicamente es un valor entre 0 y p.

El `speedup ideal` o `lineal` se obtiene cuando $S_p = p$, esto quiere decir que al trabajar con p procesadores, el problema se resolverá p veces más rápido comparado con el mejor algoritmo secuencial. Teóricamente, el `speedup` nunca excede el número de procesadores pero algunas veces se presenta un aumento de velocidad mayor a p utilizando p procesadores, es decir, $S_p > p$, esto se conoce como `superlinear speedup` y se debe a ciertas características del hardware presente en las computadoras modernas [8].

Dependiendo de como se calcule T_1 se habla de:

- `Speedup absoluto`: si T_1 se calcula con el mejor algoritmo secuencial conocido.
- `Speedup relativo`: si T_1 se calcula ejecutando el algoritmo paralelo con un sólo procesador.

2.4.2. Eficiencia

Sólo en un sistema paralelo ideal con p procesadores se puede esperar un `speedup` igual a p. En la práctica, el comportamiento ideal no se alcanza porque durante la ejecución del algoritmo paralelo, los procesadores no están dedicando el 100 % de su capacidad a los calculos que demanda el algoritmo. La `eficiencia` (E) es una medida de la fraccción de tiempo durante la cual los procesadores son usados con eficacia. Se define como la razón del `speedup` (S_p) entre el número de procesadores p. En un sistema paralelo ideal se tiene que $S_p = p$ con lo cual $E = 1$, pero como en la práctica $S_p < p$ tenemos que $0 < E < 1$, dependiendo de la eficacia con la cual son usados los procesadores. Matemáticamente la `eficiencia` es

$$E = \frac{S_p}{p} = \frac{T_1}{p \cdot T_p}$$

2.4.3. Escalabilidad

Con frecuencia los algoritmos diseñados se prueban en instancias relativamente pequeñas y usando pocos procesadores. Sin embargo, estos algoritmos intentan resolver problemas mucho más grandes y en equipos con un gran número de procesadores. Mientras que el desarrollo de código se ve simplificado al usar versiones reducidas del problema y de la máquina su desempeño y exactitud son mucho más difíciles de determinar. Se dice que un sistema paralelo es escalable si es capaz de usar de forma satisfactoria un número creciente de procesadores. Se utilizarán las métricas de `speedup` y `eficiencia` para medir el desempeño de los algortimos desarrollados.

Capítulo 3

Análisis de resultados

3.1. Algoritmos

Los algoritmos en paralelo diseñados e implementados para resolver o aproximar el problema propuesto son recursivos, tanto el determinístico como el heurístico. La base de la recursión es el caso $m = 1$, el cual consiste en resolver el siguiente problema:

$$\begin{aligned} \text{se minimize} \quad & b && && (3.1)\\ \text{sujeto a} \quad & s_1 + \cdots + s_n = c && && (3.2)\\ & s_i \cdot b \geq d_i && i = 1, \ldots, n, && (3.3)\\ & b \in \mathbb{N} && && (3.4)\\ & s_i \in \{1, \ldots, c\} && && (3.5) \end{aligned}$$

El algoritmo UnRecipiente resuelve el problema de manera óptima. Toma como entrada el número n de objetos distintos, la capacidad c del recipiente, y el vector de requerimientos para cada uno de los objetos $\mathbf{d} = [d_1, d_2, \ldots, d_n]$, tal que $c \geq n$. La salida del algoritmo es la tupla $(b^*, \mathbf{s} = [s_1, s_2, \ldots, s_n])$, donde b^* es la solución óptima para el problema y $\mathbf{s}$ determina el número de repeticiones de cada objeto en el recipiente.

Algoritmo UnRecipiente $(n, c, \mathbf{d})$

1 $s_i \leftarrow 1$ **para** $i \leftarrow 1, \ldots, n$
2 **para** $c' \leftarrow n + 1$ **hasta** c:
3 **encuentra** j **tal que** $\lceil d_j/s_j \rceil = \text{máx}\{\lceil d_i/s_i \rceil \mid i = 1, \ldots, n\}$
4 $s_j \leftarrow s_j + 1$
5 $b^* \leftarrow \text{máx}\{\lceil d_i/s_i \rceil \mid i = 1, \ldots, n\}$
6 **regresa** $(b^*, \mathbf{s})$

El análisis del tiempo de ejecución del Algoritmo UnRecipiente consiste en el análisis del ciclo de la instrucción 2, el cual se ejecuta $c - n$ veces; dentro de dicho ciclo, encontrar el índice j requerido en la instrucción 3 toma tiempo n; lo demás toma tiempo constante. Por lo tanto, el tiempo de ejecución del Algoritmo UnRecipiente es de $\Theta(n(c - n))$. Dicho algoritmo es un algoritmo *avaro* (greedy), muy eficiente, por lo que no se consideró necesario diseñar e implementar una paralelización del mismo.

En ambos casos (tanto para el heurístico como para el determinístico), solamente la primera recursión se resuelve en paralelo mediante la técnica de descomposición exploratoria; el resto de casos se resuelve de manera secuencial, pues el trabajo ya se está realizando en paralelo en cada uno de los procesadores. Por lo que se presenta primero la solución secuencial para cada uno de los problemas. Para el heurístico se tienen dos algoritmos secuenciales, ambos comparten la misma filosofía, pero en uno de ellos (HEUR-FULLSOLVE) se analizan todos los casos posibles, mientras que en el otro (HEUR-SOLVE) solo se exploran de forma adecuada algunos de ellos.

Algoritmo HEUR-SOLVE $(n, m, c, \mathbf{d})$

```
1  si m = 1:
2      regresa UnRecipiente (n, c, d)
3  determinar x_mín y x_máx (rango de la búsqueda)
4  sea q' = [q'_1, ..., q'_{n×c}] donde q'_{i+(j−1)·n} ← ⌈d_i/j⌉; i = 1, ..., n; j = 1, ..., c
5  sea q = [q_1, ..., q_k] ⊂ q' donde x_mín ≤ q_i ≤ x_máx; i = 1, ..., k
6  x* ← x_mín · m
7  para algunos q_i ∈ q
8      b_1 ← q_i
9      para algunos s^1 = [s_1, ..., s_n]^T; s_i = 0, ..., mín(c, ⌈d_i/x_mín⌉); s_1 + ··· + s_n = c
10         sea d' = [d_1, ..., d_{n'}] donde (d'_{i'} ← d_i − s_i · b_1) > 0
11         (b', S') ← Heur-Solve (n', m − 1, c, d')
12         si b_1 + b'_1 + ··· + b'_{m−1} < x*:
13             actualizar x*, b*, S*
14 regresa (b*, S*)
```

Algoritmo HEUR-FULLSOLVE $(n, m, c, \mathbf{d})$

```
1  si m = 1:
2      regresa UnRecipiente (n, c, d)
3  determinar x_mín y x_máx
4  x* ← x_mín · m
5  para cada s^1 = [s_1, ..., s_n]^T; s_i = 0, ..., mín(c, ⌈d_i/x_mín⌉); s_1 + ··· + s_n = c
6      para i ← 1 hasta n:
7          b_1 ← ((t_1 > 0)  ?  ⌈d_1/t_1⌉ : 0)
8          si x_mín ≤ b_1 ≤ x_máx:
9              sea d' = [d_1, ..., d_{n'}] donde (d'_{i'} ← d_i − s_i · b_1) > 0
10             (b', S') ← Heur-FullSolve (n', m − 1, c, d')
11             si b_1 + b'_1 + ··· + b'_{m−1} < x*:
12                 actualizar x*, b*, S*
13 regresa (b*, S*)
```

En el algoritmo determinístico secuencial (DETERMINISTICO) se utiliza el heurístico HEUR-SOLVE, que es más rápido pero cuya solución por lo general está más lejana de la óptima.

Para la paralelización del heurístico se utilizó como base HEUR-FULLSOLVE, que se ejecuta de

Algoritmo DETERMINISTICO $(n, m, c, \mathbf{d})$

si $m = 1$:
    regresa UNRECIPIENTE $(n, c, \mathbf{d})$
determinar $x_{\text{mín}}$ y $x_{\text{máx}}$
$(\mathbf{b}^*, S^*) \leftarrow$ HEUR-SOLVE $(n, m, c, \mathbf{d})$
$x^* = b_1^* + \cdots + b_n^*$
para cada $\mathbf{s}^1 = [s_1, \ldots, s_n]^T$; $s_i = 0, \ldots,$ mín$(c, \lceil d_i / x_{\text{mín}} \rceil)$; $s_1 + \cdots + s_n = c$
    $b_1 \leftarrow x_{\text{máx}}$
    mientras $b_1 \geq x_{\text{mín}}$
        sea $\mathbf{d}' = [d_1, \ldots, d_{n'}]$ **donde** $(d'_{i'} \leftarrow d_i - s_i \cdot b_1) > 0$
        $(\mathbf{b}', S') \leftarrow$ DETERMINISTICO $(n', m-1, c, \mathbf{d}')$
        $t \leftarrow b_1 + b'_1 + \cdots + b'_{m-1}$
        si $t < x^*$:
            actualizar $x^*, \mathbf{b}^*, S^*$
        $\Delta \leftarrow t - x^*$
        $b_1 \leftarrow b_1 - \Delta - 1$
regresa $(\mathbf{b}^*, S^*)$

manera recursiva. La directiva `#pragma` se utilizó en el pseudocódigo para mostrar construcciones especiales en paralelo. La directiva `#pragma omp parallel` indica el inicio de un bloque en paralelo, mientras que la directiva `#pragma omp critical` indica una sección de código que solamente puede ser ejecutada por uno de los procesadores a la vez.

Lo fundamental en el diseño de un algoritmo en paralelo es realizar una distribución eficiente del trabajo. En este caso, lo que se busca es repartir de manera eficiente las distintas permutaciones que se deben analizar entre los diferentes procesadores; para ello, lo mejor es que la repartición de permutaciones se realice de manera global por medio de una función. Cada vez que un procesador finaliza su tarea solicita a esta función una nueva permutación para continuar trabajando, esto hace que este proceso deba sincronizarse lo cual puede generar un efecto de embudo en la llamada a la función que asigna las permutaciones, sobre todo para instancias pequeñas del problema.

En el caso de guardar la solución óptima, y para evitar el problema del cuello de botella mencionado, el algoritmo se implementó de manera que cada procesador guarda el óptimo local de la solución correspondiente a cada una de las permutaciones que exploró. Luego, al final, se utiliza nuevamente la directiva `#pragma omp critical`, para escoger el óptimo global de entre todos los óptimos locales hallados por cada procesador.

El algoritmo determinístico en paralelo, igual que en el caso heurístico, utiliza la directiva `#pragma omp critical` para obtener la permutación respectiva. Pero en este caso, como el trabajo que se realiza depende del óptimo que se tenga (mientras mejor sea el óptimo, menos trabajo se realiza), es mejor siempre tener un óptimo de manera global. Por ello el óptimo se define afuera de la directiva `#pragma omp parallel`, por lo que es un valor global. Pero entonces, para actualizarlo, se requiere de la directiva `#pragma omp critical`. Esto podría representar una desventaja por el efecto de embudo comentado anteriormente, pero esto es más probable que se presente para instancias pequeñas del problema, y/o con muchos procesadores.

Algoritmo HEUR-PARALELO $(n, m, c, \mathbf{d})$

```
1  determinar x_mín y x_máx
2  #pragma omp parallel
3       x* ← x_mín · m
4       #pragma omp critical
5            obtener s^1 = [s_1, ..., s_n]^T; s_i = 0, ..., mín(c, ⌈d_i/x_mín⌉); s_1 + ··· + s_n = c
6       hasta agotar las permutaciones haga:
7            para i ← 1 hasta n:
8                 b_1 ← ((t_1 > 0)  ?  ⌈d_1/t_1⌉ : 0)
9                 si x_mín ≤ b_1 ≤ x_máx:
10                     sea d' = [d_1, ..., d_n'] donde (d'_i' ← d_i − s_i · b_1) > 0
11                     (b', S') ← HEUR-FULLSOLVE (n', m − 1, c, d')
12                     si b_1 + b'_1 + ··· + b'_{m−1} < x*:
13                          actualizar x*, b*, S*
14           #pragma omp critical
15                obtener s^1 = [s_1, ..., s_n]^T; s_i = 0, ..., mín(c, ⌈d_i/x_mín⌉); s_1 + ··· + s_n = c
16      #pragma omp critical
17           escoger la mejor solucion (b*, S*)
18 regresa (b*, S*)
```

Algoritmo DETERMINISTICO-PARALELO $(n, m, c, \mathbf{d})$

```
1  determinar x_mín y x_máx
2  (b*, S*) ← HEUR-PARALELO (n, m, c, d)
3  x* ← b*_1 + ··· + b*_m
4  #pragma omp parallel
5       #pragma omp critical
6            obtener s^1 = [s_1, ..., s_n]^T; s_i = 0, ..., mín(c, ⌈d_i/x_mín⌉); s_1 + ··· + s_n = c
7       hasta agotar las permutaciones haga:
8            b_1 ← x_máx
9            mientras b_1 ≥ x_mín
10                sea d' = [d_1, ..., d_n'] donde (d'_i' ← d_i − s_i · b_1) > 0
11                (b', S') ← DETERMINISTICO (n', m − 1, c, d')
12                t ← b_1 + b'_1 + ··· + b'_{m−1}
13                #pragma omp critical
14                     si t < x*:
15                          actualizar x*, b*, S*
16                Δ ← t − x*
17                b_1 ← b_1 − Δ − 1
18           #pragma omp critical
19                obtener s^1 = [s_1, ..., s_n]^T; s_i = 0, ..., mín(c, ⌈d_i/x_mín⌉); s_1 + ··· + s_n = c
20 regresa (b*, S*)
```

3.2. Experimentos

Se construyó un conjunto de instancias de prueba, algunas de las cuales se muestran, junto con su respectiva solución, en el apéndice A. Cada instancia fue ejecutada con los algoritmos Heur-Paralelo y Deterministico-Paralelo en un computador con las siguientes características: 24 GB de memoria RAM DDR3 1333MHz y 2 procesadores Intel 6-Core Xeon X5650 (2.66 GHz Nehalem) cada uno con 12 MB de memoria caché L3.

Como se comentó anteriormente en el análisis del problema, la complejidad del mismo depende de 3 parámetros: el número n de demandas, el número m de tipos recipientes, y de c, que es la capacidad de cada recipiente.

Para analizar el comportamiento de los algoritmos en paralelo es importante analizar instancias del problema de diversos tamaños, por lo que se consideró analizar cada algoritmo, variando solamente uno de los 3 parámetros en cada caso.

Para medir el desempeño de los algoritmos se utilizaron las siguientes métricas: el speedup relativo y la eficiencia, con 1 hasta 12 procesadores.

Se consideró importante mostrar los tiempos de ejecución para mostrar el comportamiento del problema con respecto a la variación de los parámetros (n, m y c), y para complementar el análisis de las métricas utilizadas (speedup relativo y eficiencia).

Para los gráficos del speedup relativo, se ha incluido la recta $y = x$, que indica el comportamiento ideal de un algoritmo en paralelo. Lo usual es que los resultados estén por debajo de la recta, pero en algunos casos, como ya se mencionó, podría suceder que los resultados estén por encima de la recta, lo cual se conoce como *super speedup*.

De igual manera, para los gráficos de eficiencia, se ha incluido la recta $y = 1$, que representa el comportamiento ideal de un algoritmo en paralelo.

Figura 3.1: Resultados de tiempos en t procesadores al resolver un problema con parámetros $n = 8$, $m = 4$, y $c = 3, 4, 5, 7$, con el Algoritmo DETERMINISTICO-PARALELO.

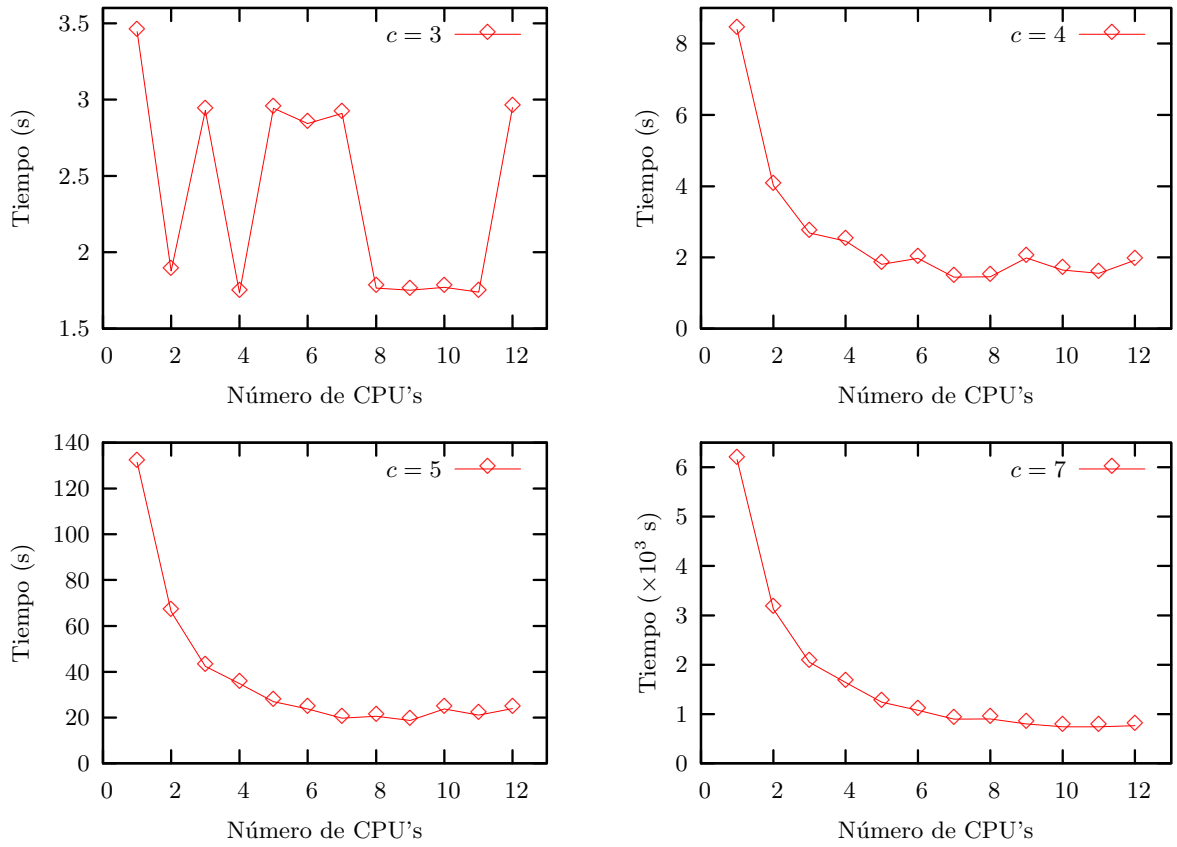


Figura 3.2: Speedup en el uso de t procesadores al resolver un problema con parámetros $n = 8$, $m = 4$, y $c = 3, 4, 5, 7$ con el Algoritmo DETERMINISTICO-PARALELO.

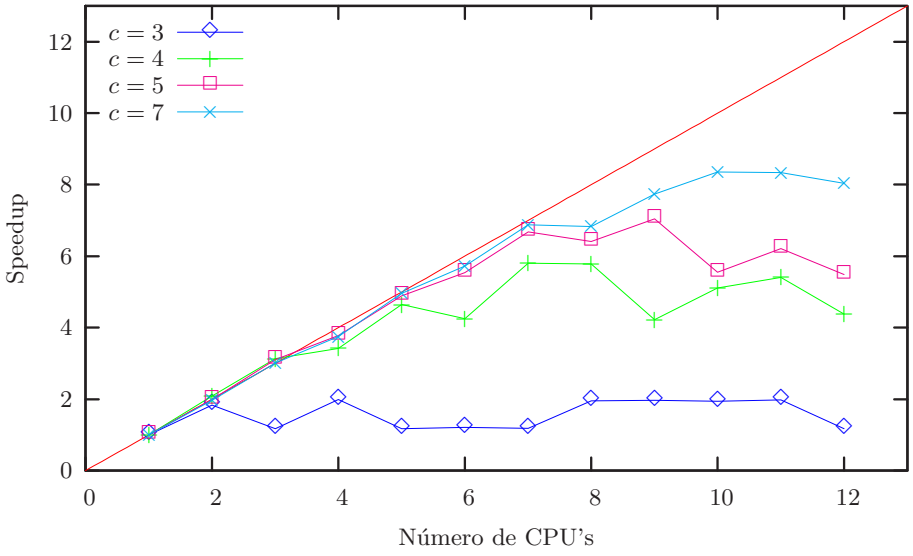


Figura 3.3: Eficiencia en el uso de t procesadores al resolver un problema con parámetros $n = 8$, $m = 4$, y $c = 3, 4, 5, 7$ con el Algoritmo DETERMINISTICO-PARALELO.

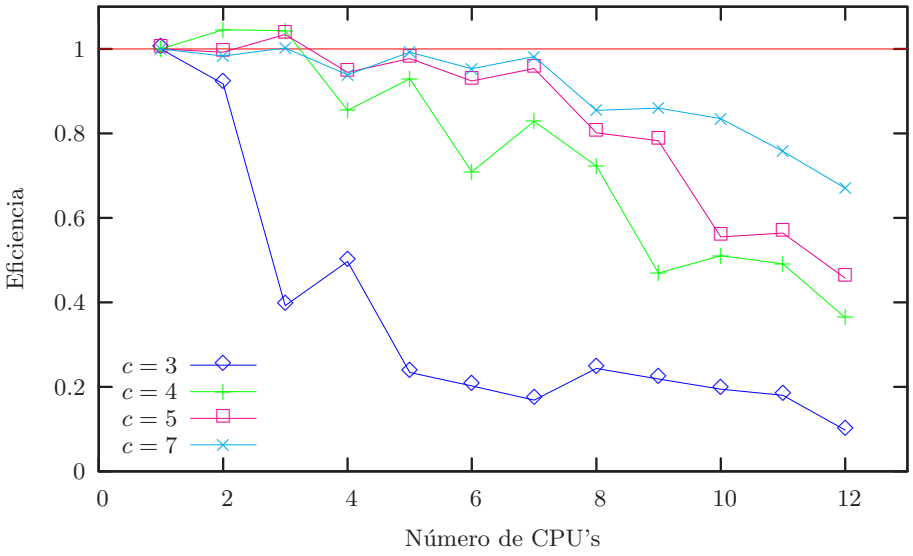


Figura 3.4: Resultados de tiempos en t procesadores al resolver un problema con parámetros $n = 8$, $m = 5$, y $c = 4, 5, 6, 8$ con el Algoritmo HEUR-PARALELO.

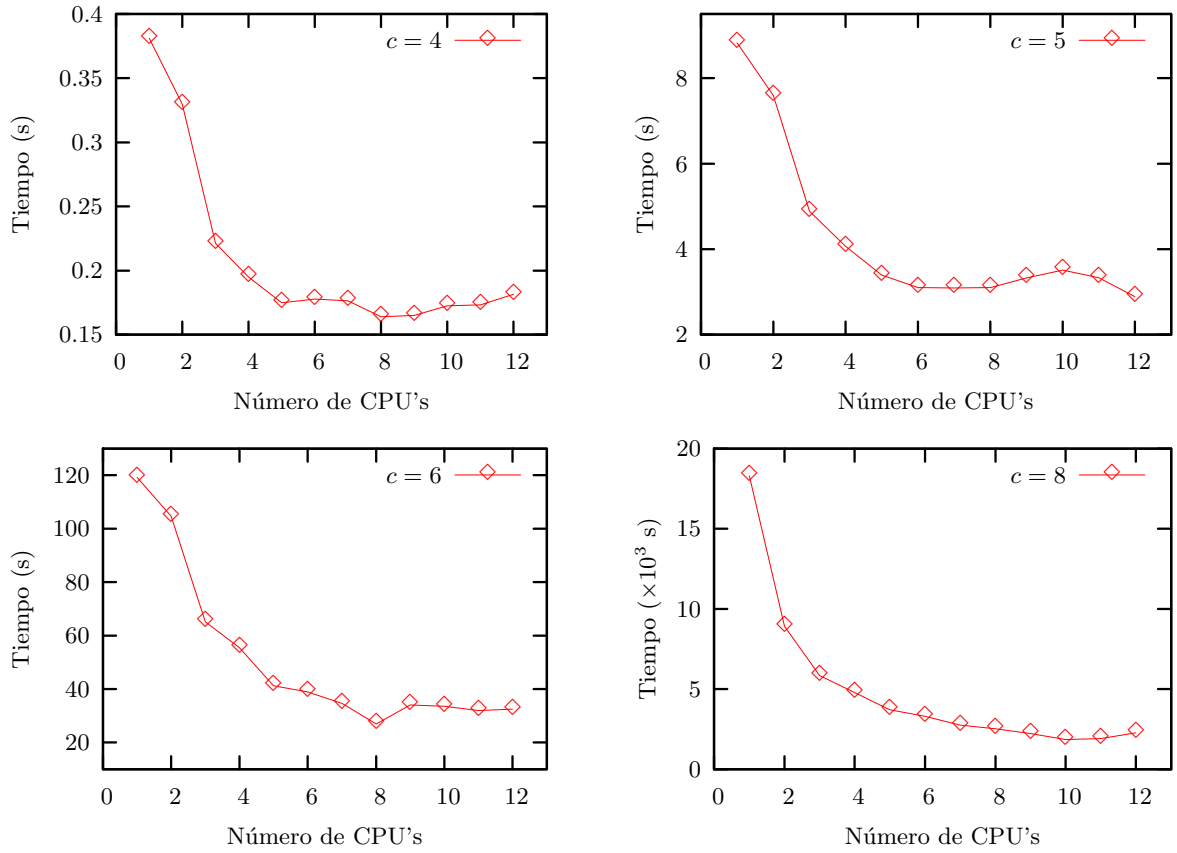


Figura 3.5: Speedup en el uso de t procesadores en la resolución de un problema con parámetros $n = 8$, $m = 5$, y $c = 4, 5, 6, 8$ con el Algoritmo Heur-Paralelo.

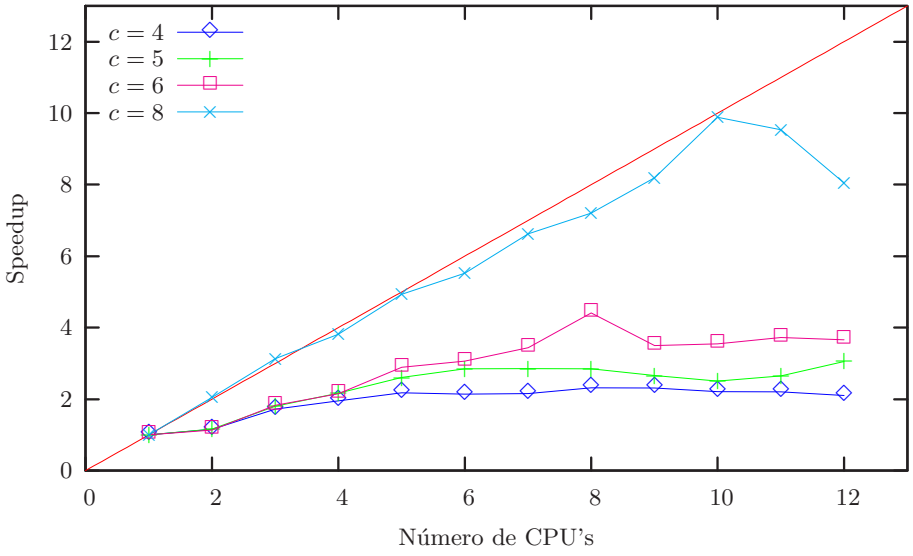


Figura 3.6: Eficiencia en el uso de t procesadores en la resolución de un problema con parámetros $n = 8$, $m = 5$, y $c = 4, 5, 6, 8$ con el Algoritmo Heur-Paralelo.

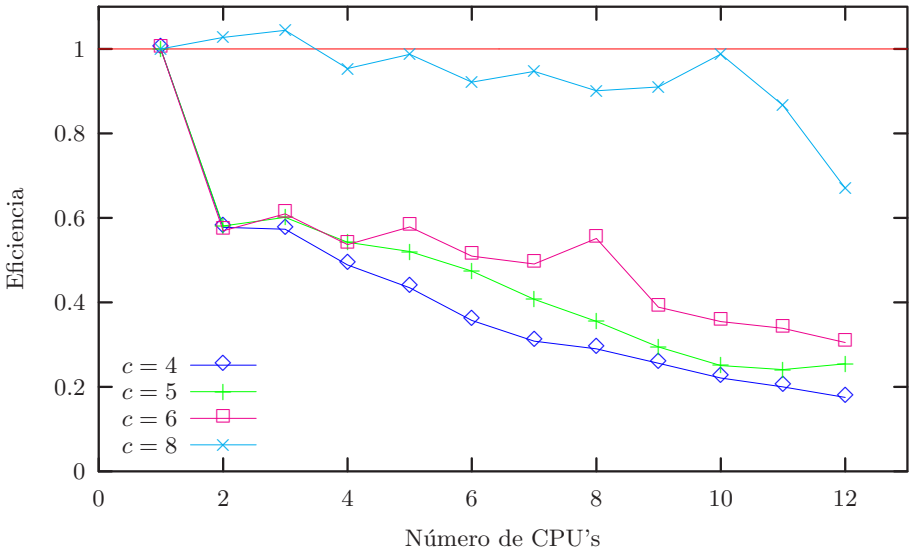


Figura 3.7: Resultados de tiempos en t procesadores al resolver un problema con parámetros $n = 4, 8, 12, 16$, $m = 3$, y $c = 10$ con el Algoritmo DETERMINISTICO-PARALELO.

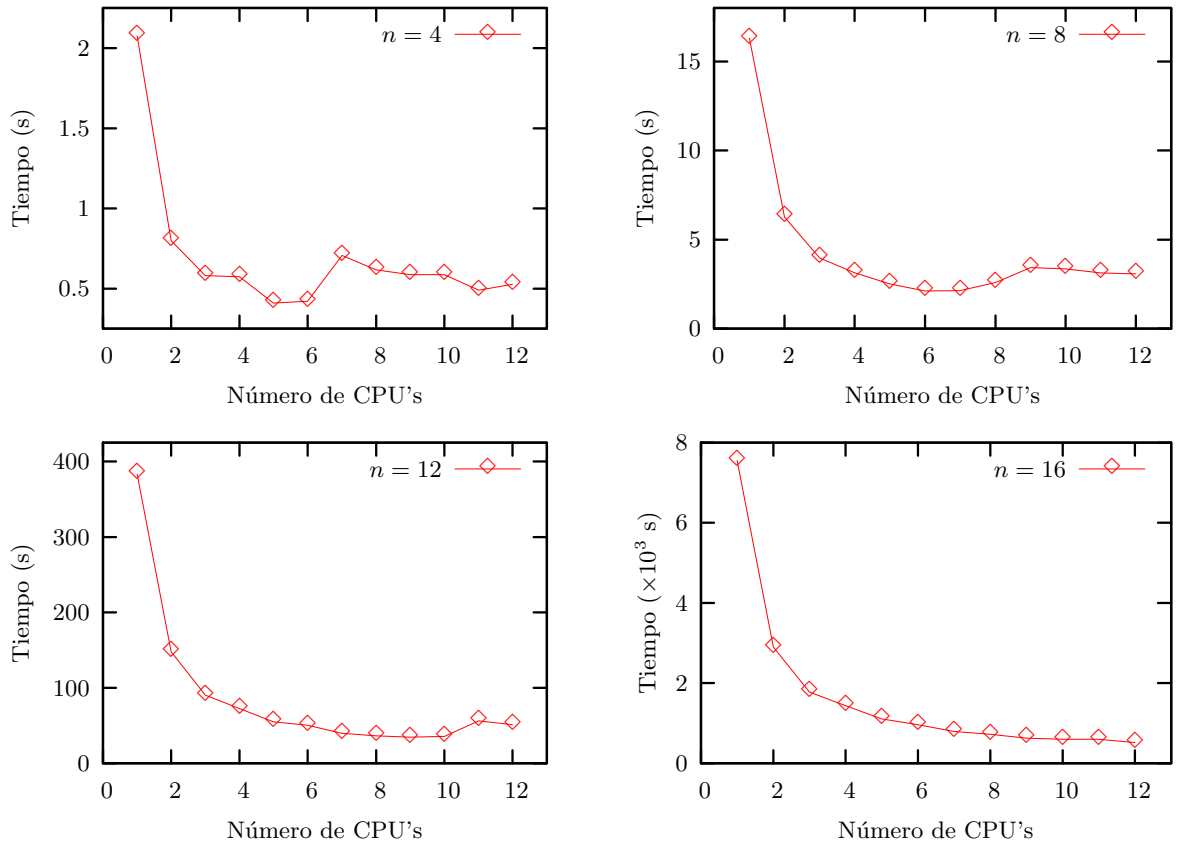


Figura 3.8: Speedup en el uso de t procesadores en la resolución de un problema con parámetros $n = 4, 8, 12, 16$, $m = 3$, y $c = 10$ con el Algoritmo DETERMINISTICO-PARALELO.

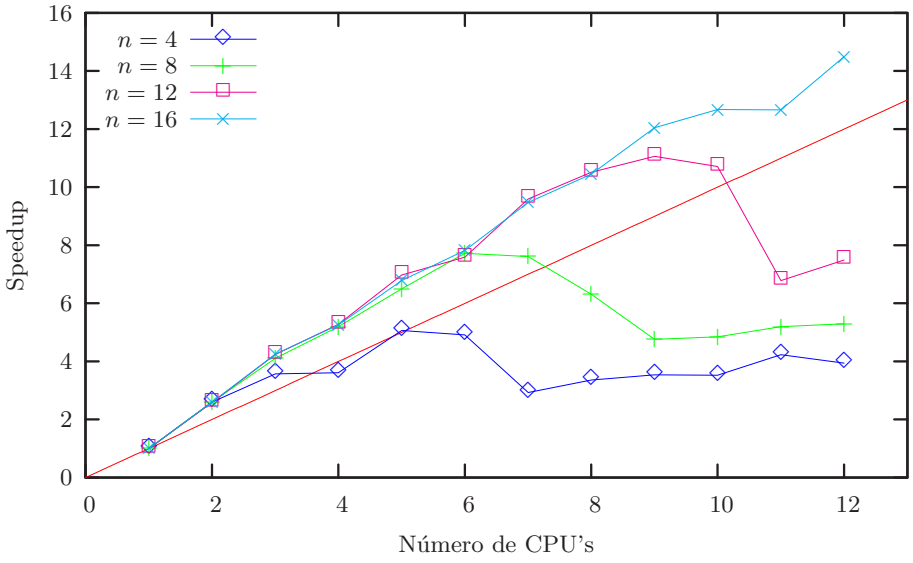


Figura 3.9: Eficiencia en el uso de t procesadores en la resolución de un problema con parámetros $n = 4, 8, 12, 16$, $m = 3$, y $c = 10$ con el Algoritmo DETERMINISTICO-PARALELO.

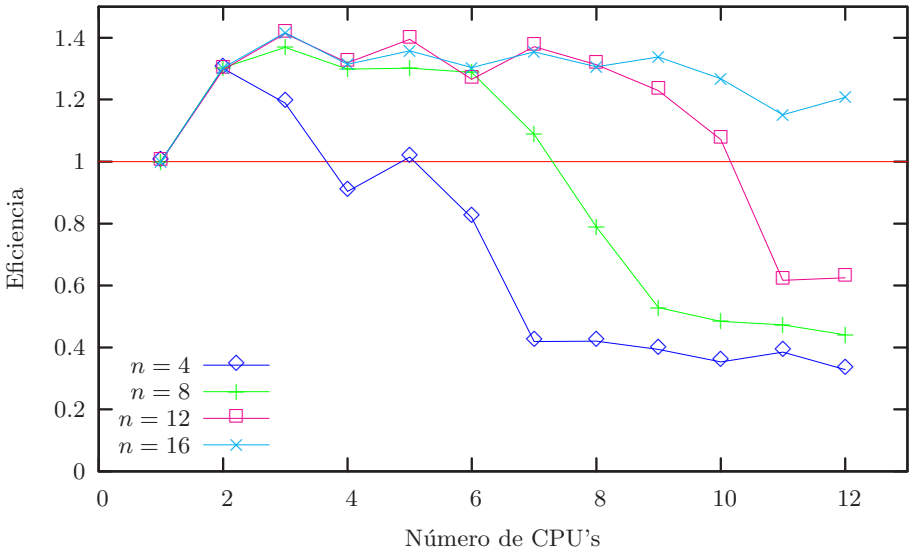


Figura 3.10: Resultados de tiempos en t procesadores al resolver un problema con parámetros $n = 4, 8, 12, 16$, $m = 3$, y $c = 10$ con el Algoritmo HEUR-PARALELO.

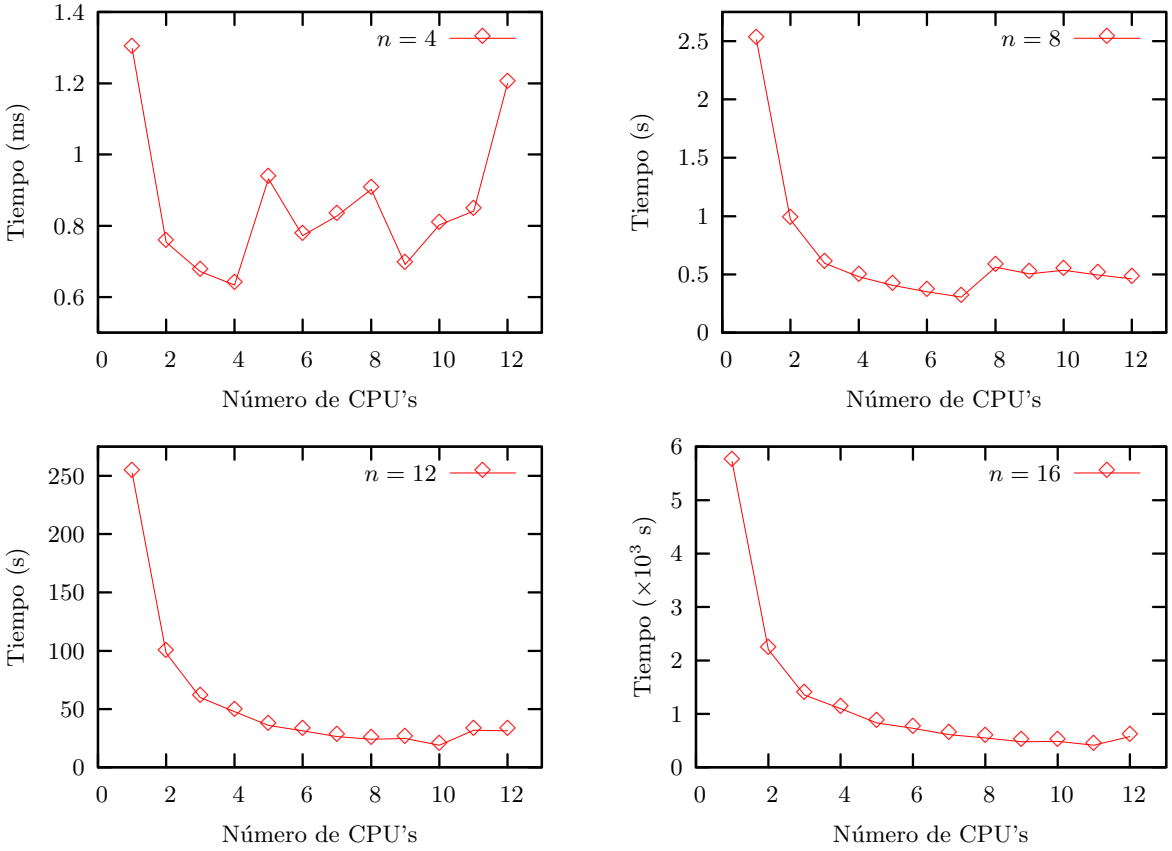


Figura 3.11: Speedup en el uso de t procesadores en la resolución de un problema con parámetros $n = 4, 8, 12, 16$, $m = 3$, y $c = 10$ con el Algoritmo HEUR-PARALELO.

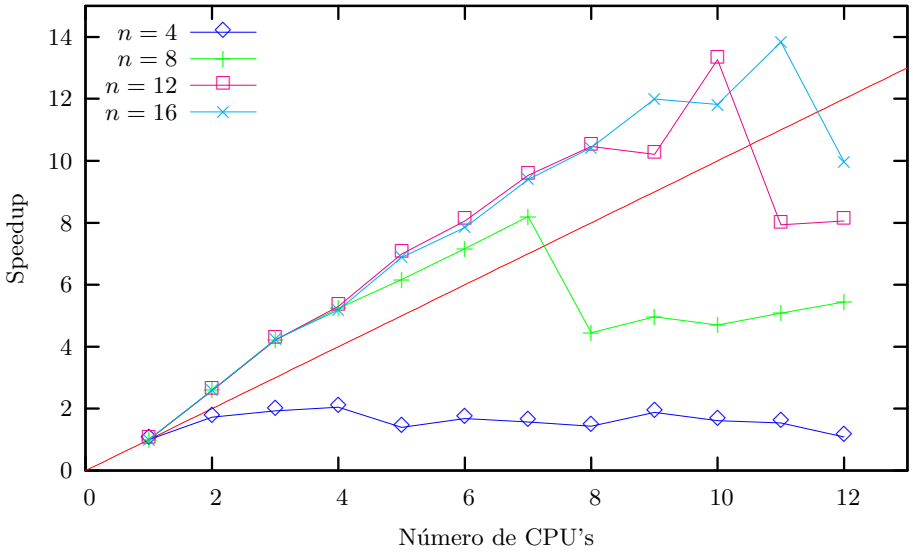


Figura 3.12: Eficiencia en el uso de t procesadores en la resolución de un problema con parámetros $n = 4, 8, 12, 16$, $m = 3$, y $c = 10$ con el Algoritmo HEUR-PARALELO.

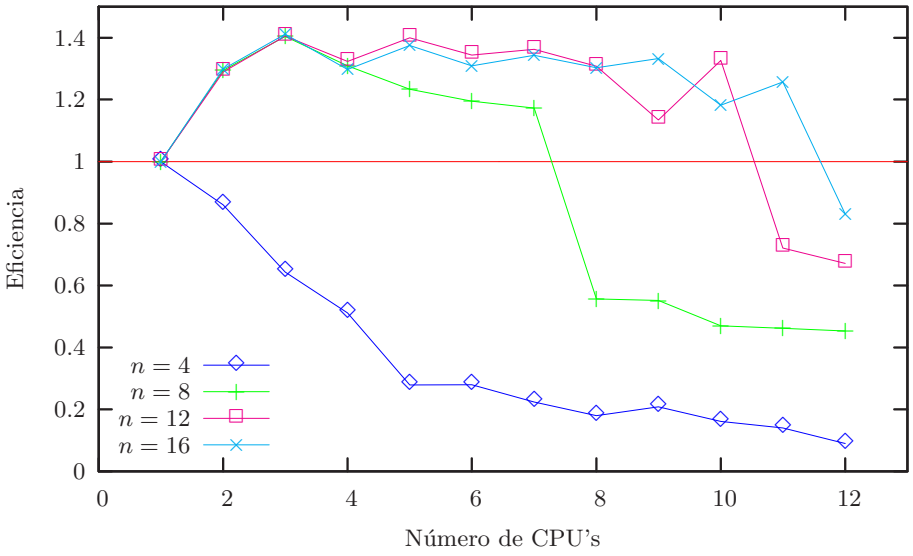


Figura 3.13: Resultados de tiempos en t procesadores al resolver un problema con parámetros $n = 8$, $m = 2, 3, 4$, y $c = 8$ con el Algoritmo DETERMINISTICO-PARALELO.

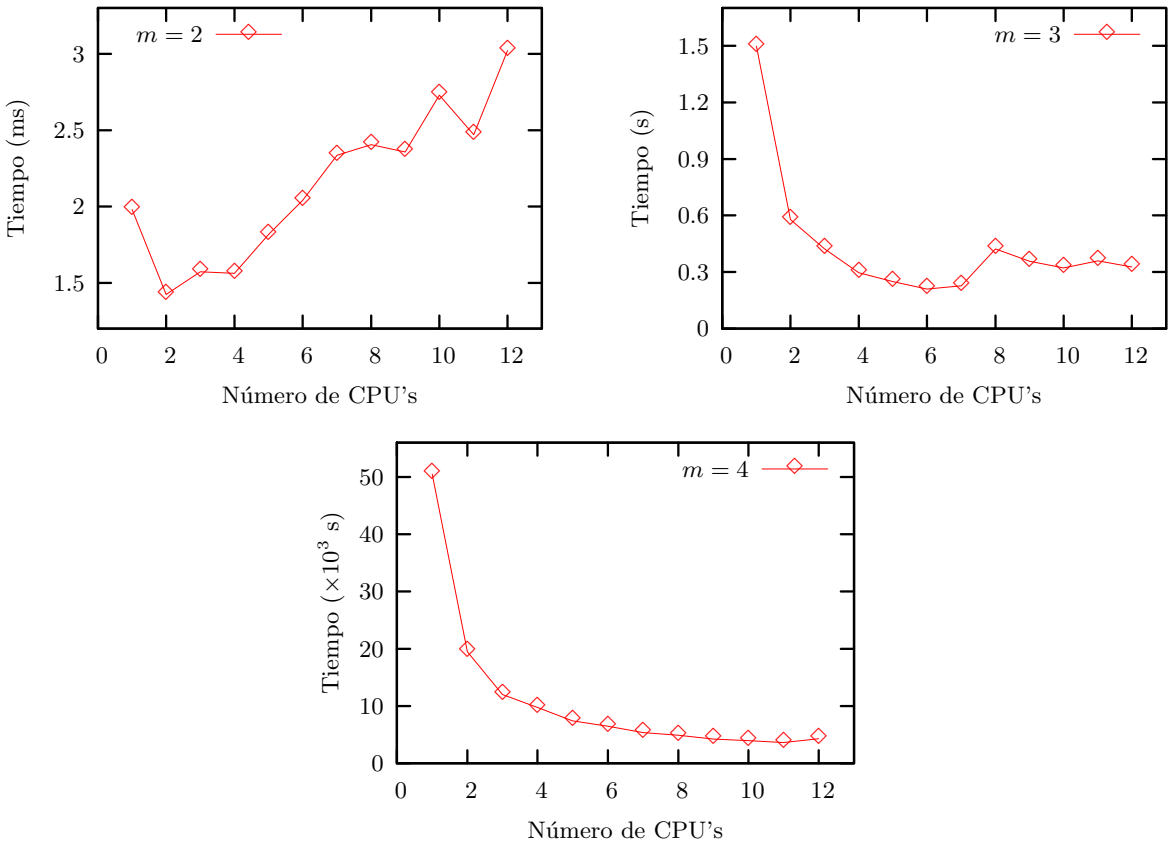


Figura 3.14: Speedup en el uso de t procesadores en la resolución de un problema con parámetros $n = 8$, $m = 2, 3, 4$, y $c = 8$ con el Algoritmo DETERMINISTICO-PARALELO.

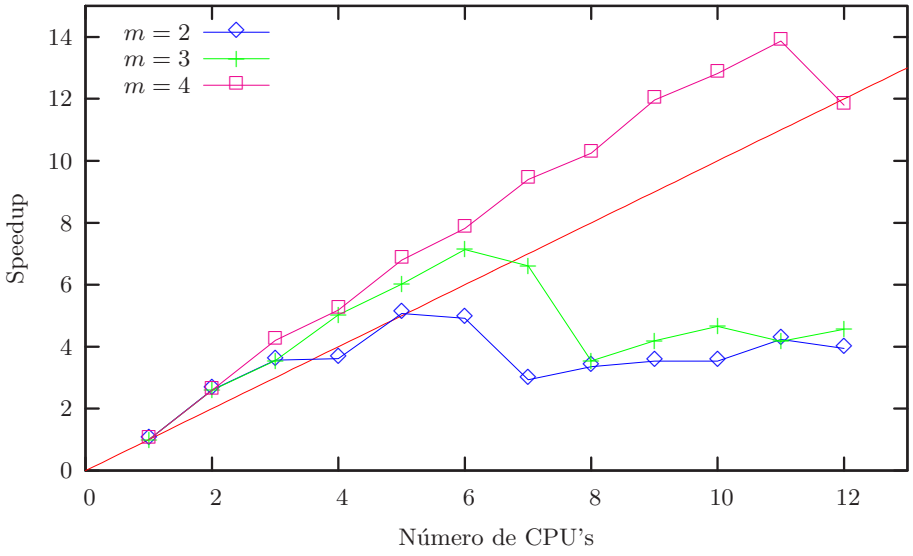


Figura 3.15: Eficiencia en el uso de t procesadores en la resolución de un problema con parámetros $n = 8$, $m = 2, 3, 4$, y $c = 8$ con el Algoritmo DETERMINISTICO-PARALELO.

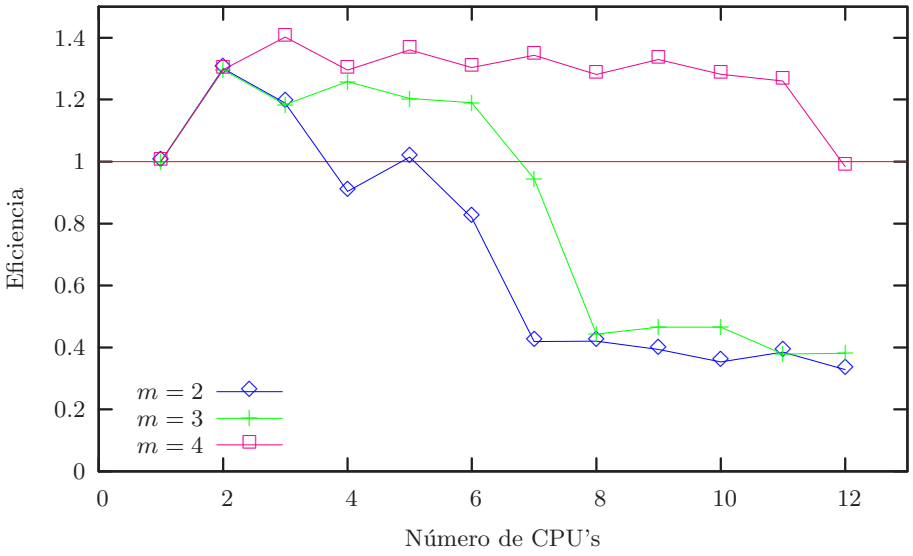


Figura 3.16: Resultados de tiempos en t procesadores al resolver un problema con parámetros $n = 8$, $m = 2, 3, 4, 5$, y $c = 8$ con el Algoritmo HEUR-PARALELO.

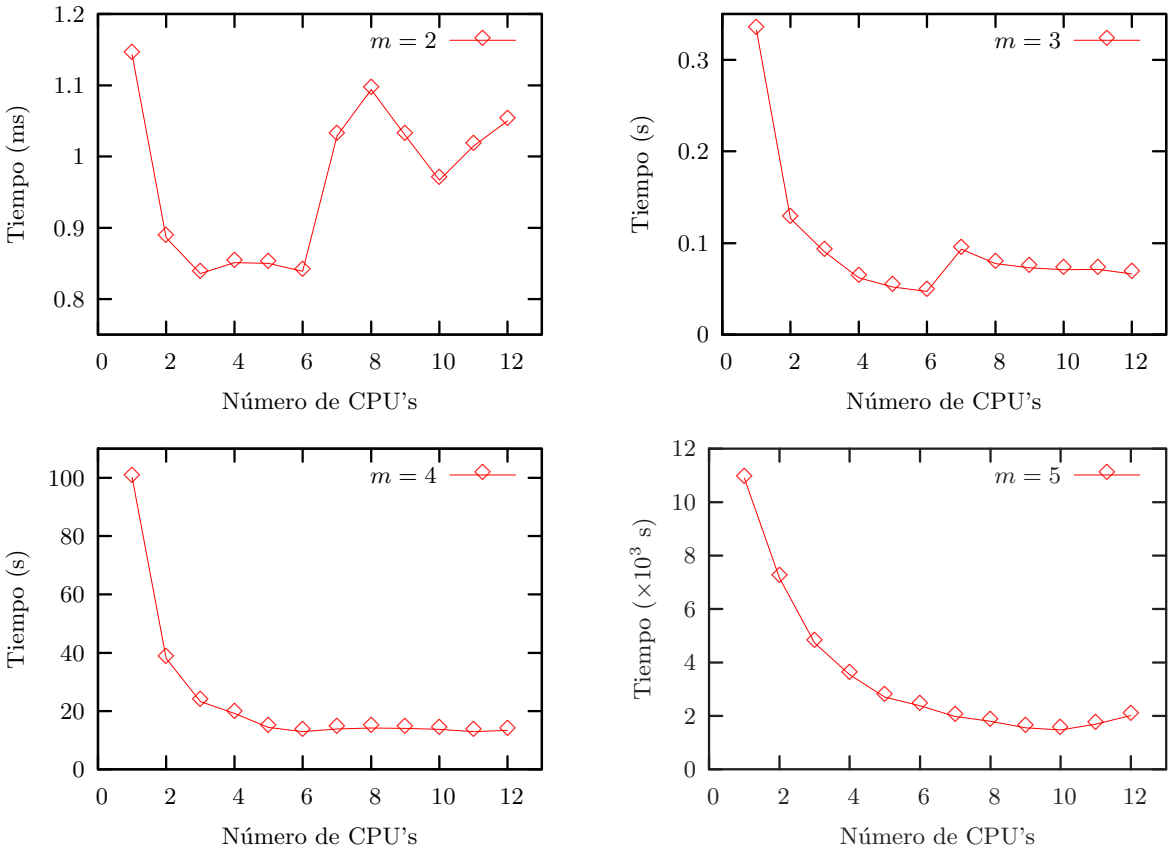


Figura 3.17: Speedup en el uso de t procesadores en la resolución de un problema con parámetros $n = 8$, $m = 2, 3, 4, 5$, y $c = 8$ con el Algoritmo HEUR-PARALELO.

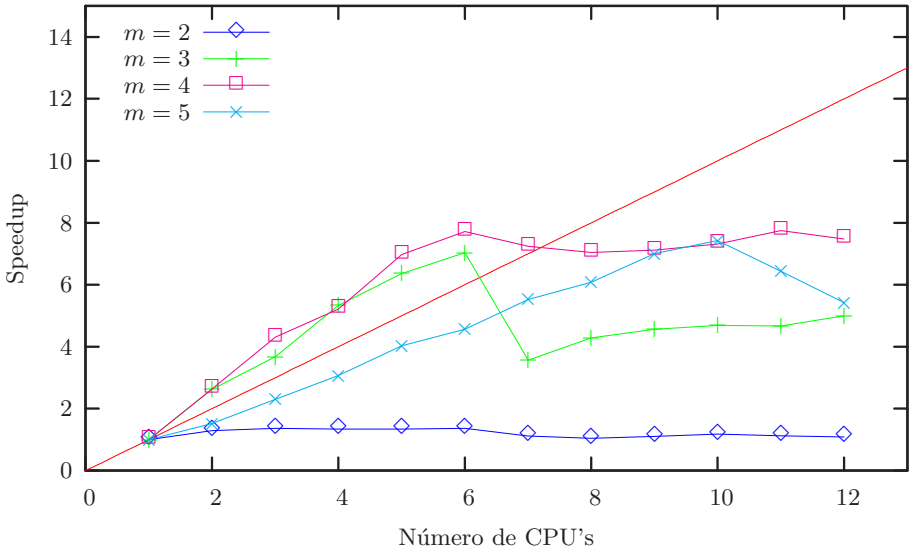
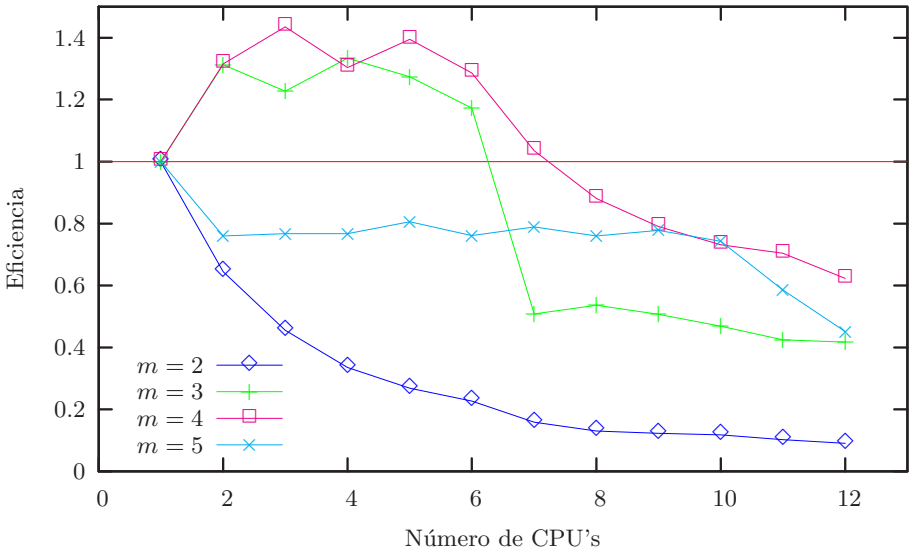


Figura 3.18: Eficiencia en el uso de t procesadores en la resolución de un problema con parámetros $n = 8$, $m = 2, 3, 4, 5$, y $c = 8$ con el Algoritmo HEUR-PARALELO.



3.3. Discusión de resultados

Al momento de analizar los resultados utilizando los tiempos de ejecución de un algoritmo en paralelo, se debe tomar en cuenta que requiere considerar no solamente el algoritmo, sino además el sistema o el equipo en el que son ejecutados. Esto hace que pueden variar de un equipo a otro debido a la configuración del equipo; pero incluso, se pueden obtener variaciones en las medidas de los tiempos en un mismo equipo, debido a la administración de memoria y de procesos que realice el equipo en el momento dado.

De los experimentos realizados (Figuras 3.1, 3.4, 3.7, 3.10, 3.13, 3.16), el tiempo de ejecución de ambos algoritmos heurístico y determinístico, crecen de manera exponencial al variar cualquiera de los parámetros n, m y c. Este resultado es de esperarse, pues la dimensión del espacio solución crece al aumentar cualquiera de estos parámetros (Ecuación 1.7).

La eficiencia mide el uso que se hace de los procesadores. Los gráficos de eficiencia muestran que los algoritmos son capaces de distribuir el trabajo en forma adecuada. De los experimentos realizados (Figuras 3.3, 3.6, 3.9, 3.12, 3.15) se observa que los algoritmos diseñados son escalables, en el sentido de que, a mayor tamaño del problema, se hace un uso más eficiente de los procesadores.

Se observa que para instancias pequeñas (Figuras 3.1, 3.10, 3.13, 3.16), el uso de un mayor número de procesadores afecta de manera negativa el desempeño del algoritmo, lo cual es natural, porque el acceso crítico a variables compartidas obliga a los procesadores a estar mayor tiempo ociosos, debido a la sincronazación de procesos que debe hacerse; no así para instancias grandes del problema, donde el uso de los procesadores es eficiente.

3.4. Conclusiones

Los resultados muestran que fue posible realizar un diseño en paralelo para resolver el problema propuesto, tanto de manera determinística como heurística.

El modelo exploratorio para programación en paralelo fue utilizado en el proceso de diseño de los algoritmos para la solución del problema planteado, y la biblioteca OpenMP mostró ser adecuada para la implementación de los algoritmos diseñados.

Los experimentos realizados evidenciaron que el diseño e implementación de los algoritmos en paralelo son escalables y eficientes para resolver instancias del problema propuesto.

Los resultados sugieren que el algoritmo heurístico puede rediseñarse para mejorar su eficiencia según los distintos valores de n, m ó c. Es decir, dependiendo de las magnitudes de los distintos valores, el algoritmo puede reorientarse para realizar una búsqueda más efectiva sobre el espacio solución.

En general, como se puede ver en los experimentos, el algoritmo diseñado permite la solución de un problema al menos $8x$ más rápido que el algoritmo secuencial asociado; y en varios casos, hasta $10x$ más rápido.

Apéndice A

Instancias de prueba

Se presentarán las instancias (junto con su respectiva solución) de algunos de los problemas utilizados. Para escribir la solución encontrada, se utilizará la notación introducida en el Cuadro 1.1. En algunos casos, el algoritmo heurístico encontró el óptimo, por lo que se presentará únicamente la solución óptima.

A.1. Problema 1

Valores de los parámetros: $n = 8$; $m = 4$; $c = 7$.
Solución del heurístico y determinístico:

	28128	20538	16193	6609	71468
77362	1	0	1	5	4
84383	3	0	0	0	1
61612	0	3	0	0	2
16193	0	0	1	0	0
89653	0	2	3	0	2
79044	2	0	1	1	14
57246	0	2	1	0	23
34722	1	0	0	1	15

A.2. Problema 2

Valores de los parámetros : $n = 8;\ m = 5;\ c = 8$.
Solución del heurístico y determinístico:

	21096	16193	14613	8667	1959	62528
77362	0	1	3	2	0	4
84383	4	0	0	0	0	1
61612	0	2	2	0	0	0
16193	0	1	0	0	0	0
89653	3	0	1	0	6	2
79044	1	0	1	5	0	0
57246	0	3	0	1	0	0
34722	0	1	1	0	2	2

A.3. Problema 3

Valores de los parámetros : $n = 8;\ m = 3;\ c = 10$.
Solución del heurístico:

	26348	15449	8479	50276
77362	2	0	3	771
84383	2	1	2	720
61612	0	4	0	184
16193	0	0	2	765
89653	1	3	2	0
79044	3	0	0	0
57246	1	2	0	0
34722	1	0	1	105

Solución del determinístico:

	28817	17361	4049	50227
77362	0	4	2	180
84383	1	3	1	566
61612	2	0	1	71
16193	0	0	4	3
89653	3	0	1	847
79044	2	1	1	0
57246	2	0	0	388
34722	0	2	0	0

A.4. Problema 4

Valores de los parámetros : $n = 8;\ m = 4;\ c = 8$.
Solución del heurístico:

	22414	16193	15259	8681	62547
77362	2	0	1	2	87
84383	1	1	3	0	1
61612	1	0	2	1	1
16193	0	1	0	0	0
89653	4	0	0	0	3
79044	0	3	2	0	53
57246	0	3	0	1	14
34722	0	0	0	4	2

Solución del determinístico:

	20538	18529	16193	7281	62541
77362	0	3	0	3	68
84383	1	1	1	4	1
61612	3	0	0	0	2
16193	0	0	1	0	0
89653	2	0	3	0	2
79044	0	3	1	1	17
57246	2	0	1	0	23
34722	0	1	1	0	0

Bibliografía

[1] Alba Enrique. Parallel Metaheuristics: A New Class of Algorithms. John Wiley & Sons, New Jersey, 2005.

[2] Pacheco, Peter. Parallel Programming with `MPI`. Morgan Kaufmann Publishers, San Francisco, 1997.

[3] Carrera, Luis-Ernesto. *Algoritmo para encontrar el óptimo a una simplificación de un problema combinatorio presente en la industria editorial*. Tesis para optar por el grado de MSc. en Matemática Computacional, CINVESTAV, México, D.F, 2006.

[4] Carrera, L. y Figueroa, G. *Un problema tipo bin-packing*. Tecnología en Marcha 24(2), 2011.

[5] Carrera, L. y Figueroa, G. *Estudio de una variante del problema bin-packing*. XVII International Symposium on Mathematical Methods Applied to the Sciences. San José-Costa Rica. Febrero. 2010.

[6] Carrera, L. y Figueroa, G. *Optimización combinatoria, optimización de Lipschitz y algoritmos evolutivos*. Informe final de proyecto. VIE. Cartago-Costa Rica. Enero. 2010.

[7] Hromkovič. Algoritmics for Hard Problems. 2nd edición, Springer Verlag, Berlin, 2004

[8] Grama, A. Gupta, A. Karypis, G. Kumar, V. Introduction to Parallel Computing. 2 edición, Addison Wesley, New York, 2003.

[9] Quinn, Michael. Parallel Programming in `C` with `MPI` and `OpenMP`. McGraw-Hill, Singapore, 2004.

[10] Gibbons, Alan y Rytter, Wojciech. Efficient Parallel Algorithms. Cambridge University Press, New York, 1990.

[11] Chapman, B. Jost, G. Van Der Pas, Ruud. Using `OpenMP`. MIT Press. Massachusetts. 2007.

[12] Talbi, El-Ghazali (editor). Parallel Combinatorial Optimization. John Wiley & Sons, New Jersey, 2006.

[13] Gonzalez, T. (editor). Handbook of Approximation Algorithms and Metaheuristics. Chapman & Hall, New York, 2007.