Paralelización a algoritmos de compresión fractal de imágenes

Silvia Piscia[1], Gabriela Guerrera[1], Ing. Mariana del Fresno[2]

ISISTAN[3] - LIDI[4]

Resumen

Se analiza la paralelización de un algoritmo clásico de compresión fractal de imagenes, utilizando procesadores heterogéneos conectados en red con un soporte de procesamiento distribuido basado en PVM y XPVM.

En particular se discuten dos métricas de interés: el speed-up obtenible (separando los tiempos propios del overhead de comunicaciones) al incrementar el número de procesadores y la pérdida resultante del índice de compresión alcanzado.

Si bien se trata de una clase de algoritmo muy particular, resulta de interés tecnológico sobre todo en aplicaciones donde la información es comprimida una vez y almacenada para su recuperación en consulta muchas veces (por ejemplo en servidores de información InterNet) ya que los índices de compresión alcanzables con una pérdida aceptable son muy altos y al mismo tiempo el algortimo de descompresión es muy rápido.

Por último se discute el efecto de variar el particionamiento de la imagen sobre los tiempos de procesamiento y la posibilidad de realizar un particionamiento adaptivo con el fin de optimizar la relación índice de compresión/pérdida aceptable sin deterioro del speed-up.

[1] Alumna de Trabajo Final de Ingeniería de Sistemas. Departamento Computación y Sistemas. Facultad de Ciencias Exactas. UNCPBA.

[2] Docente investigador del ISISTAN. Departamento Computación y Sistemas. Facultad de Ciencias Exactas. UNCPBA. E-mail: mdelfres@tandil.edu.ar

Trabajo dirigido por el Ing. Armando E. De Giusti. Inv. Principal del CONICET. Profesor TiT.Ded.Exc, Dpto. de Informática, Facultad de Ciencias Exactas, UNLP. E-mail: degiusti@ada.info.unlp.edu.ar

[3] Instituto de Sistemas de Tandil. San Martin 57, (7000) Tandil

[4] Laboratorio de Investigación y Desarrollo en Informática. Calle 50 y 115, 1º piso, (1900) La Plata.

Introducción

Una de las áreas de mayor desarrollo de la informática en los últimos años es el procesamiento paralelo. Hay varios factores que explican esta clara tendencia, entre ellos: las soluciones tecnológicas que resuelven a bajo costo el multiprocesamiento, las exigencias crecientes de potencia de cálculo, la eficiencia alcanzada en los servicios de comunicaciones y con ello la facilidad de procesamiento distribuido, así como las limitaciones naturales del procesamiento secuencial.

Básicamente, hay tres tendencias tecnológicas que responden al requerimiento de capacidad de procesamiento paralelo. Ellas son :

⇒ la investigación de nuevas arquitecturas de procesadores (RISC, DSP y DATA FLOW), así como combinaciones de las mismas en estructuras tales como hipercubos o anillos de procesadores,

⇒ la evolución de los sistemas operativos para soportar procesamiento distribuído en nodos locales y remotos, así como manejo de recursos compartidos, físicamente distribuídos (por ejemplo bases de datos),

⇒ el desarrollo de recursos de software (lenguajes, compiladores, herramientas de especificación y modelización) y la investigación en expresión y transformación de algoritmos secuenciales, de modo de explotar el paralelismo intrínseco en la aplicación, utilizando los recursos de procesamiento disponibles.

En el campo de las aplicaciones de procesamiento paralelo, se destacan aquellas relacionadas con el tratamiento de señales en tiempo real (radar, videoconferencia, reconocimiento de patrones digitales de voz e imagen, cálculo de trayectorias, GIS) o con procesamiento masivo de información fuera de línea (imágenes satelitales, imágenes médicas, registros de señales geológicas, cálculo numérico de modelos).

En la mayoría de los casos, se trata de problemas ya analizados en forma secuencial que, debido a su complejidad, motivan el análisis de una transformación del algoritmo asociado mediante metodologías de paralelización, utilizando la capacidad de multiprocesamiento disponible. Luego, es posible realizar un análisis de su comportamiento a través de técnicas de evaluación de performance de algoritmos que ejecutan cooperativamente sobre ambientes distribuídos.

En este trabajo se estudian los aspectos de la performance de algoritmos paralelos, aplicando dichos conceptos para evaluar el comportamiento de un algoritmo secuencial y de éste mismo una vez paralelizado.

Es importante comprender conceptos como perfil de paralelismo, factor de speedup y también el concepto de eficiencia del sistema. Para ello, se examinan las posibles combinaciones entre estas métricas de performance.

Dado que es difícil disponer en el ámbito académico de arquitecturas realmente paralelas, se ha trabajado sobre un ambiente distribuído en red, utilizando un soporte para procesamiento en paralelo distribuído (PVM), que permita ejecutar algoritmos en forma paralela en los procesadores de la red y evaluar la performance global de los mismos.

Se ha analizado (por sus características de cómputo intensivo) una implementación de un algoritmo de compresión fractal, basado en la propuesta de Y. Fisher [FIS94], el cual ha sido paralelizado. Se realizaron mediciones sobre el algoritmo secuencial y se registraron los resultados. También se sometió a prueba la versión paralelizada. Con los resultados disponibles, se ha llevado a cabo un estudio de los mismos, el que se desarrolla en función de las métricas estudiadas. Posteriormente, se presenta un análisis de posibles extensiones a este trabajo y las conclusiones a las que se ha arribado.

Sistemas Distribuidos y Procesamiento Paralelo

Los sistemas distribuidos consisten de computadoras individuales o procesadores que se comunican vía red. El *procesamiento paralelo* es el concepto básico que hay detrás de todas las configuraciones modernas de hardware de computador para sistemas grandes y complejos. El problema complejo se divide en un número de tareas pequeñas que son distribuídas entre varios procesadores. El uso de arquitecturas para procesamiento en paralelo dará como resultado un incremento realmente efectivo en la velocidad de cómputo, siempre que el mecanismo de comunicación entre los procesos sea eficiente.

Después que los elementos de software del procesamiento paralelo han sido determinados, el problema que resta es saber cómo distribuir estos elementos entre los procesadores. La distribución puede ser dejada completamente al sistema operativo para que controle el flujo computacional, o el programador/diseñador puede especificar la distribución usando primitivas o directivas.

Otro aspecto a considerar es que, mientras ciertas partes de un algoritmo pueden ser tratadas en paralelo, otras porciones deben ser tratadas secuencialmente; de esta forma, la arquitectura debe ser diseñada a fin de permitir ambas formas de procesamiento cuando sea requerida. Más allá de esto, la performance del sistema estará afectada por la cantidad de procesamiento paralelo versus procesamiento serial que se requiera.

Métricas para estimar performance

Cuando se requiere analizar el comportamiento de una aplicación, hay algunas cuestiones que son muy importantes, por ejemplo:

- Tiempo de ejecución de 1 aplicación sobre una arquitectura dada.
- Tiempo de ejecución de una serie de aplicaciones semejantes sobre la misma arquitectura.
- Evaluación de perfomance de los mismos algoritmos sobre arquitecturas alternativas.
- Dependencia de la perfomance respecto de los datos o el tamaño del problema.
- Incidencia de las diferentes componentes temporales en la ejecución (comunicaciones entre procesadores, tiempos muertos por sincronización, E/S, acceso a recursos compartidos , etc)
- Objetivo final de la optimización (tiempo absoluto, relación perfomance/costo de la arquitectura, grado de paralelismo efectivo alcanzado, minimización de la subutilización de los procesadores)
- Dependencia de la perfomance respecto del lenguaje o el compilador.

Todas estas cuestiones son bastante complicadas en el mundo del procesamiento secuencial, donde los conjuntos de instrucciones (RISC vs. CISC), las estrategias adoptadas, la memoria virtual y los compiladores de optimización introducen complicaciones que pueden ser engañosas. Para los procesadores paralelos, establecer puntos de referencia que permitan estimar performance en forma segura es aún más difícil. La performance a menudo depende críticamente de factores tales como elección del algoritmo, tamaño de la máquina y de los datos, y configuración de E/S.

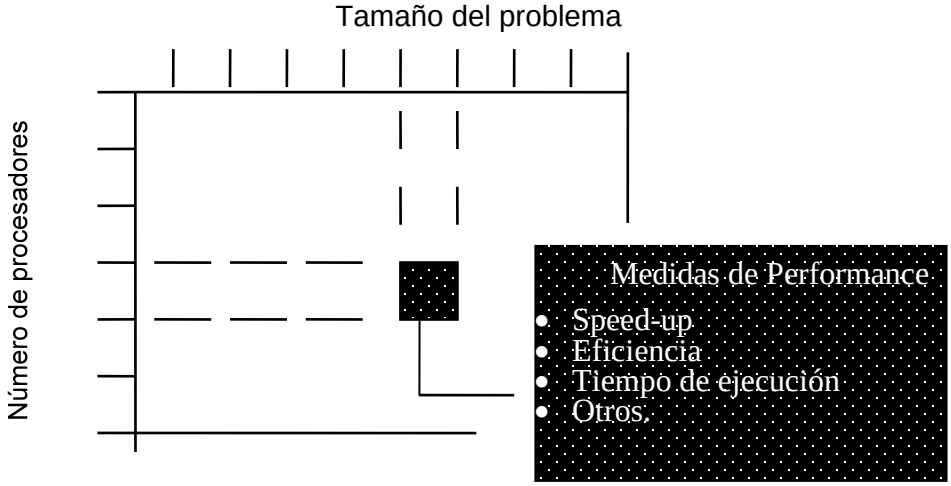
Una buena estrategia de los puntos de referencia debe tener en cuenta esto y debe estar estructurada de forma tal que se puedan hacer extrapolaciones razonables y soportables para ampliar aplicaciones punto a punto.

Ley de Amdahl

La idea clave subyacente de la Ley de Amdahl [Ref 8) es que el tiempo de ejecución total para un problema, cuando se ejecuta sobre un único procesador, puede ser dividido en dos partes. La primera es la parte *paralela* del problema. Esta puede ser acelerada aplicando más procesadores. La segunda es la parte *secuencial* del problema. Esta parte no puede ser acelerada agregando más procesadores. Brevemente, la Ley de Amdahl dice que la fracción del problema que es secuencial impone un límite inferior rígido sobre el tiempo de corrida obtenible (y de aquí, un límite superior rígido sobre el speed-up obtenible), *independiente* del número de procesadores en el arreglo paralelo.

El siguiente paso es estimar la fracción real del problema que es inherentemente secuencial. Cuanto más grande sea esta fracción menos eficientemente usada puede ser una máquina paralela grande. Sin embargo, si esta fracción es muy pequeña, puede ser que los arreglos de procesadores paralelos grandes se usen eficientemente. El tamaño de la parte secuencial del problema relativo al problema total *cambia con el tamaño del problema*. Generalmente la parte secuencial de un problema crece a una velocidad más lenta que la parte paralelizable. Así, a medida que el tamaño del problema crece, la *fracción* del tiempo de ejecución gastado en la parte secuencial puede volverse arbitrariamente pequeña, permitiendo que los arreglos paralelos arbitrariamente grandes sean aplicados eficientemente. En resumen, *las máquinas paralelas grandes necesitan problemas grandes*.

Es importante llevar la cuenta del tiempo sobre el rango de los tamaños de problemas sobre los cuales va a correr la aplicación y sobre el rango de los tamaños de máquina (es decir, número de procesadores) a considerarse.



Los puntos de referencia paralelos deben reflejar el tamaño de la máquina y el del problema.

Indicadores de performance: Factor de Speed-up y Eficiencia del sistema

Sea O(*n*) el número total de operaciones realizadas por un sistema de *n* procesadores y T(*n*) el tiempo de ejecución en unidades de paso del tiempo. En general, T(*n*) < O(*n*) si *n* procesadores realizan más de una operación en una unidad de tiempo, donde $n \geq 2$. Se supone además que T(1) = O(1) en un sistema uniprocesador. Se usa el speedup *S*(*n*) para indicar el grado de velocidad ganado en el cómputo en paralelo. El *factor de speedup* se define como:

$$S(n) = T(1)/\ T(n)$$

La *eficiencia del sistema* para un sistema de n procesadores está definida por:

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)}$$

La eficiencia es un indicador del grado de performance del speedup real conseguido comparado con el máximo valor. Como $1 \leq S(n) \leq n$, entonces $1/n \leq E(n) \leq 1$.

La eficiencia más baja corresponde al caso, en el que el código de programa completo se está ejecutando secuencialmente sobre un único procesador. La eficiencia máxima es conseguida cuando los *n* procesadores son completamente utilizados durante el período de ejecución.

Estas medidas iniciales pueden ser revisadas. Para una arquitectura, algoritmo y tamaño del problema *s* dados, el *speedup asintótico S(n,s)* es el mejor speedup que se puede obtener, variando solamente el número (*n*) de procesadores. Si se considera que el tiempo de ejecución secuencial sobre un uniprocesador es *T*(*s*,1), entonces *T*(*s*,*n*) será el tiempo de ejecución en paralelo mínimo sobre un sistema de *n*-procesadores y *h*(*s*,*n*) será la suma total de todas las comunicaciones y overheads de E/S, estos incluyen latencias de comunicación causadas por demoras de acceso a memoria, comunicaciones entre procesadores sobre un bus o sobre una red, o el overhead del sistema operativo y demoras causadas por interrupciones. Entonces, si se considera *h(s,n)* como la suma total de todos los overheads del sistema sobre un sistema de *n* procesadores, el speedup asintótico queda definido formalmente como sigue:

$$S(s,n) = \frac{T(s,1)}{T(s,n) + h(s,n)}$$

La demora del overhead *h(s,n)* es ciertamente dependiente de la aplicación así como también dependiente de la máquina. Es muy difícil obtener una fórmula para *h(s,n)* y generalmente puede considerarse despreciable.

El tamaño del problema es el parámetro independiente, sobre el cual se basan todas las otras métricas. El *T*(*s*,*n*) es mínimo en el sentido de que el problema se resuelve usando tantos procesadores como sean necesarios para conseguir el tiempo de corrida mínimo para un tamaño de problema dado.

La *eficiencia del sistema* al usar una máquina para resolver un problema dado se define por la siguiente proporción:

$$E(s,n) = \frac{S(s,n)}{n}$$

En general, la mejor eficiencia posible es uno, lo que implica que el mejor speedup es lineal o $S(s,n) = n$.

Otro de los indicadores de performance es la escalabilidad, cuya definición intuitiva podría ser: un sistema es escalable si la eficiencia del sistema es E(s,n)=1 para todos los algoritmos con cualquier número de procesadores n y para cualquier tamaño de problema s.

En general, si la carga de trabajo o el tamaño del problema se mantiene sin cambios, entonces la eficiencia decrece rápidamente a medida que crece el tamaño de máquina. La razón es que la sobrecarga se incrementa más rápido que el tamaño de máquina. Para mantener la eficiencia en el nivel deseado, se debe incrementar el tamaño de máquina y el tamaño del problema proporcionalmente.

El análisis de escalabilidad determina si el procesamiento paralelo de un problema dado puede ofrecer el mejoramiento deseado en performance.

La noción de escalabilidad está unida a las nociones de speedup y eficiencia. La definición de escalabilidad deber ser capaz de expresar los efectos de la red de interconexión de arquitecturas, de los patrones de comunicación inherentes a los algoritmos, de las restricciones físicas impuestas por la tecnología, y del costo efectividad o eficiencia del sistema.

Entre las métricas básicas que afectan a la escalabilidad de un sistema de computadora para una aplicación dada , se pueden analizar:

♦ *Tamaño de la máquina(n)* - el número de procesadores empleado en un sistema de computador paralelo.

♦ *Tamaño del problema(s)* - la cantidad de carga de trabajo computacional o número de datos para resolver un problema dado. El tamaño del problema es directamente proporcional al *tiempo de ejecución secuencial* T(s,1) para un sistema de uniprocesador debido a que cada punto de dato puede demandar una o más operaciones.

♦ *Tiempo de CPU(T)* - el tiempo de CPU real (en segundos) transcurridos en ejecutar un programa dado sobre una máquina paralela con *n* procesadores colectivamente. Este es el *tiempo de ejecución paralelo*, denotado como *T(s,n)* y es una función de *s* y de *n*.

♦ *Overhead de comunicación (h)* - la cantidad de tiempo gastado por la comunicación entre procesadores, sincronización, acceso remoto a memoria, etc. Este overhead incluye todas las operaciones sin cómputo, las cuales no involucran CPUs o dispositivos de E/S. Este overhead *h(s,n)* es una función de *s* y *n* y no es parte de *T(s,n).* Para un sistema uniproceador en monotarea el overhead es *h(s,n)* = 0.

Soporte para procesamiento paralelo distribuido: PVM

En este trabajo se ha utilizado PVM (Parallel Virtual Machine) como soporte de procesamiento paralelo distribuido. Un concepto clave en este sistema es que hace que un conjunto de computadoras aparezca como una única máquina virtual grande, de aquí su nombre. PVM usa el modelo de pasaje de mensajes para permitir a los programadores explotar el cómputo distribuido a través de una amplia variedad de tipos de computadoras.

Cómo trabaja PVM?

El software que utiliza PVM provee un ambiente de trabajo unificado dentro del cual se pueden desarrollar programas paralelos de una manera directa y eficiente, usando el hardware existente. PVM transparentemente maneja todo el ruteo de mensajes, conversión de datos y scheduling de tareas a través de una red de arquitectura de

computadoras heterogéneas. El modelo es simple y muy general, y se acomoda a una amplia variedad de estructuras de programas de aplicación. La interface de programación es deliberadamente directa, permitiendo así que las estructuras de programas simples sean implementadas de manera intuitiva. El usuario escribe su aplicación como un conjunto de *tareas* que cooperan. Estas tareas acceden a los recursos a través de una librería de rutinas de interface estándar, las cuales permiten la iniciación y terminación de tareas a través de la red así como también la comunicación y sincronización entre ellas. Las primitivas de pasaje de mensajes PVM están orientadas a vincular procesadores heterogéneos, involucrando constructores fuertemente tipados para buffering y transmisión. Los constructores de comunicación incluyen estructuras para envío y recepción de estructuras de datos así como también primitivas de alto nivel como transmisión, sincronización de barreras y suma global.

Las tareas PVM pueden tener control arbitrario y estructuras de dependencia. En otras palabras, en cualquier punto de la ejecución de una aplicación concurrente, cualquier tarea en existencia puede activar o detener otras tareas o agregar o borrar computadoras de la máquina virtual. Cualquier proceso se puede comunicar y/o sincronizar con cualquier otro. Además, es posible implementar algún control específico y estructuras de dependencia bajo el sistema PVM con el uso apropiado de constructores PVM y de sentencias de control de flujo del lenguaje anfitrión.

El paradigma general para programar aplicaciones con PVM es como sigue. Un usuario escribe uno o más programas secuenciales en un lenguaje convencional, que contienen llamados incluídos en la librería PVM. Cada programa corresponde a una tarea que integra la aplicación. Estos programas son compilados para cada arquitectura en la lista de anfitriones (pool host), y los archivos objeto resultantes son ubicados en un lugar accesible desde las máquinas en la lista de anfitriones. Para ejecutar una aplicación, un usuario típicamente comienza una copia de una tarea (usualmente el "maestro" o tarea de "iniciación") desde una máquina de la lista de anfitriones. Este proceso subsecuentemente inicia otras tareas PVM, resultando eventualmente en un conjunto de tareas activas que luego computan localmente e intercambian mensajes con cada una de las otras para resolver el problema.

Técnicas de programación básicas en PVM

Desarrollar aplicaciones para el sistema PVM - en un sentido general, por lo menos - sigue el paradigma tradicional para programación de multiprocesadores con memoria distribuída tales como nCUBE o la familia de Intel de multiprocesadores. Las técnicas básicas son similares tanto para los aspectos lógicos de la programación como para el desarrollo de los algoritmos. Sin embargo, existen diferencias significativas, en términos de : manejo de tareas (especialmente temas relacionados a creación de procesos dinámicos, nombrado y direccionamiento), fases de inicialización anteriores al cómputo real, alternativas de granularidad y heterogeneidad.

El cómputo paralelo usando un sistema tal como PVM puede ser aproximado desde tres puntos de vista fundamentales, basado en la organización de las tareas de cómputo. Dentro de cada una, hay diferentes estrategias de asignación de carga de trabajo. El primer modelo y más común para aplicaciones PVM puede ser denominado cómputo "crow" (multitud): una colección de procesos fuertemente relacionados, típicamente ejecutando el mismo código, que realizan cómputos sobre diferentes porciones de la carga de trabajo, involucrando usualmente intercambios periódicos de resultados intermedios. Este paradigma puede ser además subdividido en dos categorías:

⇒ El modelo maestro-esclavo (o nodo host) en el cual un programa de control separado, llamado maestro, es responsable por los procesos de generación, inicialización, reunión y muestra de los resultados, y tal vez el que lleva la cuenta del tiempo de las funciones. Los programas esclavos realizan el cómputo real involucrado; sus cargas de trabajo son asignadas por el maestro (estática o dinámicamente) o realizan ellos mismos las asignaciones.

⇒ El modelo de nodo-único donde múltiples instancias de un programa único ejecutan, con un único proceso (típicamente el único iniciado manualmente) haciéndose cargo de las responsabilidades no-computacionales además de contribuir al cómputo en sí mismo.

El segundo modelo soportado por PVM se llama cómputo "tree" (árbol). En este escenario, los procesos son generados (usualmente dinámicamente a medida que el cómputo progresa) en una manera semejante a un árbol, estableciendo de tal modo la relación padre-hijo (como opuesto al cómputo crow donde existe la relación semejante a una estrella). Este paradigma, si bien se usa menos comúnmente, es natural que encaje en aplicaciones donde la carga de trabajo total es conocida *a priori*, por ejemplo, en algoritmos branch-and-bound (podado), búsqueda alpha-beta (búsqueda local) y algoritmos recursivos de divide y conquista.

El tercer modelo llamado "híbrido", puede considerarse como una combinación del modelo tree y del modelo crow. Esencialmente, este paradigma posee una estructura de generación arbitraria: es decir, en cualquier punto durante la ejecución de una aplicación, la relación proceso - estructura puede parecerse a un diagrama arbitrario y cambiante.

Se debe notar que estas tres clasificaciones son hechas sobre la base de relaciones de procesos, si bien ellas frecuentemente además corresponden a topologías de comunicación. No obstante, en las tres, es posible para cualquier proceso interactuar y sincronizar con cualquier otro. Más allá de esto, como puede esperarse, la elección del modelo depende de la aplicación y debería ser seleccionado el que mejor se corresponda con la estructura natural del programa paralelizado.

Portando aplicaciones existentes a PVM

A fin de utilizar el sistema PVM, las aplicaciones deben atravesar dos etapas. La primera concierne al desarrollo de la versión paralela con memoria distribuída del o los algoritmos de la aplicación. Las decisiones de paralelización actual caen dentro de dos categorías -aquellas relacionadas a la estructura y aquellas vinculadas a la eficiencia.

Para las decisiones estructurales en la paralelización de aplicaciones, las principales decisiones se toman cuando se incluye el modelo que va a usarse. Las decisiones con respecto a la eficiencia, cuando se paraleliza para ambientes distribuídos, están generalmente orientadas hacia la minimización de la frecuencia y el volumen de las comunicaciones. Para los ambientes PVM basados en redes, la granularidad alta generalmente conduce hacia una mejor performance.

La paralelización de aplicaciones puede hacerse desde versiones secuenciales existentes, o desde versiones paralelas existentes. En el primero de los dos casos, los pasos involucrados son : seleccionar un algoritmo apropiado para cada una de las

subtareas de la aplicación, usualmente a partir de descripciones publicadas o diseñando un algoritmo paralelo, y entonces se codifica estos algoritmos en el lenguaje elegido y se realiza la interface de ellos con otros, así como también con el manejador de los procesos y otras construcciones.

Los programas paralelos existentes pueden estar basados en los paradigmas de memoria compartida o memoria distribuída. Convertir los programas de memoria compartida existentes a PVM es similar que convertir a los de código secuencial cuando las versiones de memoria compartida se basan en el paralelismo primario o nivel de loop. En el caso de programas de memoria compartida explícitos, la tarea primaria es ubicar los puntos de sincronización y reemplazar éstos con pasaje de mensajes. Para convertir código de memoria distribuída existente a PVM, la tarea principal es convertir desde un conjunto de construcciones concurrentes a otro. La principal diferencia entre PVM y otros sistemas son: (a) manejo de procesos y esquemas de direccionamiento de procesos, (b) configuración/reconfiguración de la máquina virtual y su impacto sobre las aplicaciones que se ejecutan, (c) heterogeneidad en los mensajes así como también el aspecto de heterogeneidad que trata con las diferentes arquitecturas y representaciones de datos, (d) ciertas características únicas y especializadas tales como señalización, y métodos de organización de tareas.

Compresión fractal de imágenes

La necesidad de utilizar algoritmos de compresión como etapa previa al almacenamiento o transmisión de imágenes es indudable, debido a la gran cantidad de espacio que éstas requieren. La compresión permite reducir el tamaño de los datos que describen una imagen, ya que logra disminuir la redundancia presente en ellos. Aún en casos de compresión con pérdida, es posible reconstruir una aproximación aceptable de la imagen original de acuerdo a las características del sistema de percepción visual del ser humano.

Existen diversos métodos de compresión de imágenes. Uno de los más populares es el standard JPEG (Joint Photographic Experts Group) para imágenes fijas, pero la investigación sobre otros métodos alternativos no se ha detenido y han surgido varias técnicas de codificación de imágenes, entre ellas la compresión fractal.

Una característica interesante de los fractales es su capacidad para representar esquemas complejos, como puede serlo una imagen determinada, por un modelo o "ecuación" de la misma bastante sencilla. Si se observan las nubes, los árboles, las rocas, los campos, se verá que están compuestos por estructuras muy semejantes repetidas, quizás, a diferentes escalas.

La palabra ´fractal´ fue creada por B. Mandelbrot para designar a aquellos objetos o estructuras fracturadas, con alto grado de redundancia, construídos recursivamente por copias transformadas y reducidas de ellos mismos, teniendo detalle a cualquier escala. Para citar un ejemplo, en la figura 1 se muestra un helecho : una figura común a simple vista, pero que, cuando se la analiza puede verse que está formada por transformaciones reducidas de la imagen completa.


Figura 1

La idea básica para el desarrollo de la técnica de compresión fractal ha sido la de aplicar las propiedades de autosimilitud de las imágenes fractales para la compresión de las mismas. De esta manera, M. Barnsley, creador de esta técnica de compresión, mostró cómo imágenes de alta complejidad visual se pueden describir según lo expresado anteriormente, y planteó entonces que toda la información requerida para su reconstrucción puede conocerse almacenando, en unos pocos bytes, los coeficientes de dichas transformaciones [BAR88]. Generalmente, las transformaciones consideradas son *funciones afines contractivas,* compuestas por combinaciones de rotaciones, traslaciones, reflexiones y cambios de escala en un espacio n-dimensional. Tal conjunto de transformaciones conforma la codificación de una imagen, la que M. Barnsley denominó *Iterated Function System* (IFS), el cual puede expresarse como:

$$W(S) = \bigcup_{i=1}^{n} w_i(S)$$

donde $w_i(S)$ representa cada una de las transformaciones de la imagen.

El objetivo de esta técnica de compresión es, entonces, encontrar esquemas repetitivos en una imagen dada y descubrir las transformaciones involucradas. Esta tarea no es nada sencilla ni económica, y han surgido varios planteos acerca de cómo poder realizarla en tiempo razonable y con resultados aceptables.

Una cuestión interesante es que la mayoría de las imágenes no se comportan como fractales "puros", y esto ha ocasionado críticas al enfoque planteado por Barnsley en sus trabajos originales, por parte de aquellos que opinan que la técnica no sería aplicable a las imágenes corrientes. Obviamente, una imagen convencional no contiene el tipo de similitud que aparece en una imagen fractal pura, o en las ramas de los árboles, o en los helechos y, seguramente no será posible encontrar transformaciones afines que describan la imagen completa de la manera antes descripta. Sin embargo, se puede extender el concepto de IFS a porciones de la imagen, definiendo un IFS Particionado (PIFS). Al utilizar esta variante de la propuesta original, se produce una clase diferente de autosimilitud, ya que la imagen será aproximada por la unión de porciones transformadas de sí misma. Seguramente, dado que no se trata de un fractal puro, la unión de tales regiones no formarán una copia exacta de la imagen dada, por lo que se deberá permitir algún tipo de error en su reconstrucción.

Hay principios matemáticos que sustentan la propuesta de codificación fractal, los que ya han sido analizados anteriormente [DEL97]. Uno de ellos es el teorema del punto fijo de las transformaciones contractivas (el cual asegura que será posible encontrar una única imagen como resultado de la iteración del IFS), y el teorema del Collage (que establece que cuanto más se aproxime la unión de las imágenes transformadas -collage- a la imagen original, más apropiadamente el conjunto de transformaciones proveerá una codificación de la imagen considerada).

Una implementación de compresión fractal

Una posible esquema de compresión es realizar un tratamiento sobre regiones o bloques de la imagen, encontrando semejanzas con otros dentro de la misma. Tal es el caso del método propuesto por Y. Fisher [FIS94], quien ha planteado un esquema de codificación basado en un particionamiento adaptivo de la imagen, basado en una estructura quadtree.

En este esquema de codificación se realiza una división de la imagen original en regiones no-superpuestas, denominadas rangos (R_i), las cuales deberán ser codificadas.

Por otra parte, se realiza otra división de la imagen en regiones dominio D_i, que sí pueden superponerse, ya que conforman la librería de bloques dominio, utilizada para buscar las semejanzas con los rangos. Entonces, para cada R_i, se debe realizar una búsqueda entre los D_i, para hallar uno tal que al aplicársele una transformación adecuada, difiera lo menos posible con el R_i en consideración. Como resultado de esta búsqueda, cada rango tendrá asociado un dominio tal que, al transformarse, lo codifica de una manera conveniente y se deberán almacenar los parámetros de cada transformación encontrada para la codificación de la imagen.

Según la estrategia de división quadtree, un bloque cuadrado de la imagen se divide en cuatro sub-bloques de igual tamaño, cuando no se encuentra una transformación apropiada de algún dominio. Este proceso divide la imagen recursivamente hasta que los bloques sean cubiertos con adecuado nivel de tolerancia. De acuerdo a esta partición, los bloques uniformes o sin demasiado detalle, serán más grandes y podrán cubrirse adecuadamente con algún bloque dominio de la librería. Aquellas regiones de mayor detalle, deberán subdividirse para codificarse en base a dominios más pequeños.

El conjunto de todas las transformaciones encontradas proporcionará la descripción de la imagen original que permita reconstruirla posteriormente.

La decodificación es una tarea más sencilla y rápida. Básicamente, consiste en efectuar iteraciones del conjunto de transformaciones descriptas, a partir de una imagen inicial cualquiera. Este proceso irá produciendo imágenes que se aproximarán cada vez más a la original. Cada iteración debe realizar el siguiente proceso: para cada uno de los rangos R_i de la imagen, al dominio D_i elegido para codificarlo se le debe aplicar la transformación encontrada, de acuerdo a la información que se indica en el archivo compactado.

El proceso de reconstrucción de la imagen original debe continuar hasta que se alcance el punto fijo de la transformación, o sea hasta que la nueva iteración no produzca un cambio significativo en la imagen obtenida (experimentalmente, se ha comprobado que entre cinco y diez iteraciones son suficientes para producir buenas aproximaciones).

Análisis de la paralelización del algoritmo de compresión fractal de imágenes

Si bien el esquema de compresión de imágenes basado en fractales ofrece altas tasas de compresión y velocidad de decodificación, su principal inconveniente es el de poseer costos más elevados de compresión.

Una de las posibilidades de extensión que ofrece el algoritmo planteado, y que ayudaría a reducir el tiempo de codificación, es la de paralelización. El enfoque más directo es asignar una porción diferente de la imagen a codificar a cada procesador.

Para cada uno de los procesos que codifican y decodifican una imagen, se ha generado un algoritmo "padre", encargado de dividir la imagen y asignar una porción de la misma a cada uno de los procesos que se encargarán de realizar la codificación/decodificación.

Se ha considerado siempre la generación de 4 procesos, ya que el algoritmo de compresión fractal de imágenes propuesto, utiliza el método de partición quadtree, el cual subdivide recursivamente la imagen en cuatro porciones.

Codificación

El algoritmo "padre" comienza extrayendo la cabecera del archivo, ya que ésta contiene información necesaria sólo para el manejo de la imagen; con lo que resta del archivo se genera uno nuevo (.dat) con los datos de la imagen propiamente dicha. Este es el punto justo donde se usan las facilidades para procesamiento paralelo distribuído que provee el soporte PVM; el primer paso es generar los procesos "hijos", mediante la llamada a la función pvm_spawn( ), quienes serán los encargados de codificar la imagen. El proceso padre empaqueta los datos a ser enviados a los procesos hijos, y los coloca en un buffer de envío que PVM provee para tal efecto. Los datos enviados son: todo el camino (path) para hallar el del archivo a compactar y el arreglo con los números de

proceso (tids) de los hijos generados. Una vez realizado este envío el padre se queda a la espera de respuesta por parte de los procesos hijos, éstos desarrollan sus tareas y cuando terminan se lo comunican al proceso padre, quien controla que todos los procesos hijos hayan terminado. A medida que cada proceso hijo le avisa de su finalización, el padre "mata" a este proceso mediante la llamada a la función pvm_kill( ), quedando finalizada la etapa de codificación cuando todos los procesos hijos hayan concluído las codificaciones parciales.

Para comenzar la decodificación, el proceso "padre" genera los procesos "hijosdec", que decodificarán la imagen. La información que se les envía consiste en: la ubicación de los datos que ellos deben usar como entrada y el arreglo que contiene sus tids; esta información también es empaquetada y colocada en un buffer de envío. Tras la generación de estos procesos, el padre se queda a la espera de respuesta; repitiéndose el proceso anteriormente mencionado, una vez que que éstos finalizan. El resultado del procedimiento de decodificación en paralelo genera 4 archivos (.out), cada uno de los cuales es utilizado por el proceso padre para formar la imagen reconstruída a partir de las transformaciones generadas por el proceso de compresión.

Cuando el proceso padre llama por primera vez a la función pvm_spawn( ), se generan 4 procesos "hijos", los cuales trabajan de la siguiente manera: cada proceso se pregunta cuál es su número de proceso a través de la función pvm_mytid( ) y una vez obtenido este número, se busca la posición de éste dentro del arreglo, la cual indica qué porción de la imagen le corresponde procesar. Mediante una función matemática cada proceso hijo extrae del archivo de la imagen propiamente dicha, generado por el padre (<*nombre*>.dat), la porción antes mencionada para formar su propia subimagen (parte$_i$.dat) y comenzar a codificar. La codificación se realiza según el esquema que se muestra en la figura 2:

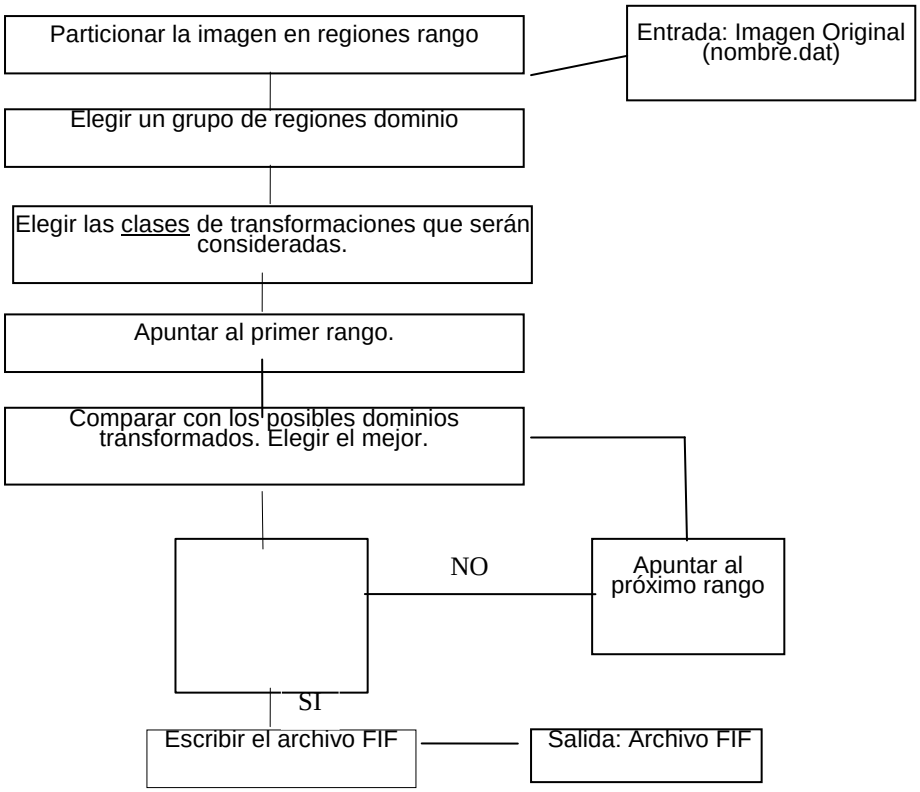


Figura 2

Etapa1. Se hace la partición de la imagen original en regiones no superpuestas, denominadas *rangos* R_i.

Etapa2. Se hace otra división de la imagen en regiones o bloques *dominio* D_i (librería de dominios), utilizada para buscar las semejanzas con los rangos.

Etapa3. Se realiza una clasificación de bloques dominios en tres clases canónicas, de acuerdo a la orientación de los niveles de brillo promedio de sus cuadrantes, y luego una sub-clasificación de éstas según los distintos órdenes de las varianzas en esos cuadrantes. Mediante la división en clases de dominios, un rango clasificado se compara sólo con aquellos dominios que pertenecen a su misma clase dentro del conjunto de aquellos potenciales.

Etapa4. Para cada R_i, se debe realizar una búsqueda entre los D_i, para hallar uno tal que al aplicársele una transformación apropiada, difiera lo menos posible con el R_i, en consideración. El conjunto de todas las transformaciones encontradas proporciona la descripción de la imagen necesaria para reconstruirla posteriormente.

Etapa5. Como resultado de la búsqueda entre los bloques dominios, cada rango tendrá asociado uno de ellos tal que, al transformarse lo codifica de una manera conveniente (según cierto criterio de calidad) y se deberán almacenar los parámetros de cada transformación encontrada para conformar la codificación de la imagen. Esta información se vuelca a un archivo con formato FIF(Fractal Image Format) que consistirá de un encabezado con información acerca de la elección de las regiones rangos, seguida de una lista de pares (coeficientes afines,bloques dominio), que generarán cada bloque rango de la imagen original.

Así concluye la codificación de la imagen original. Luego cada uno de los hijos le envía al proceso padre un mensaje que contiene su id de proceso y el tiempo que tardó en codificar. Para poder conocer el tiempo insumido por cada proceso hijo, se utiliza la función time( ), al inicio y finalización de cada uno de ellos.

Descompresión

Cuando el proceso padre llama a la función pvm_spawn( ) para realizar la descompresión, se generan 4 procesos "hijosdec", los cuales trabajan de la siguiente manera: cada proceso se pregunta cuál es su número de proceso a través de la función pvm_mytid( ) y una vez obtenido este número, se busca la posición de éste dentro del arreglo, la cual indica que porción de la imagen (parte$_i$.trn) le corresponde decodificar. La decodificación se realiza según los pasos mostrados en la figura 3:

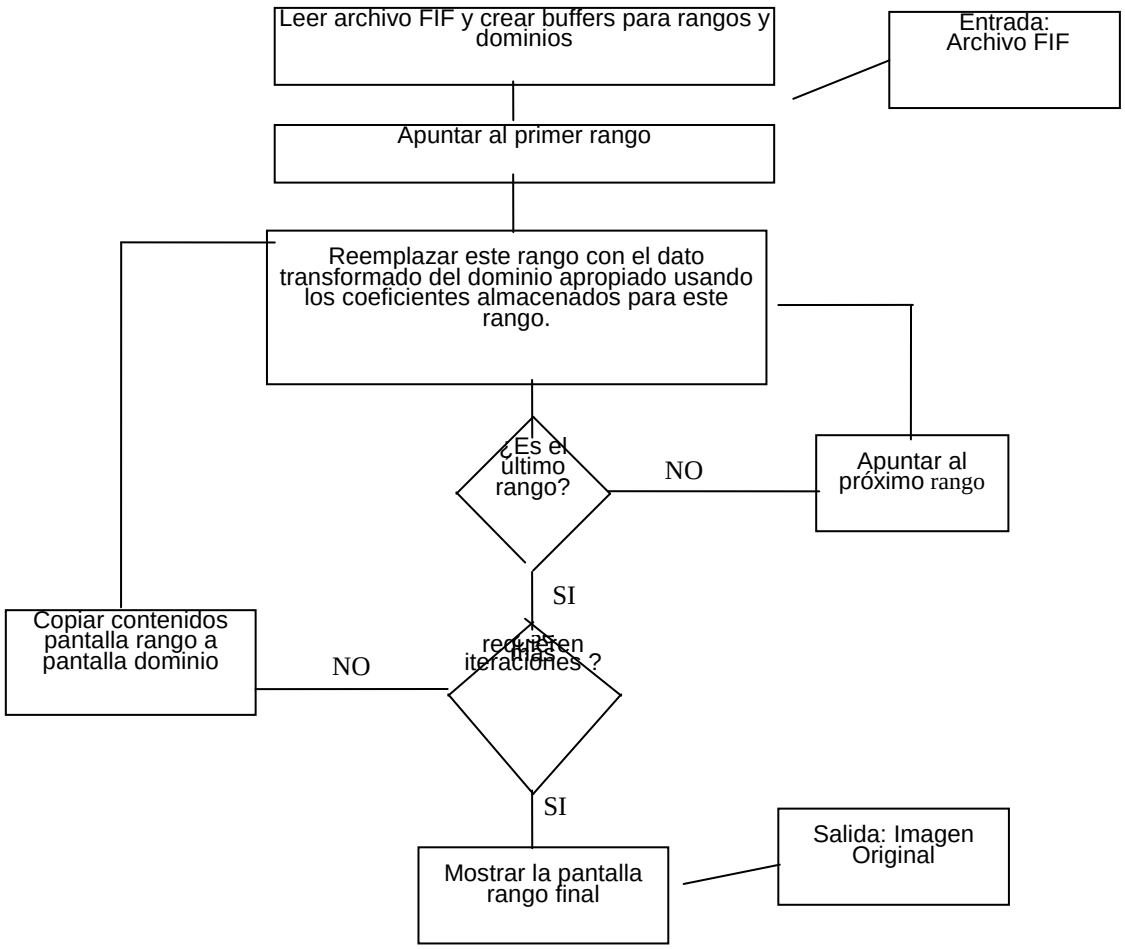


Figura 3

Como se explicó, el proceso consiste en efectuar algunas iteraciones de la transformación W descripta, sobre una imagen inicial cualquiera. Cada iteración debe realizar el siguiente proceso: para cada uno de los rangos R_i de la imagen, al dominio D_i elegido para codificarlo se le aplicar la transformación encontrada, de acuerdo a la información que se indica en el archivo compactado.

Análisis de resultados

En este trabajo, se han tenido en cuenta diferentes imágenes de 256 x 256 pixels, correspondientes al formato TIFF (Tagged Information File Format). Para cada una de ellas, se han tomado los tiempos (en segundos) de ejecución de los procesos codificador y decodificador, tanto para el algoritmo secuencial como para el paralelizado con 4 procesadores.

A continuación se muestran los resultados obtenidos para algunas de las imágenes, en tablas correspondientes a la codificación y decodificación, para tres corridas diferentes:

Codificación:

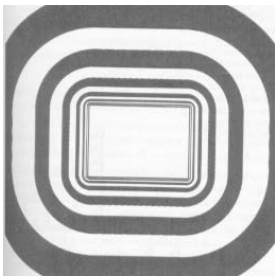
IMAGEN	ALGORITMO SECUENCIAL	ALGORITMO PARALELIZADO (4 PROCESADORES *)
BARN256.TIF	10 9 9	7 8 8
LENNA.TIF	14 12 12	10 10 9
LAND.TIF	13 15 12	10 9 9

(*) 3 PENTIUM 100 Mhz. 16 Mb - 1 PC 486 100 Mhz. 16 Mb.

Decodificación:

IMAGEN	ALGORITMO SECUENCIAL	ALGORITMO PARALELIZADO (4 PROCESADORES *)
BARN256.TRN	6 5 6	2 1 2
LENNA.TRN	5 6 7	2 1 2
LAND.TRN	6 6 6	2 3 2

(*) 3 PENTIUM 100 Mhz. 16 Mb - 1 PC 486 100 Mhz. 16 Mb.


BARN256.TIF


LENNA.TIF


LAND.TIF

En base a estos datos, se hicieron estimaciones de performance (en base al promedio de las diferentes ejecuciones), utilizando el factor de speed-up y calculando la eficiencia, para este algoritmo en particular:

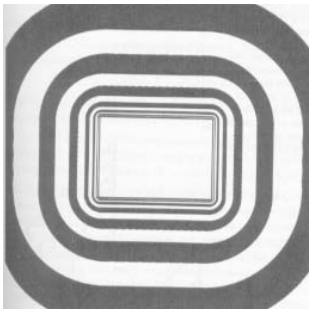
Codificación:

IMAGEN	SPEED-UP	EFICIENCIA
BARN256.TIF	1.26	0.31
LENNA.TIF	1.31	0.33
LAND.TIF	1.43	0.36

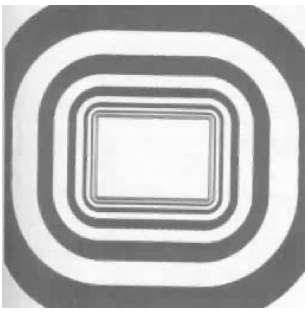
Decodificación:

IMAGEN	SPEED-UP	EFICIENCIA
BARN256.TRN	3.41	0.85
LENNA.TRN	3.61	0.90
LAND.TRN	2.57	0.64

Se han realizado pruebas con distintos parámetros que controlan elm proceso de la codificación. Las tasas de compresión obtenidas llegan a 20:1 (5% del tamaño original), con una degradación poco apreciable de la calidad de la imagen. En este caso, se ha trabajado con un criterio de aceptación de bloques semejantes bastante riguroso, por lo que los índices de compresión obtenidos no han sido demasiado altos, si bien la calidad de las imágenes una vez reconstruídas es altamente aceptable, como puede apreciarse en las siguientes imágenes.



Barn256.tif (original)



Barn468.tif (reconstruída)

Conclusiones y trabajo futuro

En este trabajo se ha analizado la paralelización de un algoritmo de compresión fractal de imagenes. Se han descripto las características del mismo y se han estudiado los resultados de diferentes ejecuciones, analizando el speed-up y la eficiencia obtenida.

Como se puede apreciar, con la paralelización del algoritmo de compresión fractal de imágenes descripto se ha logrado una reducción en el tiempo de procesamiento, en la codificación y la decodificación. En la primera, se ha mejorado en más del 30%, mientras que en la decodificación el aumento es más notorio, alcanzando hasta un 90%. Queda analizar, en una próxima etapa, el comportamiento del algoritmo paralelizado con mayor cantidad de procesadores, como también considerar imágenes de diferente tamaño.

Bibliografía

[BAR88] BARNSLEY MICHEL, "A better way to compress images", Byte, enero 1988.

[DEL97] DEL FRESNO MARIANA., DE GIUSTI ARMANDO. "Estudio sobre la implementación de la compresión fractal de imágenes". Proceedings III ICIE, 1997, Buenos Aires.

[DON94] DONGARRA, JACK y otros."PVM: Paralell Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing". Cambridge, Massachusetts y London, England, MIT Press, 1994. 279 págs.

[FIS94] FISHER, YUVAL y otros. "Fractal Image Compression", Theory and Application. San Diego, California, Yuval Fisher Editor, 1994. 341 págs.

[GON92] GONZALEZ, RAFAEL C. y WOODS, RICHARD E. "Digital Image Processing" reimpresa con correcciones junio 1992, Reading, Massachusetts, Addison-Wesley, 1992. 711 págs.

[HER91] HERMANN, D. W. y BURKITT A. N. "Parallel Algorithms in Computacional Science". Springer-Verlag, 1991.

[HWA93] HWANG, KAI. "Advanced Computer Architecture: Paralelism, Scalability, Programability", McGraw-Hill, 1993.

[LAW92] LAWSON, HAROLD W. y otros. "Parallel Processing in Industrial Real-Time Applications". Englewood Cliffs, New Jersey, Prentice-Hall, 1992, 512 págs.

[MOR] MORSE, STEPHEN H. "Practical Parallel Computing".

[RIM92] RIMMER, STEVE. "Supercharged Bitmapped Graphics". 1ed. 2da impresión, s.l., Windcrest/McGraw-Hill, 1992. 645 págs.

[MIC87] Tag Image File Format Specification. Revisión 5.0, Seattle, Aldus/Microsoft,1987, 1988.