Algoritmos paralelos para la solución de problemas de optimización discretos aplicados a la decodificación de señales

TESIS DOCTORAL

Marzo 2009

Presenta: Rafael Arturo Trujillo Rasúa

Dirigen: Dr. Antonio M. Vidal Maciá
Dr. Víctor M. García Mollá

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia

Índice general

1 Introducción y Objetivos

En este primer capítulo se muestra la motivación del trabajo desarrollado, dando una visión general de la tesis doctoral, planteando los objetivos que se pretenden cubrir, así como la estructuración del resto de capítulos.

1.1 Motivación

Un problema de optimización consiste en encontrar una o varias soluciones, de un conjunto de soluciones admisibles, que minimiza o maximiza el rendimiento de un determinado proceso. En cualquier tarea que involucre una toma de decisiones, en el propósito de tomar la "mejor" decisión entre varias disponibles, siempre puede surgir un problema de optimización. Ejemplos hay varios: encontrar una distribución de recursos que permita ejecutar cierta tarea en el menor tiempo posible, encontrar el camino más corto que pase por un conjunto dado de locaciones, encontrar las menores dimensiones de una viga que da soporte a cierta edificación, etc. Por lo general la medida de cuán buena es la opción o decisión que se desea tomar se describe mediante una función objetivo, de esta forma los métodos computacionales de optimización se encargan de seleccionar la mejor alternativa en dependencia de la definición de dicha función objetivo.

En los años recientes se ha sido testigo del vertiginoso progreso de la informática desde el punto

de vista tecnológico, especialmente en el incremento de la capacidad de cómputo de las computadoras personales al ser incorporados en estas procesadores paralelos. Aparejado y motivado por este hecho ha crecido notablemente la atención e interés por el campo de la optimización. Un claro ejemplo de este fenómeno es la amplia accesibilidad que existe hacia herramientas software de optimización, como el paquete de Optimización de MATLAB® [1] y otros muchos paquetes comerciales.

En [72] se define formalmente el problema de optimización sin restricciones como la búsqueda de un punto x^* tal que $f(x^*) \leqslant f(x)$ para todo x donde $f : \mathbb{R}^n \to \mathbb{R}$. Para estos problemas de optimización, donde el espacio de la solución es un espacio continuo, existen numerosos métodos que se basan en la información ofrecida por las derivadas de la función objetivo para construir las direcciones de búsqueda y llegar a la solución. En el caso que las segundas derivadas de la función pueden ser calculadas o estimadas de manera muy precisa, pueden usarse los métodos tipo Newton o quasi-Newton [52, 92, 96]. Hoy en día hay una amplia disponibilidad de implementaciones sofisticadas de métodos basados en derivadas, con estrategias de globalización por regiones de confianza [29] o por búsqueda lineal [95], y con opciones para generar aproximaciones del gradiente (el vector de las primeras derivadas parciales) y de la matriz Hessiana (matriz de las segundas derivadas parciales). Además, existen herramientas para efectuar derivaciones de manera automática [17, 18, 19, 20, 57], así como lenguajes de modelado que calculan las derivadas automáticamente [21, 47].

Sin embargo, uno de los tipos de problemas de optimización que surgen en la práctica con mucha frecuencia son aquellos que involucran variables que deben tomar valores discretos (por ejemplo, valores enteros). Es decir, el dominio de solución es un espacio multi-dimensional discreto. Estos problemas surgen cuando las variables representan unidades indivisibles, como por ejemplo personas, máquinas, etc., y que por tanto no se admiten soluciones fraccionadas. Debido a su naturaleza combinatoria, los problemas de optimización discretos presentan por lo general una complejidad computacional exponencial, y por tanto son mucho más complicados de resolver que los problemas continuos. Además, ya para estos problemas no son adecuados los métodos basados en derivadas, que por lo general convergen muy rápidamente, y por ende no procede ninguna de las herramientas mencionadas anteriormente.

De cara a la solución de estos problemas, no es viable hacer una búsqueda exhaustiva por el espacio discreto, pues se llegaría al caso peor para una complejidad exponencial [49]. En su lugar,

generalmente la solución se obtiene modelando los elementos del espacio como caminos dentro de un grafo o un árbol, la función objetivo se define en términos de los costos de los arcos y el problema entonces es reformulado a encontrar un camino entre un nodo inicial del grafo (que representa un punto inicial del espacio) y un nodo final (que representaría una solución al problema). Los esquemas de solución por Ramificación y Poda (*Branch-and-Bound*) [79], por programación dinámica [14] y por búsqueda heurística [69, 98] usan este tipo de modelación.

Existe otra clase de métodos que no usan derivadas y que perfectamente pueden usarse en la solución de problemas de optimización en espacios discretos: Los métodos basados en Búsqueda Directa (*Direct Search*) [73]. Los métodos de Búsqueda Directa se caracterizan por el hecho de que el proceso de la toma de decisiones se basa sólo en los valores de la función objetivo, por lo que no requieren el uso de las derivadas, ya sean explícitas o aproximadas, para determinar las direcciones de descenso. Este tipo de métodos fueron propuestos formalmente y aplicados ampliamente en los años 60 [64, 110, 94], pero en la década de los 70 fueron relegados a un segundo plano debido fundamentalmente a la falta de coherencia en su desarrollo y análisis matemático. Sin embargo continúan siendo fáciles de programar y algunos de ellos bastante fiables. En los últimos años los métodos de Búsqueda Directa han vuelto a tomar cierto protagonismo debido por una parte a la aparición de una formalización de sus aspectos matemáticos [23, 101] y por otra al advenimiento de la computación paralela y distribuida [34, 120]. La Búsqueda Directa puede ser usada en cualquier tipo de problema de optimización porque para su funcionamiento sólo necesita como información el valor de la función objetivo en cada punto del espacio de búsqueda. La gran mayoría de los trabajos publicados los ubican en el campo de la optimización continua, aunque bien pueden ser usados en la optimización discreta o en la optimización de funciones que no son de naturaleza numérica. De hecho, en [16, 101, 122] se proponen y formalizan técnicas de globalización usando retículas cuadradas como espacio por el cual el punto de iteración debe moverse durante la búsqueda.

La popularidad de los métodos de Búsqueda Directa persiste porque en la práctica han obtenido resultados favorables. Se han propuesto variantes basadas en heurísticas que para determinados problemas se logra una convergencia global similar a la lograda por métodos quasi-Newton con técnicas de globalización. Por otro lado, simultáneamente al surgimiento de la Búsqueda Directa se manejaron las

primeras ideas para lograr una hibridación de esta con los métodos quasi-Newton [110], comenzando el proceso de optimización por la Búsqueda Directa y luego "refinando" el resultado obtenido mediante un método basado en derivadas que siempre tendrán una mayor velocidad de convergencia local.

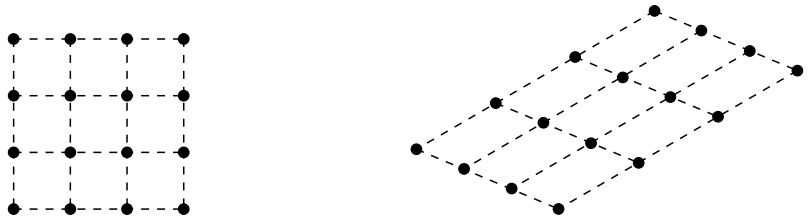
En esta tesis hemos enfocado nuestro interés en la resolución de problemas de optimización discretos mediante métodos por Búsqueda Directa y otros métodos surgidos a partir del estudio del problema en cuestión y que emplean técnicas *Branch-and-Bound* para su funcionamiento. El problema discreto en el cual se centra la investigación es el problema de mínimos cuadrados en espacios discretos:

$$\min_{s \in \mathcal{A}^m} \|x - Hs\|^2 \tag{1.1}$$

donde $x \in \mathbb{R}^{n \times 1}$, $H \in \mathbb{R}^{n \times m}$ y $\mathcal{A}$ es un conjunto de valores enteros, por lo que $\mathcal{A}^m$ es un subconjunto del espacio entero m-dimensional $\mathbb{Z}^m$.

Usualmente este problema se describe en términos de retículas (*Lattices*). Si los elementos de $\mathcal{A}$ están igualmente espaciados, el conjunto $\mathcal{A}^m$ forma una retícula rectangular como la mostrada en la figura 1.1(a). Cuando los elementos de $\mathcal{A}^m$ son multiplicados por la matriz H entonces la retícula sufre una especie de deformación, y puede resultar similar a como se muestra en la figura 1.1(b). El problema (1.1) es equivalente al problema de encontrar en una retícula generada por una matriz H el punto más cercano a un punto dado x, conocido por las siglas CVP (*Closest Vector Problem*), y del cual se conoce es un problema NP-*completo* [90]. Una búsqueda exhaustiva por toda la retícula para encontrar la solución de máxima verosimilitud (ML, *Maximum-Likelihood*) a (1.1) requeriría un tiempo exponencial tanto en el caso peor [58] como en el caso promedio [6] lo cual es prácticamente inviable.

Este tipo de problemas se ha utilizado ampliamente para expresar o solucionar numerosos problemas del mundo real. Por ejemplo en los sistemas de acceso múltiple por división de código (CDMA - *Code Division Multiple Access*) [22], en los sistemas de posicionamiento global (GPS, *Global Positioning Systems*) [60] y en los sistemas de criptografía [5, 53, 42]. En esta tesis nos centraremos de manera muy particular en el problema de la decodificación de señales en los sistemas de comunicaciones inalámbricos MIMO (*Multiple Input - Multiple Output*).



(a) (b)

Figura 1.1: (a)Retícula rectangular; (b)Retícula deformada.

La comunicación mediante arquitecturas con antenas MIMO [15] es uno de los desarrollos más significativos de la última década en el campo de las comunicaciones *wireless*. Éstas han emergido como uno de los sectores más pujantes de la industria de las telecomunicaciones, debido a la alta demanda de una conectividad cada vez más sofisticada que permita la comunicación en cualquier momento y en cualquier lugar.

Los sistemas MIMO están formados por múltiples antenas transmisoras y receptoras. En comparación con los tradicionales sistemas de sólo una antena transmisora y una antena receptora, los sistemas MIMO pueden ofrecer aumentos de capacidad en determinadas circunstancias de propagación [45, 119], los cuales pueden proporcionar mayor fiabilidad y calidad de la transmisión. Es significativo además que los beneficios ofrecidos por la tecnología MIMO pueden ser conseguidos sin la necesidad de recursos espectrales añadidos, los cuales no son sólo caros sino también escasos.

En los sistemas MIMO, cada equipo transmisor tiene asociadas M antenas transmisoras y cada receptor N antenas receptoras. En la figura 1.2 se muestra un sistema MIMO 2×2, es decir, con dos antenas de transmisión y dos antenas de recepción. La propagación de la señal no se realiza a través de un único canal, pues existe un canal entre cada antena transmisora y cada antena receptora, lo que obliga a representar la propagación a través de una matriz, que se conoce como matriz de canales, denotada usualmente por $\mathbf{H}$. El elemento h_{ij} representa la función de transferencia compleja entre la antena transmisora j y la antena receptora i.

Se han propuesto varias tecnologías MIMO dirigidas a diferentes escenarios de las telecomunicaciones. El sistema *Bell-Labs Layered Space Time* (BLAST) [44, 135] es una arquitectura diseñada para alcanzar altas tasas de transmisión de datos. En este esquema se transmiten de manera simultánea

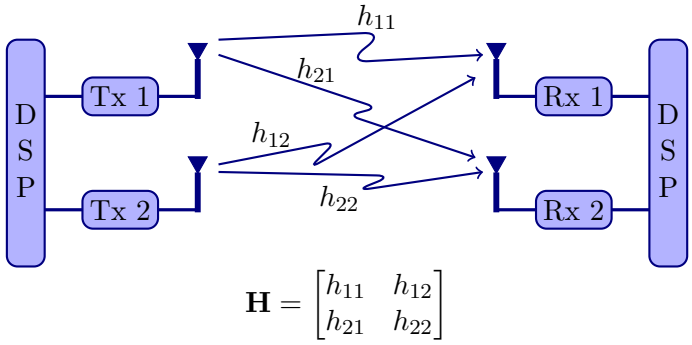


Figura 1.2: Sistema MIMO 2×2

diferentes flujos de datos desde todas las antenas transmisoras (éstas se solapan en frecuencia y tiempo), y las antenas receptoras reciben la superposición de todos los flujos transmitidos y les aplican algún método de procesado de señal para decodificarla. Otra tecnología son los sistemas de Codificación Espacio-Temporal (STC) [8, 116, 117, 118], en los cuales el mismo flujo de datos se transmite por cada una de las antenas transmisoras, así se reciben réplicas incorreladas y se pueden aplicar diversidad en recepción con ellas. Con los sistemas STC se persigue mayor fiabilidad en la transmisión de datos, mientras que con los sistemas BLAST se busca mayor velocidad de transmisión. Ambos esquemas pueden alcanzar altas tasas de transferencia empleando constelaciones de gran tamaño.

En la figura 1.3 se muestra el modelo general del proceso de transmisión-recepción de un sistema MIMO.

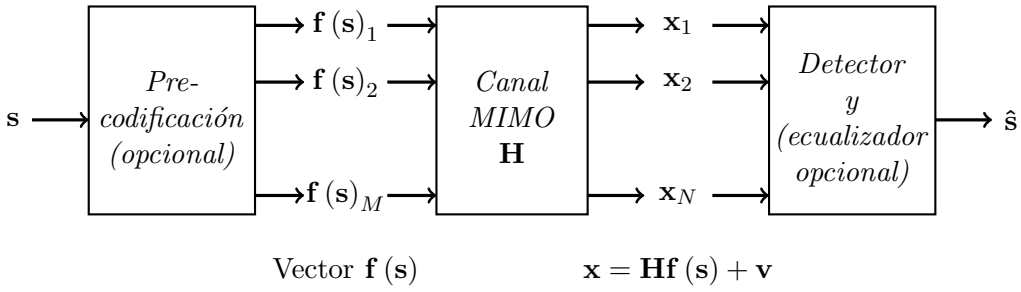


Figura 1.3: Modelo general de un sistema de comunicaciones MIMO

Después de una pre-codificación opcional se transmite, a través de las antenas transmisoras, una señal compleja $\mathbf{s}$, donde las partes reales e imaginarias de cada componente pertenecen a un conjunto

discreto finito $\mathcal{A}$. La señal viaja a través de los distintos canales del sistema, por lo que en el equipo receptor se recibe una señal $\mathbf{x}$ que es resultado de la combinación lineal de la señal transmitida, perturbada además con el ruido interferente captado en las antenas receptoras. Los datos recibidos desde las salidas del canal deben deshacerse en recepción para obtener una buena estimación de los símbolos transmitidos. Dicha estimación es representada usualmente por el vector $\hat{\mathbf{s}}$.

Existen diversas formas de realizar la pre-codificación opcional al vector $\mathbf{s}$: de forma directa, mediante una pre-codificación lineal o mediante una pre-codificación no lineal. Suponiendo que la señal es transmitida de manera directa sin pre-codificación, la relación entrada-salida del sistema sería:

$$\mathbf{x} = \mathbf{H}\mathbf{s} + \mathbf{v} \tag{1.2}$$

Por razones computacionales, este modelo complejo se transforma a un modelo real equivalente que conlleva entonces a la resolución de (1.1).

Dado que encontrar la solución ML mediante búsqueda exhaustiva no es posible en términos prácticos, a lo largo de los últimos años ha existido un gran interés por parte de investigadores académicos e industriales en soluciones de menor complejidad [68]. De esta forma han surgido métodos llamados sub-óptimos o heurísticos que se basan en la inversión de la matriz de canales para encontrar la solución y que presentan una complejidad temporal cúbica; y existen por otro lado los llamados métodos óptimos que sí obtienen la solución exacta pero a un coste computacional mayor (aunque polinómica para ciertos márgenes de relación señal-ruido [62, 131]), pues basan su funcionamiento en el recorrido de un árbol de posibles soluciones siguiendo una estrategia *Branch-and-Bound*. En esta última categoría pertenecen los métodos *Sphere-Decoding*[41, 107, 3, 112], que forman parte de las principales motivaciones de la presente tesis.

Las ganancias en capacidad de transmisión que se pueden alcanzar en los sistemas MIMO utilizando los algoritmos de detección mencionados anteriormente, se obtienen a cambio de un aumento de la complejidad computacional en el receptor. La implementación de estos métodos para arquitecturas convencionales secuenciales tiene el inconveniente de que no es capaz de alcanzar los requerimientos de prestaciones que son necesarios en sistemas de comunicación de banda ancha. Teniendo en cuenta

que en muchos casos la decodificación en sistemas MIMO requiere soluciones en tiempo real, es una necesidad imperativa transportar los modelos de solución hacia arquitecturas paralelas con el objetivo de reducir los tiempos de respuesta en los receptores. Dado que existen métodos que pueden resolver el problema (1.1) con complejidad polinómica es posible entonces la aplicación efectiva del paralelismo (en caso que las soluciones secuenciales tuvieran una complejidad exponencial haría falta un número exponencial de procesadores). Con los significativos avances en la tecnología VLSI (*Very Large Scale Integration*) los computadores paralelos con cientos de elementos de proceso están cada vez más accesibles desde el punto de vista económico, a costos muy por debajo que hace una década atrás. Esta tecnología ha impulsado el interés por el uso del procesamiento paralelo en aplicaciones que requieren resolver problemas de optimización basados en algoritmos de búsqueda [133].

Haciendo un resumen de las consideraciones realizadas, la presente tesis se centra en el desarrollo de variantes de paralelización de métodos de Búsqueda Directa y métodos *Sphere-Decoding* para resolver el problema de mínimos cuadrados en enteros que surge en la decodificación de señales en sistemas inalámbricos MIMO. La paralelización está dirigida para distintas arquitecturas, bien sea arquitecturas con memoria compartida, memoria distribuida y esquemas híbridos. Adicionalmente se intentó mejorar el rendimiento de los algoritmos no sólo mediante el recurso del paralelismo, sino proponiendo mejoras en los propios algoritmos secuenciales. En la optimización por Búsqueda Directa se diseñaron e implementaron diversas variantes, las cuales tuvieron muy buenos resultados en la resolución de un problema complejo como es el Problema Inverso Aditivo de Valores Singulares [27], pues lograron converger y obtener mejor precisión en la solución que los métodos basados en derivadas tipo Newton. Este hecho sirvió de premisa para aplicar los algoritmos diseñados al problema de mínimos cuadrados discretos, donde sólo fue necesario modificar los parámetros iniciales de los algoritmos. Como se verá en capítulos posteriores, los resultados de la Búsqueda Directa en la decodificación de señales son alentadores, pues lograron alcanzar en la generalidad de las experimentaciones realizadas la solución ML empleando tiempos menores que otras variantes conocidas de algoritmos de solución exacta. Por su parte, en los métodos *Sphere-Decoding*, se realiza un aporte al proponer el uso de la descomposición de valores singulares (SVD - *Singular Value Decomposition*) para obtener radios que estrechen un poco más el espacio de búsqueda de la solución, y para obtener un orden de decodificación

que implique una menor cantidad de nodos a recorrer en el árbol de búsqueda. La idea de aplicar la SVD en estos métodos es por la abundante y útil información que brinda esta descomposición desde el punto de vista geométrico sobre la retícula generada.

En sentido general, tanto para los métodos de Búsqueda Directa como para los métodos *Sphere-Decoding* se realiza un amplio estudio del estado del arte publicado en trabajos recientes, se proponen algunas variantes en los métodos con el objetivo de disminuir los costos temporales y obtener mayor calidad en la solución obtenida, y se logran desarrollar rutinas portables tanto secuenciales como paralelas. Las librerías están diseñadas e implementadas con un alto grado de abstracción y encapsulamiento de modo que puedan ser usadas no sólo para solucionar el problema (1.1) sino que permiten abordar cualquier problema de optimización numérica con estos métodos.

1.2 Objetivos

En este trabajo se tiene como objetivo general diseñar, implementar y evaluar algoritmos secuenciales y paralelos para resolver eficientemente el problema de mínimos cuadrados en espacios discretos, particularmente el problema aplicado a la decodificación de señales en sistemas inalámbricos MIMO. En función del objetivo general planteado, nos trazamos los siguientes objetivos específicos:

- Estudio del problema matemático de mínimos cuadrados en enteros, haciendo énfasis en el estudio del problema desde el punto de vista geométrico, o sea, en el Problema del Punto más Cercano en Retículas (CVP, *Closest Vector Problem*)

- Estudio de los principales algoritmos planteados en la literatura que resuelven el problema enunciado, especialmente aquellos que realizan la búsqueda en una estructura de árbol, siguiendo un estrategia de Ramificación y Poda (*Branch-and-Bound*), y que tienen como interpretación geométrica la búsqueda del punto más cercano en el interior de una hiper-esfera.

- Estudio detallado de los principales algoritmos de Búsqueda Directa. Comparación de estos algoritmos entre sí, y con los algoritmos que usan derivadas.

- Desarrollo de una librería portable y optimizada de métodos de Búsqueda Directa implementados

de forma secuencial y paralela para arquitecturas de memoria distribuida, que pueda ser usada en la solución de diversos problemas de optimización. Las variantes diseñadas e implementadas deben ser evaluadas y probadas para escoger las de mejor rendimiento.

- Aplicación de los métodos de Búsqueda Directa implementados en problemas concretos de ingeniería o de álgebra lineal numérica, especialmente en el problema de mínimos cuadrados en espacios discretos, comparándolos en todo momento con métodos diseñados específicamente para cada problema en cuestión.

- Desarrollo de una librería portable y eficiente de métodos tipo *Sphere-Decoding* implementados de forma secuencial y paralela siguiendo el esquema *Branch-and-Bound*. En las versiones secuenciales han de proponerse mejoras en algunas fases de estos métodos, como pueden ser la selección de radios de hiper-esfera que acoten el espacio de búsqueda y el orden en que las componentes de la solución deben ser determinadas. Las versiones paralelas han de poder ejecutarse en arquitecturas de memoria compartida, distribuida así como en esquemas compuestos por multiprocesadores conectados en red. Las variantes diseñadas e implementadas deben ser evaluadas y probadas para escoger las de mejor rendimiento.

Durante el transcurso de la tesis no se debe obviar la realización de un estudio de las arquitecturas paralelas actuales, así como de las herramientas software necesarias para su máximo aprovechamiento. Como herramientas hardware se ha contado con clusters de multiprocesadores, con los cuales se puede hacer paralelismo a varios niveles. Las herramientas software fueron seleccionadas siguiendo criterios de portabilidad, robustez y eficiencia, y en tal sentido se escogieron las bibliotecas estándar BLAS y LAPACK para el cálculo algebraico, el entorno MPI para la programación en memoria distribuida y la librería de directivas de compilación OpenMP para el programación en memoria compartida. Además de ello el código está implementado en el lenguaje de programación ANSI C, y el entorno utilizado trabaja en el sistema operativo LINUX/UNIX.

1.3 Estructura del documento de la tesis doctoral

Este documento de tesis está estructurado de la siguiente forma:

- En el capítulo 2 se presentan, introducen y fundamentan las herramientas de computación paralela utilizadas durante el desarrollo de la investigación. Primeramente se mencionan los distintos índices de prestaciones empleados en el análisis de los algoritmos paralelos, posteriormente se verán las características de cada una de las herramientas hardware, y se finaliza con la descripción del conjunto de librerías y entornos usados en la programación de las rutinas.

- El capítulo 3 muestra un estado del arte del problema de mínimos cuadrados discreto y sus métodos de solución. Se plantea la formulación del problema desde el punto de vista geométrico, abordando el Problema del Punto más Cercano en retículas, además de la descripción de cómo se llega a dicho problema a partir de la decodificación de señales en los sistemas de comunicaciones inalámbricos MIMO. Se hace un estudio de los métodos de solución que resultaron de interés nuestro, haciendo especial énfasis en los métodos de búsqueda por árbol conocidos como *Sphere-Decoding*.

- En el capítulo 4 se abordan los métodos de optimización basados en Búsqueda Directa. Se da un repaso al estado del arte en el campo de la optimización mediante los métodos de Búsqueda Directa, donde se define lo que es la búsqueda directa, se describen las distintas clases de métodos, se describe el framework GSS (*Generating Set Search*) y se numeran los últimos avances de estos métodos en el campo de la computación paralela. Posteriormente se describen los distintos algoritmos de búsqueda directa diseñados e implementados en el trabajo, y se describe la paralelización de los algoritmos secuenciales de mejores resultados. Finalmente se exponen los resultados de estos métodos en dos problemas de optimización: El primer problema es el Problema Inverso Aditivo de Valores Singulares (PIAVS) el cual es un problema continuo, y el segundo problema es el Problema del Vector más Cercano en Retículas (CVP) que se obtiene a partir del problema de la decodificación en sistemas MIMO.

- El capítulo 5 describe el trabajo realizado con el objetivo de obtener mejoras computacionales en los métodos ML basados en búsqueda en árbol, específicamente en los métodos *Sphere-Decoding* y *Automatic Sphere-Decoding*. Es aquí donde se aborda el uso de la descomposición de valores singulares en la selección de radios de búsqueda en el SD, así como en el preprocesado y

ordenación previa.

- En el capítulo 6 se describen las versiones paralelas diseñadas e implementadas de los métodos *Sphere-Decoding* y *Automatic Sphere-Decoding*, para distintos entornos paralelos.

- El último capítulo está dedicado a plantear las conclusiones obtenidas del trabajo realizado, y adicionalmente a comentar las líneas que se recomiendan para proseguir el trabajo futuro.

2

Herramientas de Computación Paralela

En este capítulo se introducen y se fundamentan las herramientas de computación paralela utilizadas en la presente tesis. Primeramente se mencionan los distintos índices de prestaciones empleados en el análisis de los algoritmos paralelos, posteriormente se verán las características de cada una de las herramientas hardware, y se finaliza con la descripción del conjunto de librerías y entornos usados en la programación de las rutinas.

2.1 Motivación

En los últimos años proliferan cada vez más problemas que requieren un alto grado de cómputo para su resolución. La existencia de estos problemas complejos computacionalmente exige la fabricación de computadores potentes para solucionarlos en un tiempo adecuado.

Los computadores secuenciales utilizan técnicas como segmentación o computación vectorial para incrementar su productividad. También se aprovechan de los avances tecnológicos, especialmente en la integración de circuitos, que les permite acelerar su velocidad de cómputo. Actualmente se han alcanzado límites físicos en el crecimiento de las prestaciones, de modo que sobrepasarlos requerirían costes económicos no tolerables por los fabricantes y mucho menos por los usuarios. Es por ello que el paralelismo ha pasado a ser la forma más barata y consecuente de mantener la línea de incremento

de prestaciones.

Con varios computadores menos avanzados y de coste inferior pueden obtenerse buenas prestaciones al unirlos conformando un multicomputador. Además, en un computador secuencial la velocidad siempre estará limitada a la tecnología existente, por lo que tiene un tiempo limitado de vida realmente útil. En un multicomputador es fácil aumentar las prestaciones, aumentando el número de procesadores que lo forman, por encima de un computador secuencial. Los multicomputadores, o clusters de estaciones de trabajo, ofrecen una buena relación calidad/precio frente a los supercomputadores. Todo ello es posible debido al avanzado desarrollo tecnológico en las redes de interconexión, que permiten buenas velocidades de comunicación.

Por todo esto surge la necesidad de trabajar e investigar en la computación paralela, para mejorar las prestaciones obtenidas en máquinas secuenciales. Los programas paralelos son mucho más difíciles de concebir que sus homólogos secuenciales, pero sin duda ofrecerán mejores prestaciones. Se buscará siempre repartir la carga del problema entre todos los procesadores disponibles, y al estar mejor repartido el problema se resolverá en menos tiempo.

2.2 Evaluación de algoritmos paralelos

En esta sección se presentarán algunos índices que se tienen en cuenta para analizar las prestaciones de los algoritmos paralelos.

2.2.1 Tiempo de ejecución

El tiempo de ejecución es el índice de prestaciones más intuitivo. Es un parámetro absoluto pues permite medir la rapidez del algoritmo sin compararlo con otro.

En el caso de un programa secuencial, consiste en el tiempo transcurrido desde que se lanza su ejecución hasta que finaliza. En el caso de un programa paralelo, el tiempo de ejecución es el tiempo que transcurre desde el comienzo de la ejecución del programa en el sistema paralelo hasta que el último procesador culmine su ejecución [55].

Para sistemas paralelos con memoria distribuida el tiempo paralelo con p procesadores, T_P, se determina de modo aproximado mediante la fórmula

$$T_P \approx T_A + T_C - T_{SOL} \tag{2.1}$$

donde T_A es el *tiempo aritmético*, es decir, el tiempo que tarda el sistema multiprocesador en hacer las operaciones aritméticas; T_C es el *tiempo de comunicación*, o sea, el tiempo que tarda el sistema multiprocesador en ejecutar transferencias de datos; y T_{SOL} es el *tiempo de solapamiento*, que es el tiempo que transcurre cuando las operaciones aritméticas y de comunicaciones se realizan simultáneamente. El tiempo de solapamiento suele ser muchas veces imposible de calcular tanto teórica como experimentalmente, en cambio puede influir bastante en el tiempo total de ejecución del algoritmo paralelo. No obstante, la dificultad de su cálculo condiciona que se realice la aproximación:

$$T_P \approx T_A + T_C \tag{2.2}$$

El tiempo aritmético se expresa en cantidad de FLOPs, donde el FLOP es una medida que indica la velocidad que tarda el procesador en realizar una operación aritmética en punto flotante. El tiempo aritmético depende de dos parámetros de la red, uno es el tiempo de envío de una palabra, que se denota con el símbolo τ, y el otro es el tiempo que transcurre desde que el procesador decide enviar el mensaje hasta que el mensaje circula por la red, que se denota con el símbolo β.

2.2.2 Ganancia de velocidad (*Speed-Up*)

El *Speed-Up* para p procesadores, S_p, es el cociente entre el tiempo de ejecución de un programa secuencial, T_S, y el tiempo de ejecución de la versión paralela de dicho programa en p procesadores, T_P. Dado que pueden haber distintas versiones secuenciales, se elige el T_S de la versión secuencial más rápida. Este índice indica la ganancia de velocidad que se ha obtenido con la ejecución en paralelo.

$$S_P = \frac{T_S}{T_P} \tag{2.3}$$

Por ejemplo, un *Speed-Up* igual a 2 indica que se ha reducido el tiempo a la mitad al ejecutar el programa con varios procesadores.

En el mejor de los casos el tiempo de ejecución de un programa en paralelo con p procesadores

será p veces inferior al de su ejecución en un sólo procesador, teniendo todos los procesadores igual potencia de cálculo. Es por ello que el valor máximo que puede alcanzar el *Speed-Up* de un algoritmo paralelo es p. Generalmente el tiempo nunca se verá reducido en un orden igual a p, ya que hay que contar con la sobrecarga extra que aparece al resolver el problema en varios procesadores, debido a sincronizaciones y dependencias entre ellos.

2.2.3 Eficiencia

La *eficiencia* es el cociente entre el *Speed-Up* y el número de procesadores. Significa el grado de aprovechamiento de los procesadores para la resolución del problema. El valor máximo que puede alcanzar es 1, que significa un 100 % de aprovechamiento.

$$E = \frac{S_p}{p} \tag{2.4}$$

2.2.4 Escalabilidad

La escalabilidad es la capacidad de un determinado algoritmo de mantener sus prestaciones cuando aumenta el número de procesadores y el tamaño del problema en la misma proporción. En definitiva la escalabilidad suele indicarnos la capacidad del algoritmo de utilizar de forma efectiva un incremento en los recursos computacionales. Un algoritmo paralelo escalable suele ser capaz de mantener su eficiencia constante cuando aumentamos el número de procesadores incluso a base de aumentar el tamaño del problema. Si un algoritmo no es escalable, aunque se aumente el número de procesadores, no se conseguirá incrementar la eficiencia aunque se aumente a la vez el tamaño del problema, con lo que cada vez se irá aprovechando menos la potencia de los procesadores.

La escalabilidad puede evaluarse mediante diferentes métricas [55]. Es conveniente tener en cuenta las características de los problemas con los que se está tratando para elegir la métrica adecuada de escalabilidad. En este trabajo se ha utilizado el *Speed-Up* escalado como se ha definido en [55] por su facilidad de uso cuando los resultados experimentales están disponibles. De esta forma el *Speed-Up* escalado queda definido como

$$S_P = \frac{T_S\,(kW)}{T_{kP}\,(kW)} \tag{2.5}$$

donde W sería el el costo teórico computacional del algoritmo secuencial. El comportamiento del *Speed-Up* escalado al variar k nos indica cómo cambia el *Speed-Up* cuando se escala el número de procesadores y el tamaño del problema en la misma proporción.

2.3 Herramientas hardware

En este trabajo se han utilizado multicomputadores de memoria distribuida, cuyos componentes son computadoras personales o estaciones de trabajo conectadas mediante una red de interconexión de paso de mensajes. También se ha podido contar con clusters conformados por nodos multiprocesadores, de modo que existe un paralelismo a dos niveles desde el punto de vista lógico: un nivel global que se corresponde con un esquema distribuido, pues los nodos están conectados mediante una red de interconexión, y un nivel local que se corresponde con un esquema compartido, pues los procesadores que componen cada nodo comparten una misma memoria física.

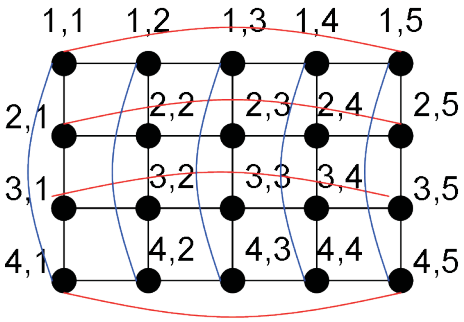
2.3.1 Máquina *Kefren*

Kefren es otro cluster de memoria compartida que trabaja bajo el ambiente Linux, Red Hat 8 (Figura 2.1(a)). Consta de 20 nodos biprocesadores Pentium Xeon a 2 Ghz, interconectados mediante una red SCI con topología de Toro 2D en malla de 4×5 (Figura 2.1(b)). Cada nodo consta de 1 Gigabyte de memoria RAM. Todos los nodos están disponibles para cálculo científico.

Se ha comprobado la latencia (β) y la tasa de transferencia de la red (τ) del cluster *Kefren*, y los resultados fueron los siguientes.

$$\tau = 6 \times 10^{-9} seg/byte$$
$$\beta = 5 \times 10^{-6} seg$$

(a) Vista frontal



(b) Red de interconexión SCI

Figura 2.1: Máquina *Kefren*

2.3.2 Máquina *Rosebud*

La máquina paralela Rosebud es un cluster heterogéneo compuesto por 6 ordenadores conectados mediante una red Fast-Ethernet. Los 6 ordenadores se agrupan en 3 pares de computadores, que se diferencian entre sí en cuanto a potencia de cálculo y en cuanto a arquitectura.

Los dos primeros computadores, identificados por los nombres `rosebud01` y `rosebud02`, presentan procesadores Pentium IV con velocidades de 1.6 GHz y 1.7 GHz respectivamente, y una memoria RAM con capacidad de 1 Gbyte. Los computadores del segundo par, llamados `rosebud03` y `rosebud04`, son biprocesadores Xeon con velocidad de procesamiento de 2.2 GHz y con 3.5 Gbyte de memoria RAM (ver Figura 2.2(a)). Estos dos primeros pares presentan una arquitectura i386.

El tercer par esta compuesto por dos estaciones Fujitsu Primergy RXI600 (`rosebud05` y `rosebud06`), cada una con 4 procesadores Dual-Core Intel Itanium®2 a 1,4 GHz compartiendo 8 GByte de memory RAM (ver Figura 2.2(b)).El procesador Itanium®2 presenta una arquitectura de 64 bits. En total, en el cluster *Rosebud* se cuenta con $1 + 1 + 2 + 2 + 8 + 8 = 22$ núcleos computacionales.

Los nodos del 1 al 4 fueron utilizados sobre todo para poner a punto las rutinas implementadas,

(a) Nodos `rosebud03` y `rosebud04`

(b) Uno de los nodos `rosebud05` y `rosebud06`

Figura 2.2: Máquina *Rosebud*

Tipo de comunicación	Tiempo de Latencia (β)	Tasa de transferencia (τ)
`rosebud05 - rosebud05`	$2{,}2 \times 10^{-5} seg$	$9{,}26 \times 10^{-11} seg/byte$
`rosebud05 - rosebud06`	$1{,}32 \times 10^{-4} seg$	$10{,}8 \times 10^{-9} seg/byte$

Tabla 2.1: Valores calculados de τ y β para los diferentes tipos de comunicación entre `rosebud05` y `rosebud06`

antes de pasar a probarlas en los nodos 5 y 6 que son más potentes.

Para este cluster se hizo un estudio para determinar los valores de los parámetros de la red. En la tabla 2.1 se muestran los resultados obtenidos para los nodos `rosebud05` y `rosebud06`, que fueron con los cuales se realizaron los experimentos que se publican en esta tesis

2.4 Herramientas software

Las herramientas software se componen por el ambiente paralelo de programación, el lenguaje de programación, y las librerías secuenciales y paralelas de álgebra lineal numérica. En nuestro caso hemos empleado el lenguaje de programación ANSI C, la librería de paso de mensajes MPI (*Message Passing Interface*) [109] para las arquitecturas de memoria distribuida, el API de multihilado directo explícito para memoria compartida OpenMP [26], y las librerías de álgebra lineal BLAS [37] y LAPACK [9].

2.4.1 Librerías de Álgebra Lineal

Con el objetivo de lograr portabilidad y eficiencia en las rutinas, tanto secuenciales como paralelas, se usaron en su implementación funciones de las librerías secuenciales estándar de álgebra lineal BLAS

(*Basic Linear Algebra Subprograms*) y LAPACK (*Linear Algebra PACKage*).

La librería BLAS se compone de funciones de alta calidad que se basan en la construcción de bloques para efectuar operaciones básicas con vectores y matrices. BLAS está construido en tres niveles: el nivel 1 que efectúa operaciones vector-vector, el nivel 2 que efectúa operaciones matriz-vector y el nivel 3 que efectúa operaciones matriz-matriz. Esta herramienta es eficiente, portable y de amplia disponibilidad, por lo que se utiliza comúnmente en el desarrollo de software del álgebra lineal de altas prestaciones.

La librería LAPACK proporciona rutinas para resolver sistemas de ecuaciones, problemas de mínimos cuadrados, problemas de valores propios, vectores propios y valores singulares. También proporcionan rutinas para factorización de matrices, tales como LU, Cholesky, QR, SVD, Schur; tanto para matrices con estructuras específicas o matrices generales.

2.4.2 Entorno de paso de mensajes MPI

El paradigma de paso de mensaje lo podemos encontrar implementado en numerosas bibliotecas, algunas implementadas para todo tipo de máquina paralela como PICL [51], PVM [50, 114], PARMACS [24], P4 [78] y MPI [38, 97]; y otras que fueron implementadas para máquinas específicas como MPL [40], NX [99] y CMMD [128].

Nuestros algoritmos se apoyan en la biblioteca MPI debido a su actualidad y disponibilidad de distribuciones en prácticamente cualquier arquitectura. Esta biblioteca proporciona tipos de datos, procedimientos y funciones con las cuales el programador puede desarrollar aplicaciones paralelas para sistemas multiprocesadores, tanto para redes locales (LAN) como para redes de área amplia (WAN). MPI ofrece portabilidad, sencillez y potencia.

MPI permite comunicaciones punto a punto mediante operaciones de envío y recepción. Éstas pueden ser bloqueantes o no bloqueantes, permitiendo estas últimas el solapamiento entre el cálculo y las comunicaciones. MPI también permite operaciones colectivas, como por ejemplo barreras de sincronización, difusiones, recolecciones, distribuciones, operaciones de reducción como máximos, mínimos, sumas, etc. Mediante los *comunicadores*, en MPI se pueden plantear subconjuntos de nodos entre los que se forman especies de subredes con semejantes características y operaciones a la red

global, así como topologías virtuales con las que se crean conexiones lógicas pre-establecidas entre ciertos nodos.

Nuestras implementaciones requerirán de MPI sólo operaciones elementales, como son envíos y recepciones bloqueantes y no bloqueantes, barreras, difusiones y recolecciones.

2.4.3 Librería OpenMP para arquitecturas de memoria compartida

El paradigma de memoria compartida es completamente distinto al de memoria distribuida. La idea básica se centra en ejecutar en una serie de hilos o *threads*, las tareas concurrentes que expresa el algoritmo paralelo, dentro del mismo espacio de memoria. El algoritmo paralelo debe controlar el acceso de lectura y escritura a la zona de variables compartidas para evitar los problemas que originan las condiciones de carrera y las dependencias de los datos, evitando, por tanto, incorrecciones en la ejecución.

En 1995, IEEE especifica el estándar API 1003.1c-1995 POSIX. También comúnmente conocido como PThreads [81], POSIX ha emergido como la API estándar para la programación multi-hilo. Sin embargo, la programación paralela con esta librería se hace a un muy bajo nivel, y por tanto es más compleja de llevar a cabo y de poner a punto. Es por ello que su uso se ha restringido principalmente para la programación de sistemas operativos, en lugar de aplicaciones.

La librería OpenMP [30, 103] es un API usado para multihilado directo explícito y paralelismo de memoria compartida. Con OpenMP se puede programar a un nivel superior si lo comparamos con los PThreads, pues con las directivas no debe preocuparse por la inicialización de los atributos de los objetos, por la configuración de algunos argumentos de los hilos, y por la partición del espacio de datos. Consta de tres componentes API principales: directivas del compilador, rutinas de librería de tiempo de ejecución y variables de entorno. Es una librería portable, pues está especificado para los lenguajes C/C++ y Fortran y se ha implementado en múltiples plataformas incluyendo la mayoría de Unix y Windows. Es además muy sencilla de usar, ya que con tan sólo 3 o 4 directivas se puede implementar un paralelismo significativo. Además, proporciona capacidad para paralelizar de forma incremental un programa secuencial, al contrario de las librerías de paso de mensajes, que requieren una aproximación todo-o-nada.

2.4.4 Esquemas paralelos híbridos

Un algoritmo paralelo concebido para una arquitectura de memoria distribuida puede, bajo ciertas condiciones, ejecutarse en un sistema que posea una arquitectura de memoria compartida, si se proveen los mecanismos necesarios para emular las comunicaciones. De manera que cada proceso se traduciría en un hilo o *thread* y los mecanismos de transmisión o recepción serían los provistos para tal efecto. A la inversa es más complejo, a veces ineficiente o incluso imposible, debido a que la traducción de los mecanismos provistos para memoria compartida no tienen un fiel reflejo en memoria distribuida.

En esta tesis empleamos, con el cluster *Rosebud*, sistemas con esquemas híbridos: memoria distribuida en los nodos de cálculo y dentro de ellos, varios elementos de proceso compartiendo memoria. De modo que podemos diseñar el algoritmo de manera regular, suponiendo que todos los elementos de proceso son independientes, pero luego implementar las comunicaciones entre elementos dentro de un mismo nodo utilizando mecanismos de memoria compartida, y entre elementos de proceso de distintos nodos, utilizando los de memoria distribuida.

3

El problema de mínimos cuadrados discreto

En este capítulo se muestra un estado del arte del problema de mínimos cuadrados discreto y sus métodos de solución. Se plantea la formulación del problema desde el punto de vista geométrico, abordando el Problema del Punto más Cercano en retículas, además de la descripción de cómo se llega a dicho problema a partir de la decodificación de señales en los sistemas de comunicaciones inalámbricos MIMO. Se hace un estudio de los métodos de solución, haciendo especial énfasis en los métodos de búsqueda por árbol.

3.1 Formulación del problema

En el capítulo 1 se realizó una breve descripción del problema que se pretender resolver en este trabajo. Se hizo mención a la analogía que existe entre el problema de decodificación de señales en los sistemas de comunicaciones inalámbricos con arquitecturas MIMO y el problema del punto más cercano en retículas, conocido por las siglas CVP (*Closest Vector Problem*). En esta sección se describirán ambos problemas de manera más formal. Se introducirán y definirán terminologías necesarias para la comprensión de los algoritmos de solución que posteriormente serán descritos en el presente capítulo.

3.1.1 Problema del Punto más Cercano en retículas

En la teoría de retículas, una *matriz generadora* B es una matriz de valores reales cuyas columnas son linealmente independientes. Sean n, m el número de filas y columnas respectivamente de B, y sea $\mathbb{R}^n$ el espacio euclideano. Una retícula en $\mathbb{R}^n$ generada por B es el conjunto:

$$\mathcal{L}(B) = \{x = (x_1, x_2, \ldots, x_n) : x = B\lambda,\ \lambda \in \mathbb{Z}^m\}$$

Las columnas de B son llamadas vectores bases de $\mathcal{L}$, y los valores de n y m son el rango y la dimensión de la retícula respectivamente. El ejemplo más sencillo de una retícula en $\mathbb{R}^n$ es precisamente el conjunto $\mathbb{Z}^n$ de todos los vectores con coordenadas enteras. En este caso, la retícula sería $\mathcal{L}(I)$ donde I es la matriz identidad.

Dos retículas son *idénticas* si todos los puntos que la componen son iguales dos a dos. Dos matrices generadoras B_1 y B_2 generan idénticas retículas $\mathcal{L}(B_1) = \mathcal{L}(B_2)$ si y sólo si

$$B_1 = B_2 W$$

donde W es una matriz cuadrada de enteros tales que $|\det W| = 1$.

Una matriz generadora B_2 es una representación *rotada y reflejada* de otra matriz generadora B_1 si

$$B_1 = QB_2$$

donde $QQ^T = I$. Si B_2 es cuadrada y triangular superior, se denomina representación triangular superior de B_1. Toda matriz generadora tiene una representación triangular superior.

Las retículas son estructuras omnipresentes en el campo de las matemáticas y han sido estudiadas de manera extensiva (ver [25] y [59] para una completa referencia). Sólo en el área de la combinatoria es posible formular varios problemas en función de retículas. La programación entera [70], la factorización de polinomios con coeficientes racionales [80] y la factorización de números enteros [106] son sólo algunas de las áreas donde se pueden presentar problemas de la teoría de retículas. En numerosos casos es necesario determinar si un cuerpo convexo en $\mathbb{R}^n$ contiene un punto perteneciente a una retícula dada. En otros casos, el interés recae en encontrar en una retícula el punto de menor longitud.

El Problema del Punto más Cercano (*Closest Vector Problem*, en abreviatura CVP) es el problema de encontrar, para una retícula $\mathcal{L}$ y un punto $x \in \mathbb{R}^n$, un vector $\hat{c} \in \mathcal{L}$ tal que

$$\|x - \hat{c}\| \leq \|x - c\|, \quad \forall c \in \mathcal{L} \tag{3.1}$$

donde $\|\cdot\|$ denota la norma euclidiana. Encontrar el punto $\hat{c}$ es equivalente a encontrar un vector $\hat{s} \in \mathbb{Z}^m$, pues $\hat{c}$ siempre se obtendría mediante una multiplicación $B\hat{s}$.

El problema CVP es uno de los problemas en retículas que son NP-*completos*, es decir, para los cuales no existe solución en un tiempo polinomial. La complejidad existente en la solución ha conllevado a la formulación de aproximaciones a este problema. Los algoritmos que le dan solución a estas aproximaciones del problema garantizan solamente que la solución encontrada está alejada del óptimo a una distancia dependiente de cierto factor dado γ. En la aproximación del problema CVP, la ecuación (3.1) quedaría modificada de la siguiente forma:

$$\|x - \hat{c}\| \leq \gamma \|x - c\|, \quad \forall c \in \mathcal{L} \tag{3.2}$$

En este caso, el factor de aproximación γ puede estar en función de cualquier parámetro asociado a la retícula, como por ejemplo su rango n, para que se cumpla el hecho de que el problema se vuelve cada vez más complejo a medida que el valor de este parámetro se incrementa.

A lo largo de los últimos 25 años se han publicado numerosos trabajos donde se analiza la complejidad computacional del problema CVP. En 1981 van Emde Boas demostró en [130] que el problema CVP es un problema NP-*duro*. En un trabajo posterior por Micciancio [89], se puede encontrar una demostración un poco más sencilla. En [10] se puede encontrar la demostración de que la aproximación del CVP es también NP-*duro* para cualquier factor constante, incluso la búsqueda de una solución sub-óptima dentro de un factor $n^{c/\log\log n}$ para cierta constante $c > 0$ es NP-*duro* [36]. Luego, en [90] se demuestra que efectivamente el problema CVP es NP-*completo* [1].

Es importante hacer mención a otro de los problemas difíciles existentes en la teoría de retículas: El Problema del Vector más Corto (*Shortest Vector Problem*, en abreviatura SVP), y que consiste en

[1] Para más información sobre las clases de complejidades ver [67]

encontrar, para una retícula $\mathcal{L}$, un vector $\hat{c} \in \mathcal{L}$, $\hat{c} \neq 0$ tal que

$$\|\hat{c}\| \leq \|c\|, \quad \forall c \in \mathcal{L} \tag{3.3}$$

Este problema también es conocido como un problema NP-*duro*[130].

3.1.2 La decodificación en los sistemas MIMO

En el capítulo 1 se dio una descripción amplia sobre los sistemas de comunicaciones inalámbricas con arquitectura MIMO, así como el proceso de transmisión/recepción de señales en estos sistemas. En esta sección se enuncian las condiciones que cumplen los parámetros involucrados en el modelo del proceso de transmisión/recepción, y se describe cómo se realiza la conversión del modelo complejo al modelo real.

Como se vio anteriormente, la relación entrada-salida del sistema viene dada por:

$$\mathbf{x} = \mathbf{H}\mathbf{s} + \mathbf{v} \tag{3.4}$$

La señal que se transmite es una señal compleja ($\mathbf{s} = [\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_M]^T \in \mathbb{C}^M$), donde las partes reales e imaginarias de cada componente pertenecen a un conjunto discreto $\mathcal{A}$. La matriz de canales $\mathbf{H}$ es una matriz general de valores complejos con N filas y M columnas, cuyas entradas $h_{ij} \sim \mathcal{N}_c(0, 1)$. El ruido interferente es un vector $\mathbf{v} \in \mathbb{C}^N$ aleatorio que sigue una distribución gaussiana con media cero y varianza σ^2.

El conjunto discreto $\mathcal{A}$, a partir del cual se escogen los símbolos a transmitir, es finito ($|\mathcal{A}| = L$), y en el campo de las telecomunicaciones es conocido como *constelación* o alfabeto de símbolos. En la presente tesis se supondrá que los símbolos pertenecen a la constelación L-PAM:

$$\mathcal{A}_L = \left\{ \frac{-L+1}{2}, \frac{-L+3}{2}, \ldots, \frac{L-3}{2}, \frac{L-1}{2} \right\} \tag{3.5}$$

donde habitualmente L es una potencia de 2. Las siglas PAM se refieren a *Pulse-Amplitude Modulation*. Para este caso la relación señal-ruido (denotado por ρ y cuyas siglas son SNR por *signal-to-noise ratio*)

y la varianza del ruido está inversamente relacionados:

$$\sigma^2 = \frac{m\left(L^2-1\right)}{12\rho} \tag{3.6}$$

Por razones computacionales, el modelo complejo (3.4) se transforma a un modelo real, en el cual el vector s de dimensión $m = 2M$, y los vectores x y v de dimensiones $n = 2N$ se definen como:

$$s = \left[\begin{array}{cc} \mathrm{Re}\left(\mathbf{s}\right)^T & \mathrm{Im}\left(\mathbf{s}\right)^T \end{array}\right]^T,\ x = \left[\begin{array}{cc} \mathrm{Re}\left(\mathbf{x}\right)^T & \mathrm{Im}\left(\mathbf{x}\right)^T \end{array}\right]^T,\ v = \left[\begin{array}{cc} \mathrm{Re}\left(\mathbf{v}\right)^T & \mathrm{Im}\left(\mathbf{v}\right)^T \end{array}\right]^T,$$

y la matriz H de dimensiones $n \times m$ como:

$$H = \left[\begin{array}{cc} \mathrm{Re}\left(\mathbf{H}\right) & \mathrm{Im}\left(\mathbf{H}\right) \\ -\mathrm{Im}\left(\mathbf{H}\right) & \mathrm{Re}\left(\mathbf{H}\right) \end{array}\right]$$

Luego, el modelo real equivalente a (3.4) está dado por:

$$x = Hs + v \tag{3.7}$$

La estimación de la señal transmitida (la obtención del vector $\hat{s}$) a partir de la señal recibida x y la matriz de canales H consiste en resolver el problema CVP.

$$\min_{s \in \mathcal{A}_L{}^m} \|x - Hs\|^2 \tag{3.8}$$

El detector óptimo minimiza el promedio de la probabilidad del error, es decir, minimiza la probabilidad $P\left(\hat{s} \neq s\right)$

3.1.3 Métodos de solución

Una primera solución aproximada para el problema CVP consiste en resolver el problema en su versión continua

$$\min_{s \in \mathbb{R}^m} \|x - Hs\|^2 \tag{3.9}$$

y luego redondear cada componente de la solución a $\mathbb{Z}$. Sin embargo, para varios problemas (especialmente cuando H está mal condicionada) se obtienen soluciones muy alejadas de la solución óptima.

Para encontrar la solución exacta al problema CVP una idea general consiste en identificar una región de $\mathbb{R}^n$ en cuyo interior se encuentre el vector solución, y luego investigar cada punto de la retícula que se encuentre dentro de dicha región. El desarrollo de los algoritmos para el CVP ha seguido dos vertientes principales, inspiradas en dos trabajos principalmente: el publicado por Pohst en 1981 [100] en el cual se propone que la región para la búsqueda del punto óptimo sea una hiperesfera, y el publicado por Kannan en 1983 [70] en el cual propone un paralelepípedo rectangular. Posteriormente en [41] y en [71] se publicaron versiones extendidas de los trabajos de Pohst y Kannan respectivamente.

Ambas estrategias se diferencian en cómo han sido presentadas en las publicaciones. Los artículos donde se trata el método de Pohst generalmente describen y analizan cuestiones de implementación del método, mientras que aquellos que tratan el método de Kannan casi siempre se enfocan en cuestiones relacionadas con su convergencia asintótica. Es por ello que en la literatura nunca se han comparado entre sí ambas estrategias.

Una condición que beneficia significativamente el rendimiento de todos los métodos (sean exactos o heurísticos), es que la matriz generadora de la retícula H sea ortogonal. En este caso incluso el método heurístico mencionado inicialmente encontraría la solución exacta del problema. Sin embargo, en la práctica las columnas de H no son ortogonales. No es viable la ortogonalización de H mediante la descomposición QR o cualquier otra descomposición porque destruye la estructura de la retícula, pues si s tiene coordenadas enteras, el vector Rs no necesariamente debe tener coordenadas enteras. Es por ello que un recurso que puede complementar cualquier propuesta de solución al problema CVP es la *reducción* de bases generadoras. La *reducción* de la base generadora de una retícula consiste en encontrar una matriz $m \times m$ invertible T, tal que T y T^{-1} tengan todas sus componentes enteras, y que la matriz $G = HT$ sea lo más ortogonal posible. Teniendo la matriz T, se resolvería entonces el problema CVP para la retícula con base G y con la solución obtenida $\hat{t}$ se obtendría la solución del problema $\hat{s}$ haciendo $\hat{s} = T\hat{t}$

De modo particular en el problema de la decodificación de señales en sistemas MIMO, que es el problema (3.8), a lo largo de los últimos años ha existido un gran interés por parte de investigadores académicos e industriales en soluciones de baja complejidad. Estos métodos sofisticados se pueden clasificar en tres tipos [68]:

- Basados en la inversión de la matriz de canal del sistema MIMO: En este caso se encuentran receptores lineales como el forzador de ceros (ZF, *Zero-Forcing*) o el de error cuadrático medio mínimo (MMSE, *Minimum Mean Square Error*) que pueden aplicarse realizando una detección lineal, con los cuales los resultados son generalmente pobres, o una detección con cancelación sucesiva de interferencias (SIC, *Succesive Interference Cancellation*). La cancelación sucesiva de interferencias puede ser ordenada (OSIC, *Ordered* SIC) con notables mejores prestaciones [61]. La característica principal de estos métodos es la inversión de la matriz de canal del sistema MIMO. Estos métodos son conocidos también como heurísticos o sub-óptimos, pues no siempre logran encontrar la solución óptima del problema, pero la complejidad temporal es $O\left(m^3\right)$.

- Basados en la búsqueda sobre una estructura de árbol: En este caso existen los métodos pertenecientes a la familia *Sphere-Decoding*. Todos ellos comparten una misma idea básica, que es la de realizar la búsqueda de la solución en el interior de una hiper-esfera con centro en el punto x, y su funcionamiento lo basan en el recorrido de un árbol de posibles soluciones. Sus diferencias radican en esencia en la forma en que recorren el árbol de posibles soluciones y en el orden en que son detectadas las componentes del vector solución $\hat{s}$. Entre estos métodos se encuentra el propuesto por Fincke y Pohst [41] y el propuesto por Schnorr y Euchner [107, 3] que realizan un recorrido en profundidad del árbol de soluciones buscando la mejor de todas; y el propuesto por Karen Su en [112] denominado *Automatic Sphere-Decoding* (ASD) donde el recorrido del árbol se realiza usando una cola de prioridades en la cual se almacenan los nodos. Todos estos algoritmos encuentran siempre la solución ML. Otros métodos que también basan su búsqueda en el recorrido de un árbol de posibles soluciones son el método K-Best [84, 85] y el método SD con terminación anticipada [12, 66], aunque estos no son métodos exactos porque no encuentran la solución ML. Los métodos SD han recibido gran atención al poder resolver el problema de la detección ML en sistemas MIMO con una complejidad polinómica en cierto margen de relación señal-ruido [62, 131]; en el peor de los casos (canales mal condicionados o SNR baja) SD puede tener una complejidad cercana a los métodos de búsqueda exhaustiva para encontrar la solución ML.

- Soluciones iterativas: Aprovechan que la mayor parte de los sistemas de comunicaciones hacen

uso de códigos para la corrección de errores. Así, en estas soluciones existe un intercambio de información *soft* o probabilística entre el detector y el decodificador, que permite refinar los resultados de cada uno de ellos en varias iteraciones, de manera que al terminar se obtienen mejores prestaciones que si cada una de estas etapas trabaja por separado. Entre los detectores hay soluciones heurísticas [111], DFE (*Decision Feedback Equalizer*) y métodos *Sphere-Decoding* modificados [132]

En las secciones siguientes se brindarán más detalles de los métodos de solución para el problema CVP aplicado a la decodificación de señales. Se abordará primeramente la reducción de bases generadoras, que aunque no es posible aplicarla en la detección en los sistemas MIMO, sí es un recurso valioso para resolver el problema CVP general.

3.2 Reducción de bases generadoras

Para cualquier retícula $\mathcal{L}$ pueden existir diferentes matrices bases generadoras. De hecho, si B es una base, también lo es $B' = BP$ para cualquier matriz P tal que tanto P como P^{-1} tengan todos sus componentes pertenecientes a $\mathbb{Z}$. Siendo $B_1, B_2, \ldots$ las matrices bases de una retícula $\mathcal{L}$, se puede suponer que existe cierta forma de ordenar o establecer un *ranking* entre dichas matrices, y de este modo una o varias matrices B_i pueden clasificarse como "buenas" bases de $\mathcal{L}$. El proceso de seleccionar buenas bases para una determinada retícula, dado cierto criterio, se llama *reducción*. En la reducción se parte de una base B de una retícula $\mathcal{L}$ y se obtiene otra base B' de $\mathcal{L}$ pero "mejor" que B según cierto criterio de reducción. Desafortunadamente no existe una noción única y claramente bien definida acerca de qué características debe cumplir una buena base, por lo que han surgido varios criterios de reducción. Por lo general, se considera una buena base para una retícula cuando sea "casi ortogonal" y sus vectores columnas sean de corta longitud.

3.2.1 Reducción de Minkowski

Uno de los primeros criterios de reducción de retículas es el propuesto por Minkowski en [91], conocido actualmente como *reducción de Minkowski*, y que establece que una base $B = [\mathrm{b}_1, \mathrm{b}_2, \ldots, \mathrm{b}_m]$

está Minkowski-reducida si para todo $i = 1, \ldots, m$ el vector b_i satisface $\|\mathrm{b}_i\| \leq \|\mathrm{b}'_i\|$ para cualquier vector b'_i tal que con la secuencia $\mathrm{b}_1, \ldots, \mathrm{b}_{i-1}, \mathrm{b}'_i$ se puede completar una base, es decir, que existen vectores $\mathrm{b}'_{i+1}, \ldots, \mathrm{b}'_m$ tal que $\mathrm{b}_1, \ldots, \mathrm{b}_{i-1}, \mathrm{b}'_i, \mathrm{b}'_{i+1}, \ldots, \mathrm{b}'_m$ es una base de $\mathcal{L}(B)$

Las matrices bases Minkowski-reducidas siempre contendrán el vector de menor longitud de la retícula. Se van seleccionando vectores bases b_i tomando el vector de menor longitud de la retícula $\mathcal{L}$ y que a su vez no sea combinación lineal de los vectores bases ya seleccionados previamente $\mathrm{b}_1, \ldots, \mathrm{b}_{i-1}$. La reducción de Minkowski ha recibido mucha atención, particularmente en la teoría de números [25, 39]. En [2] y [63] se pueden encontrar algoritmos para calcular bases Minkowski-reducidas de una determina retícula.

3.2.2 Reducción de Korkine-Zolotarev (KZ)

Otro criterio de reducción es el conocido como la reducción de Korkine-Zolotarev o reducción KZ [75]. Si bien en la reducción de Minkowski se trata de que los vectores bases tengan la menor longitud posible, en la reducción KZ se persigue que la matriz base sea lo más ortogonal posible.

Una medida que comúnmente se usa para cuantificar la ortogonalidad de una matriz B es la conocida como *defecto de ortogonalidad* [80]

$$\frac{\prod_{i=1}^{m} \|\mathrm{b}_i\|}{|\det(B)|} \tag{3.10}$$

La minimización de esta medida implica la búsqueda de una base cuyos vectores sean todos casi ortogonales. Dado que el *defecto de ortogonalidad* es proporcional al producto de las longitudes de los vectores bases, entonces se debe realizar la búsqueda de vectores casi ortogonales y a la vez con la menor norma euclidiana posible.

En su definición, la reducción KZ se basa en el proceso de ortogonalización de Gram-Schmidt: Para toda secuencia de vectores $\mathrm{b}_1, \ldots, \mathrm{b}_m$ se definen los correspondientes vectores $\mathrm{b}^*_1, \ldots, \mathrm{b}^*_m$ como

$$\mathrm{b}^*_i = \mathrm{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathrm{b}^*_j \tag{3.11a}$$

$$\mu_{i,j} = \frac{\langle \mathrm{b}_i, \mathrm{b}_j^* \rangle}{\langle \mathrm{b}_j^*, \mathrm{b}_j^* \rangle} \tag{3.11b}$$

donde $\langle \mathrm{x}, \mathrm{y} \rangle = \sum_{i=1}^{n} x_i y_i$. Para cada i, b_i^* es la componente de b_i ortogonal al espacio generado por los vectores $\mathrm{b}_1, \ldots, \mathrm{b}_{i-1}$. En particular, el espacio generado por $\mathrm{b}_1, \ldots, \mathrm{b}_i$ es el mismo que el generado por $\mathrm{b}_1^*, \ldots, \mathrm{b}_i^*$ y $\langle \mathrm{b}_i^*, \mathrm{b}_j^* \rangle = 0$ para todo $i \neq j$.

Siendo B^* la correspondiente ortogonalización de Gram-Schmidt de una base B, la base B está KZ-reducida si y sólo si para todo $i = 1, \ldots, m$

- b_i^* es el vector de menor longitud en $\pi_i(\mathcal{L}(B))$
- para todo $j < i$ los coeficientes Gran-Schmidt $\mu_{i,j}$ de B satisfacen $|\mu_{i,j}| \leq 1/2$

donde la función

$$\pi_i(\mathrm{x}) = \sum_{j \leq i} \left(\frac{\langle \mathrm{x}, \mathrm{b}_j^* \rangle}{\|\mathrm{b}_j^*\|^2} \right) \mathrm{b}_j^* \tag{3.12}$$

proyecta ortogonalmente al vector x en el espacio generado por $\mathrm{b}_1^*, \ldots, \mathrm{b}_i^*$

Otra forma para determinar si una matriz generadora está reducida según el criterio KZ parte del estudio de la representación triangular superior de la base. Sea

$$U(B) = [\mathrm{u}_1, \mathrm{u}_2, \ldots, \mathrm{u}_m] = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1m} \\ 0 & u_{22} & \cdots & u_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{mm} \end{bmatrix}$$

la representación triangular superior de una base generadora B. La matriz $U(B)$ está KZ-reducida si $n = 1$ o si se cumplen cada una de las siguientes condiciones:

1. u_1 es la solución del problema SVP en la retícula $\mathcal{L}(U(B))$
2. $|u_{1k}| \leq \frac{|u_{11}|}{2}$ para $k = 2, \ldots, m$
3. La submatriz $\begin{bmatrix} u_{22} & \cdots & u_{2m} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & u_{mm} \end{bmatrix}$ es KZ-reducida.

Se completa el análisis teniendo en cuenta que una matriz generadora es KZ-reducida si y sólo si su representación triangular inferior es KZ-reducida.

En la reducción de Minkowski, los vectores bases b_i se añaden a la matriz base sólo si es el vector de menor longitud que permite extender la matriz base. En la reducción KZ la selección del vector b_i se basa en su longitud dentro del complemento ortogonal del espacio generado por los vectores previamente determinados $b_1, \ldots, \mathrm{b}_{i-1}$. Esta definición de la reducción KZ es intrínsicamente secuencial, pues el cálculo de $\|\mathrm{b}_i^*\|$ depende directamente de los vectores $\mathrm{b}_1, \ldots, \mathrm{b}_{i-1}$.

3.2.3 Reducción de Lenstra-Lenstra-Lovasz (LLL)

Aunque tanto la reducción de Minkowski como la reducción de Korkine-Zolotarev proveen un marco para el estudio de la reducción de bases generadoras de retículas, el cálculo de una base reducida usando cualquiera de estos dos criterios es en general un problema difícil. Actualmente no se conocen algoritmos con complejidad polinomial que calculen reducciones de una retícula $\mathcal{L}$ que cumplan las condiciones estrictas de la reducción de Minkowski o la reducción KZ. Si existiera un algoritmo que calcule alguna de estas reducciones, entonces se tendría la capacidad de determinar en una retícula el vector de menor norma euclidiana en un tiempo polinomial, simplemente calculando la base reducida y el vector b_1 sería el vector buscado.

La reducción Lenstra-Lenstra-Lovasz (LLL) [80] fue publicada en 1982 en el campo de la factorización de polinomios de coeficientes racionales. Además de aportar otro criterio de reducción de matrices bases, el aporte realmente significativo consiste en un algoritmo con complejidad polinomial que realiza la transformación de una base $B = [\mathrm{b}_1, \mathrm{b}_2, \ldots, \mathrm{b}_m]$ a una base $B' = [\mathrm{b}'_1, \mathrm{b}'_2, \ldots, \mathrm{b}'_m]$ LLL-reducida.

Una base $B = [\mathrm{b}_1, \mathrm{b}_2, \ldots, \mathrm{b}_m]$ está LLL-reducida con parámetro δ $(1/4 < \delta < 1)$ si se cumplen las siguientes condiciones:

1. $|\mu_{i,j}| \leq 1/2$ para todo $i > j$, donde $\mu_{i,j}$ son los coeficientes de la ortogonalización de Gram-Schmidt de la base B.

2. para cualquier par de vectores consecutivos b_i, b_{i+1} se cumple

$$\delta \left\| \pi_i \left(\mathrm{b}_i \right) \right\|^2 \leq \left\| \pi_i \left(\mathrm{b}_{i+1} \right) \right\|^2 \tag{3.13}$$

El algoritmo para la reducción LLL convierte una base B en otra B' realizando dos tipos de transformaciones. La primera transformación realiza una reducción de tamaño de los vectores bases. En este paso el algoritmo LLL busca el mayor valor de j para el cual exista un $i > j$ y además que $\mu_{i,j}$ viole la condición 1 ($|\mu_{i,j}| > 1/2$). Siendo $\lfloor \mu_{i,j} \rceil$ el entero más cercano a $\mu_{i,j}$, se realiza la transformación

$$\mathrm{b}_i \leftarrow \mathrm{b}_i - \lfloor \mu_{i,j} \rceil \mathrm{b}_j$$

los valores $\mu_{i,k}$, con $k < j$, se sustituyen por $\mu_{i,k} - \lfloor \mu_{i,j} \rceil \mu_{j,k}$, y el valor $\mu_{i,j}$ se sustituye por $\mu_{i,j} - \lfloor \mu_{i,j} \rceil$. Después de estos cambios, las condiciones 1 y 2 se cumplen [80]. El segundo tipo de transformación consiste en el intercambio de columnas de la matriz B. Aquí el algoritmo LLL busca el menor valor de i para el cual los vectores b_i y b_{i+1} no cumplen la condición 2, y por tanto realiza el intercambio $\mathrm{b}_i \rightleftarrows \mathrm{b}_{i+1}$ para forzar el cumplimiento de dicha condición. El algoritmo LLL realiza alternadamente estas dos transformaciones hasta que las condiciones 1 y 2 se cumplan al mismo tiempo.

En el algoritmo 1 se muestran formalmente los pasos de la reducción LLL.

En [90] se enuncia y se demuestra el teorema que plantea que existe un algoritmo de complejidad polinomial que calcula la reducción LLL con el parámetro $\delta = (1/4) + (3/4)^{m/(m-1)}$. Aparejado a este resultado, está el hecho de que existe una relación entre el parámetro δ de la reducción LLL y la longitud del primer vector de la base B. Esta relación viene dada por

$$\left\| \mathrm{b}_1 \right\| \leq \left(2 \Big/ \sqrt{4\delta - 1} \right)^{m-1} \lambda_1 \tag{3.14}$$

donde λ_1 es la longitud del vector más corto de la retícula. Siendo $\delta = (1/4) + (3/4)^{m/(m-1)}$ entonces se obtiene que $\left\| \mathrm{b}_1 \right\| \leq \left(2/\sqrt{3} \right)^m \lambda_1$. De modo que el vector b_1 es una aproximación a la solución del problema SVP con factor $\left(2/\sqrt{3} \right)^m$.

Algoritmo 1 Reducción LLL

Entrada: Base $B = [\mathrm{b}_1, \mathrm{b}_2, \ldots, \mathrm{b}_m]$ de una retícula $\mathcal{L}$
Salidas: Una base B de $\mathcal{L}$ LLL-reducida
La matriz de transformación $T = [\mathrm{t}_1, \mathrm{t}_2, \ldots, \mathrm{t}_m]$ con componentes enteras.

1: $T \leftarrow I$ {*se inicializa* T *con la matriz identidad*}
2: **para** $i = 1, \ldots, m$ **hacer**
3: **para** $j = i - 1, \ldots, 1$ **hacer**
4: $c_{ij} \leftarrow \lfloor \langle \mathrm{b}_i, \mathrm{b}_j \rangle / \langle \mathrm{b}_j, \mathrm{b}_j \rangle \rceil$
5: $\mathrm{b}_i \leftarrow \mathrm{b}_i - c_{ij}\mathrm{b}_j$
6: $\mathrm{t}_i \leftarrow \mathrm{t}_i - c_{ij}\mathrm{t}_j$
7: **fin para**
8: **fin para**
9: **si** $\delta \left\| \pi_i \left(\mathrm{b}_i \right) \right\|^2 > \left\| \pi_i \left(\mathrm{b}_{i+1} \right) \right\|^2$ para algún i, $1 \leq i \leq m$ **entonces**
10: Intercambiar b_i con b_{i+1}
11: Intercambiar t_i con t_{i+1}
12: Ir a 2
13: **sino**
14: Retornar B y T
15: **fin si**

Con el surgimiento del algoritmo LLL se logró soluciones en tiempo polinomial a aproximaciones de los problemas SVP y CVP. En [11] se presenta y analiza un algoritmo, nombrado Algoritmo del Plano más Cercano (*nearest plane algorithm*), que usa la reducción LLL para resolver el problema CVP aproximado dentro de un factor $2\left(2/\sqrt{3}\right)^m$.

Sobre el algoritmo LLL se han publicado mejoras en [108] y [105], además de que ha sido modificado varias veces [28]. En [104] se presenta una algoritmo de reducción que incluye por un lado la reducción LLL y por otro la reducción KZ. El algoritmo de reducción que se propone en dicho trabajo, recibe el nombre de *Block Korkine-Zolotarev* (BKZ) y combina las reducciones LLL y KZ. Primero reduce, usando el criterio KZ, un conjunto de vectores consecutivos y luego aplica el criterio LLL para reducir los bloques de vectores. A medida que aumenta el tamaño de los bloques, las reducciones obtenidas por el algoritmo son mejores, pero a un costo de tiempo mayor, llegando incluso a consumir un tiempo exponencial. Con el algoritmo BKZ se puede mejorar el factor de aproximación al problema CVP a $2^{O\left(n(\ln\ln m)^2/\ln m\right)}$, y si se usa el algoritmo probabilístico publicado en [7] en la reducción BKZ, este factor puede ser reducido a $2^{O(m\ln\ln m/\ln m)}$

3.2.4 Otros criterios de reducción de bases

En [93] se define a una base B como θ-ortogonal si el ángulo entre cada vector y el subespacio determinado por el resto de los vectores es al menos θ. Aquellas bases θ-ortogonales donde θ es al menos $\frac{\pi}{3}$ las llaman *casi ortogonales*. En dicho trabajo se demostró que una base $\frac{\pi}{3}$-ortogonal siempre contiene el vector distinto de cero de menor longitud de la retícula. De esta manera el problema SVP pasa a ser trivial para las bases $\frac{\pi}{3}$-ortogonales, tal y como es para las bases ortogonales.

Posteriormente en [32] se demuestra que las bases $\frac{\pi}{3}$-ortogonales pueden ser reducidas según el criterio KZ en un tiempo polinomial. También realizan un estudio relacionado con las longitudes de los vectores de las bases θ-ortogonales, donde plantean que si todos los vectores bases tienen longitudes no mayores a $\frac{2}{\cos\theta}$ veces la longitud del vector solución del problema SVP, entonces la base está reducida según el criterio de Minkowski.

3.3 Métodos heurísticos

Debido a la alta complejidad que presenta el problema CVP, en el campo de las comunicaciones inalámbricas han surgido técnicas heurísticas o de aproximación, con el objetivo de que el problema pueda resolverse en un tiempo computacionalmente razonable. Dentro de estos, los más comunes en la decodificación de señales en los sistemas con arquitectura MIMO son el ZF (*Zero Forcing*) y el MMSE (*Minimum Mean Square Error*), que puede aplicarse realizando una detección lineal o una detección SIC (*Succesive Interference Cancellation*). Se puede realizar a su vez una ordenación previa de los símbolos a detectar, teniendo así una detección por cancelación sucesiva de interferencias ordenada (OSIC).

A continuación se verá más detalladamente cada uno de los métodos.

3.3.1 Zero-Forcing

El funcionamiento de este algoritmo de detección consiste en realizar una inversión de la matriz de canales del sistema MIMO, para satisfacer el criterio del *Zero-Forcing* por medio del cálculo de la pseudoinversa del canal H. Una vez calculada la pseudoinversa de H, la estimación de los símbolos

transmitidos se obtendría mediante:

$$\hat{s} = \left\lfloor H^{\dagger} x \right\rceil_{\mathcal{A}} \tag{3.15}$$

donde $H^{\dagger}$ denota la pseudo-inversa de H ($H^{\dagger} = \left(H^T H\right)^{-1} H^T$), y $\lfloor \cdot \rceil_{\mathcal{A}}$ denota la operación de aproximar cada elemento del vector obtenido al elemento más cercano en el conjunto discreto $\mathcal{A}$. El punto $\hat{s}$ obtenido por el método ZF es también conocido como el punto de Babai [58].

La complejidad del método ZF está esencialmente determinada por la complejidad del cálculo de la pseudo-inversa de la matriz H. Una variante para evitar el cálculo de la pseudo-inversa es emplear la factorización QR, $H = QR$, donde Q es una matriz ortogonal $n \times m$ ($Q^T = Q^{-1}$) y R es una matriz triangular superior $m \times m$ [54]. El algoritmo ZF usando la descomposición QR sería el siguiente:

Algoritmo 2 ZeroForcing(H, x)

Entrada: Una matriz $H \in \mathbb{R}^{n \times m}$ y un vector $x \in \mathbb{R}^n$
Salida: Un vector $\hat{s} \in \mathcal{A}^m$

1: $[Q, R] \leftarrow$ Factorización QR de H
2: $y \leftarrow Q^T x$
3: Resolver el sistema triangular $Rs = y$
4: **para** $i = 1, \ldots, m$ **hacer**
5: $\quad \hat{s}_i \leftarrow \lfloor s_i \rceil_{\mathcal{A}}$
6: **fin para**

También se puede usar la descomposición de valores singulares (SVD) en lugar de la descomposición QR. En cualquier caso, la complejidad del método ZF es de orden cúbico en relación al tamaño de la matriz H.

Para matrices H mal condicionadas el detector ZF solamente funciona bien en la región donde la relación señal-ruido (SNR) es alta, es decir, cuando en la transmisión apenas hubo ruido interferente, y por tanto la señal x apenas fue perturbada. Con una SNR baja, se produce con el ZF una amplificación del ruido considerable.

3.3.2 Zero-Forcing SIC

Para aumentar las prestaciones del receptor Zero Forcing tradicional se puede utilizar técnicas no lineales consistentes en la cancelación sucesiva de símbolos interferentes (SIC). Se considerará la señal

proveniente de cada antena transmisora como la deseada y el resto como interferencias. Los símbolos detectados de cada antena transmisora se van eliminando del vector de señal recibida, así la siguiente señal a ser detectada verá una señal de interferencia menos.

Este método toma del punto de Babai calculado sólo el valor de una de sus componentes, digamos $\hat{s}_m$. De modo que se supone correctamente estimado el valor de $\hat{s}_m$ y se cancela su efecto para obtener un problema entero de mínimos cuadrados con $m-1$ indeterminadas. El proceso se repite para calcular $\hat{s}_{m-1}$ y así sucesivamente. En la literatura también es conocida esta técnica como *Nulling-and-Cancelling*, y en el ámbito de las comunicaciones se conoce también como *decision-feedback equalization* (DFE).

Luego de la descomposición QR de la matriz H, tenemos que el modelo de transmisión es el siguiente:

$$x = QRs + v \tag{3.16}$$

Multiplicando por Q^T por la izquierda a ambos miembros, se llega a otro modelo de transmisión equivalente:

$$y = Rs + w \tag{3.17}$$

donde $y = Q^T x$ y $w = Q^T v$. Debido a que Q^T es ortogonal, el nuevo vector ruido w mantiene las propiedades de ser Gaussiano con media nula y varianza unidad. De forma explícita quedaría (3.17) como:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & \dots & r_{1m} \\ 0 & r_{22} & \cdots & r_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & r_{mm} \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_m \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \tag{3.18}$$

Se observa en (3.18) que y_m es solamente una versión escalada ruidosa de s_m que puede ser estimada directamente, aprovechando que R es triangular superior. Realizada la estimación de s_m, se actualizan para la siguiente iteración las componentes y_k haciendo $y_k \leftarrow y_k - r_{k,m}\hat{s}_m$.

Algoritmo 3 ZeroForcingSIC(H, x)

Entrada: $H \in \mathbb{R}^{n \times m}$ la matriz de canal del sistema
$x \in \mathbb{R}^n$ el vector recibido.
Salida: Un vector $\hat{s} \in \mathcal{A}^m$

1: $[Q, R] \leftarrow$ Factorización QR de H
2: $y \leftarrow Q^T x$
3: **para** $i = m, m-1, \ldots, 1$ **hacer**
4: $\quad \hat{s}_i \leftarrow \lfloor y_i / R_{i,i} \rceil_{\mathcal{A}}$
5: $\quad$ **para** $j = 1, \ldots, i-1$ **hacer**
6: $\quad\quad y_j \leftarrow y_j - R_{j,i}\hat{s}_i$
7: $\quad$ **fin para**
8: **fin para**

3.3.3 MMSE OSIC

El esquema SIC anteriormente explicado tiene el incoveniente de que un símbolo estimado erróneamente provocaría un error en la estimación de los siguientes símbolos, ya que dependen de él para ser detectados. El esquema OSIC propone detectar en primer lugar el símbolo con una mayor relación señal-ruido, es decir, el símbolo que se ha recibido con mayor fiabilidad. Dicho símbolo detectado, como en el caso sin ordenación, se extrae del resto de las señales recibidas y se cancela su efecto. Siendo $\hat{s}_k$ el símbolo detectado en una primera instancia, se tiene un sistema con $m-1$ columnas, donde la nueva matriz de canal se obtiene eliminándole la columna k a la matriz H. De la misma forma se procede para detectar el resto de los símbolos.

La operación de anular los símbolos se puede realizar usando el criterio ZF aunque presenta varios problemas. Por una parte puede encontrar matrices singulares no invertibles y por tanto no proporcionar una solución. Y por otra parte, el detector ZF sólo cancela las interferencias pero no del ruido, y por eso lo aumenta notablemente. Para evitar estos problemas una alternativa es usar el criterio MMSE, que anula la componente con menor error cuadrático medio.

La idea de los esquemas de detección MMSE es incluir en el proceso de decodificación información del ruido interferente en la transmisión. En estos esquemas se propone extender la matriz de canales

H a una matriz H_α de $n+m$ filas y m columnas mediante

$$H_\alpha = \begin{pmatrix} H \\ \sqrt{\alpha} I \end{pmatrix} \tag{3.19}$$

La estimación MMSE proporciona entonces la siguiente solución:

$$\hat{s} = \left\lfloor H_\alpha^\dagger x \right\rceil_{\mathcal{A}} \tag{3.20}$$

donde $\alpha = 1/\rho$ siendo ρ el valor del SNR de la transmisión.

Para desarrollar la cancelación sucesiva de interferencias de manera ordenada, el método MMSE-OSIC se basa en la matriz de covarianza del error en la estimación de la solución (que es una matriz simétrica y definida positiva)

$$P = E\left\{(s-\hat{s})(s-\hat{s})^*\right\} = \begin{bmatrix} H \\ \sqrt{\alpha} I \end{bmatrix}^\dagger \left(\begin{bmatrix} H \\ \sqrt{\alpha} I \end{bmatrix}^\dagger \right)^* = (\alpha I + H^* H)^{-1}$$

Si $P_{jj} = \text{mín}\left\{diag\,(P)\right\}$, entonces s_j es la componente de s con mayor señal-ruido. Se intercambiarían entonces las filas j y m de la matriz P y, consecuentemente, se debe hacer el mismo intercambio de filas en s y el correspondiente intercambio de columnas en H y de filas en $H_\alpha^\dagger$. La componente s_m puede ser decodificada como

$$\hat{s}_m = \left\lfloor H_{\alpha,m}^\dagger x \right\rceil_{\mathcal{A}} \tag{3.21}$$

donde $H_{\alpha,j}^\dagger$ es la j-ésima fila de $H_\alpha^\dagger$ (denominado el vector anulador). Si $H = (\mathrm{h}_1, \mathrm{h}_2, \ldots, \mathrm{h}_m)$ entonces se puede cancelar el efecto de la estimación de s_m en la señal recibida haciendo

$$x' = x - \mathrm{h}_m \hat{s}_m \tag{3.22}$$

El procedimiento debe ser repetido para la matriz reducida H', obtenida eliminando la m-ésima columna de H. Ello implica que para la próxima iteración se debe recalcular la pseudoinversa, esta vez de H'. A continuación se muestran los pasos del algoritmo

Algoritmo 4 MMSE-OSIC(H, α, x)

Entrada: $H \in \mathbb{R}^{n\times m}$ la matriz de canal del sistema
α el valor recíproco del SNR
$x \in \mathbb{R}^n$ el vector recibido.
Salida: La estimación de la señal transmitida $\hat{s} \in \mathcal{A}^m$

1: $H^{(m)} \leftarrow H$
2: **para** $k = m, m-1, \ldots, 1$ **hacer**
3: $\quad H_\alpha^{(k)} \leftarrow \begin{bmatrix} H^{(k)} \\ \sqrt{\alpha} I_k \end{bmatrix}$
4: $\quad P^{(k)} \leftarrow \left(\alpha I_k + \left(H^{(k)}\right)^* H^{(k)}\right)^{-1}$
5: $\quad$ **si** $P_{jj}^{(k)} = \text{mín}\left\{diag\left(P^{(k)}\right)\right\}$ **entonces**
6: $\quad\quad$ Intercambiar las filas j y k de $P^{(k)}$, de s y de $H_\alpha^{(k)\dagger}$
7: $\quad\quad$ Intercambiar las columnas j y k de $H^{(k)}$
8: $\quad$ **fin si**
9: $\quad \hat{s}_k \leftarrow \left\lfloor H_{\alpha,k}^{(k)\dagger} x \right\rceil_{\mathcal{A}}$
10: $\quad x^{(k-1)} \leftarrow x^{(k)} - H_k^{(k)} \hat{s}_k$ $\{H_k^{(k)}$ *es la fila k-ésima de* $H^{(k)}\}$
11: $\quad H^{(k-1)}$ es el resultado de eliminar la k-ésima columna de $H^{(k)}$
12: **fin para**

La complejidad computacional del método MMSE-OSIC está condicionada fundamentalmente por el cálculo de la pseudoinversa en cada iteración. Cuando $m = n$, se necesita calcular la pseudoinversa de una serie de matrices con dimensiones m, $m-1$, …, 1. La complejidad computacional del cálculo de la pseudoinversa de esta serie de matrices tiene un orden $O\left(m^4\right)$, un grado mayor que el orden de la complejidad del método ZF. Este cálculo iterativo se evita en [61], donde se propone una variante denominada *Square-Root* que pretende reducir el número de cálculos aritméticos, sobre todo en los pasos más complejos computacionalmente, como es el cálculo de $P^{(k)}$ y de $H_\alpha^{(k)\dagger}$.

Para ello realizan la descomposición QR de la matriz H_α de (3.19)

$$\begin{bmatrix} H \\ \sqrt{\alpha} I \end{bmatrix} = QR = \begin{bmatrix} Q_\alpha \\ Q_2 \end{bmatrix} R \tag{3.23}$$

Luego,

$$P = (\alpha I + H^* H)^{-1} = (R^* R)^{-1}$$

por lo que R^{-1} vendría siendo una especie de "raíz cuadrada" de P. Es decir, $P^{1/2} = R^{-1}$, $P^{1/2} P^{*/2} =$

P. De esta forma, la pseudoinversa de la matriz de canal extendida queda expresada como

$$H_\alpha^\dagger = P^{1/2} \begin{bmatrix} Q_\alpha \\ Q_2 \end{bmatrix}^* = P^{1/2} Q_\alpha^* \tag{3.24}$$

Por otro lado, es sencillo demostrar que el elemento de la diagonal de P de menor valor se corresponde con la fila de menor norma euclidiana de $P^{1/2}$. Si se intercambian en $P^{1/2}$ la fila de menor longitud con la última fila, se puede encontrar una transformación unitaria Σ (puede ser mediante rotaciones de Givens o reflexiones de Householder) que anule los elementos de la última fila menos el elemento de la diagonal principal.

$$P^{1/2}\Sigma = \begin{bmatrix} A_{(m-1)\times(m-1)} & y_{(m-1)\times 1} \\ 0_{1\times(m-1)} & p_m^{1/2} \end{bmatrix} \tag{3.25}$$

donde $p_m^{1/2}$ es un escalar. Como la transformación es unitaria, la matriz $P^{1/2}\Sigma$ sigue siendo una raíz cuadrada de P.

Queda entonces investigar cómo actualizar la raíz cuadrada $P^{1/2}$ para cada iteración, es decir, como obtener $P^{(k-1)/2}$ luego de eliminar la última fila de $P^{(k)/2}$.

Desarrollando $\alpha I + HH^*$ se tiene:

$$\begin{aligned}
\alpha I + HH^* &= \left(P^{1/2}P^{*/2}\right)^{-1} = \left(P^{1/2}\Sigma\Sigma^* P^{*/2}\right)^{-1} = \left(\Sigma^* P^{*/2}\right)^{-1}\left(P^{1/2}\Sigma\right)^{-1} \\
&= \left(\begin{bmatrix} A^*_{(m-1)\times(m-1)} & 0_{(m-1)\times 1} \\ y^*_{1\times(m-1)} & p_m^{*/2} \end{bmatrix}\right)^{-1} \left(\begin{bmatrix} A_{(m-1)\times(m-1)} & y_{(m-1)\times 1} \\ 0_{1\times(m-1)} & p_m^{1/2} \end{bmatrix}\right)^{-1} \\
&= \begin{bmatrix} \left(A^*_{(m-1)\times(m-1)}\right)^{-1} & 0_{(m-1)\times 1} \\ \times & \times \end{bmatrix} \begin{bmatrix} \left(A_{(m-1)\times(m-1)}\right)^{-1} & \times \\ 0_{1\times(m-1)} & \times \end{bmatrix} \\
&= \begin{bmatrix} \left(A_{(m-1)\times(m-1)} A^*_{(m-1)\times(m-1)}\right)^{-1} & \times \\ \times & \times \end{bmatrix}
\end{aligned} \tag{3.26}$$

Efectuando otro desarrollo:

$$\alpha I + HH^* = \alpha I + \begin{bmatrix} H^{(m-1)*} \\ \mathrm{h}_m^* \end{bmatrix} \begin{bmatrix} H^{(m-1)} & \mathrm{h}_m \end{bmatrix} = \begin{bmatrix} \alpha I + H^{(m-1)} H^{(m-1)*} & \times \\ \times & \times \end{bmatrix}$$

Comparando este último resultado con el obtenido en (3.26) se obtiene

$$\left(A_{(m-1)\times(m-1)} A^*_{(m-1)\times(m-1)} \right)^{-1} = \alpha I + H^{(m-1)} H^{(m-1)*} = \left(P^{(m-1)} \right)^{-1}$$

y por tanto $A_{(m-1)\times(m-1)}$ es una raíz cuadrada de $P^{(m-1)}$, es decir, $P^{(m-1)/2} = A_{(m-1)\times(m-1)}$.

De (3.21), (3.23) y (3.25), en la primera iteración el vector anulador es

$$H^{\dagger}_{\alpha,m} = \begin{bmatrix} 0_{1\times(m-1)} & p_m^{1/2} \end{bmatrix} Q^*_\alpha = p_m^{1/2} \mathrm{q}^*_{\alpha,m}$$

donde $\mathrm{q}_{\alpha,m}$ es la m-ésima columna de Q_α. En sentido general, $H^{\dagger}_{\alpha,k} = p_k^{1/2} \mathrm{q}^*_{\alpha,k}$

En resumen, toda la información que se necesita para determinar el orden de detección y los vectores anuladores se puede encontrar a partir de $P^{1/2}$ y Q_α. Con el recurso de la factorización QR y el esquema de actualización de la matriz $P^{1/2}$ para cada iteración, el algoritmo 4 quedaría expresado como se muestra en el algoritmo 5

El algoritmo *Square-Root* de [61] también incluye una forma de calcular $P^{1/2}$ y Q_α de manera recursiva usando el filtro de Kalman, de modo que se ahorra realizar la descomposición QR y la inversión de la matriz R. Con este último recurso la complejidad del método se reduce a $O\left(m^3\right)$ y ya es comparable computacionalmente con el método *Zero-Forcing*.

Desde el punto de vista de la detección, sin dudas el criterio MMSE es mejor que el criterio ZF, y en ambos casos la mejora es considerable si se usa el esquema OSIC para estimar cada símbolo transmitido. Una comparación respecto a la tasa de error en la detección (BER - *Bit Error Rate*) se puede encontrar en [137].

Algoritmo 5 MMSE-OSIC-QR(H, α, x)

Entrada: $H \in \mathbb{R}^{n\times m}$ la matriz de canal del sistema
α el valor recíproco del SNR
$x \in \mathbb{R}^n$ el vector recibido.

Salida: La estimación de la señal transmitida $\hat{s} \in \mathcal{A}^m$

1: $H_\alpha^{(m)} \leftarrow \begin{bmatrix} H \\ \sqrt{\alpha} I_m \end{bmatrix}$
2: $[Q_\alpha^{(m)}, R] \leftarrow$ Factorización QR de $H_\alpha^{(m)}$
3: $P^{(m)/2} \leftarrow R^{-1}$
4: **para** $k = m, m-1, \ldots, 1$ **hacer**
5: **si** j es la fila de menor longitud en $P^{(k)/2}$ **entonces**
6: Intercambiar las filas j y k de $P^{(k)/2}$, de s y de $H_\alpha^{(k)\dagger}$
7: Intercambiar las columnas j y k de H
8: **fin si**
9: Calcular una transformación unitaria Σ que anule los elementos de la última fila de $P^{(k)/2}$ excepto el elemento en (k,k)

$$P^{(k)/2} \sum = \begin{bmatrix} P^{(k-1)/2} & y_{(m-1)\times 1} \\ 0_{1\times(m-1)} & p_m^{1/2} \end{bmatrix}$$

10: $Q_\alpha^{(k)} \leftarrow Q_\alpha^{(k)} \Sigma$ {Actualización de la matriz Q_α}
11: $\hat{s}_k \leftarrow \left\lfloor p_k^{1/2} q_{\alpha,k}^* x \right\rceil_{\mathcal{A}}$
12: $x^{(k-1)} \leftarrow x^{(k)} - \underline{h}_k \hat{s}_k$ {$\underline{h}_k$ *es la fila k-ésima de* H}
13: $Q_\alpha^{(k-1)} \leftarrow$ primeras $k-1$ columnas de $Q_\alpha^{(k-1)}$
14: **fin para**

3.4 Métodos Maximum-Likelihood

Existen técnicas más precisas que los métodos descritos en la sección 3.3, aunque más costosas. Como se puede ver en [68], respecto a la tasa de error en la detección (BER), la solución ML claramente supera la solución que ofrecen los métodos heurísticos, incluso la del MMSE-OSIC. Esta sección está dedicada al estudio de estas técnicas ML, que en sentido general se agrupan en una denominación llamada *Sphere-Decoding*. Se verá en esta sección que cada una de estas técnicas sigue un esquemas de Ramificación y Poda en cualquier de sus variantes. Es por ello que primeramente se hace una breve introducción a este esquema de solución de problemas, antes de hacer un estudio del estado del arte de las distintas vertientes por donde los métodos *Sphere-Decoding* han sido mejorados.

3.4.1 Esquemas de Ramificación y Poda

El método de diseño de algoritmos Ramificación y Poda (cuyo nombre proviene del término *Branch and Bound* [79]) es ampliamente usado para resolver problemas de optimización en espacios discretos. En estos métodos la búsqueda se realiza sobre un árbol que se va construyendo a medida que se van generando alternativas de posibles soluciones a partir de otras intermedias. Cada nodo del árbol representa precisamente una solución parcial o final del problema. Se necesita, por tanto, una estructura lineal para almacenar los nodos generados y para poder determinar en cada momento cuál analizar.

El esquema general de Ramificación y Poda consiste en tres etapas. La primera de ellas, denominada *Selección*, se encarga de extraer un nodo de la estructura. En la segunda etapa, llamada *Ramificación*, se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior. Por último se realiza la tercera etapa, la *Poda*, que es descartar aquellos nodos generados en la etapa anterior que no cumplan cierta condición, adicionando el resto de los nodos *aceptables* a la estructura. Estas tres etapas se repiten hasta que se encuentre la(s) solución(es) deseada(s), o bien hasta que no queden más nodos que ramificar en la estructura. Al inicio del algoritmo, la estructura debe ser inicializada con al menos un nodo.

Del tipo de estructura utilizada depende el tipo de recorrido del árbol que se realiza, y en consecuencia el orden de selección de los nodos. Si se quiere explorar el árbol siguiendo un recorrido en

profundidad (*Depth-First*), se utilizaría entonces una pila (estructura LIFO, *Last In - First Out*), mientras que si se quiere seguir un recorrido a lo ancho (*Breadth-First*) se utilizaría una cola (estructura FIFO, *First In - First Out*). También se podría utilizar para almacenar los nodos una estructura de montículo (*heap*), conocida también como cola de prioridades, pero en este caso los nodos requieren una función de coste h para decidir en cada momento qué posición ocuparía en el *heap*.[2]

Lo que le da valor a esta técnica es la posibilidad de disponer de distintas estrategias de exploración del árbol y de acotar la búsqueda de la solución, que en definitiva se traduce en eficiencia. También existe la posibilidad de ejecutar estos métodos en paralelo [56, 55], pues pueden existir varios procesos que realicen de forma independiente extracciones, expansiones y ramificaciones de nodos, pues ningún nodo necesita de otro para ejecutar alguna de estas operaciones.

Tienen como desventaja que son costosos en cuanto a memoria, pues cada nodo debe ser autónomo, en el sentido que ha de contener toda la información para el proceso de ramificación y poda, lo que imposibilita que se disponga de una estructura global para construir la solución.

Un primer esquema de Ramificación y Poda (ver Algoritmo 6) es el que recorre el árbol de soluciones hasta encontrar una primera solución. El funcionamiento del esquema es sencillo: Se construye un nodo inicial a partir del cual se comenzará la búsqueda de la solución del problema. Dicho nodo inicial se almacena en la `Estructura` y a partir de ahí se suceden operaciones de extracción desde la `Estructura`, y ramificaciones del nodo extraído (que genera nuevos nodos supuestamente más cercanos a la solución). Estas operaciones se realizan hasta que el nodo extraído de la estructura sea en efecto una solución del problema. En caso de que se vacíe la `Estructura` sin antes encontrar una solución, entonces se retorna una solución nula.

El Tipo de Dato Abstracto `Nodo` ha de contener toda la información relativa a una solución parcial o final del problema que se pretende resolver. Sobre este tipo de datos están definidas las siguientes operaciones:

- `Expandir`: Construye y devuelve los nodos hijos de un nodo dado. Realiza el proceso de ramificación del algoritmo.

[2] Para más información sobre las distintas estructuras de datos ver [4]

Algoritmo 6 Algoritmo General de Ramificación y Poda para encontrar la primera solución

Variables:

```
    E: Estructura;
    N: Nodo;
    hijos : ARREGLO [1..MAXHIJOS] DE Nodo;
    nHijos : ENTERO;
 1: Adicionar(E, NodoInicial()) {Adiciona un nodo inicial a la Estructura E}
 2: mientras EstaVacia(E) hacer
 3:   N ← Extraer(E) {Se extrae de E un nodo y se asigna a N}
 4:   si EsSolucion(N) entonces
 5:     retornar N {El nodo N es la primera solucion encontrada}
 6:   fin si
 7:   [hijos, nHijos] ← Expandir(N);
 8:   para i = 1, ..., nHijos hacer
 9:     si EsAceptable(hijos[i]) entonces
10:       Adicionar(E, N) {Adiciona N a la Estructura de Datos Lineal E}
11:     fin si
12:   fin para
13: fin mientras
14: retornar NoHaySolucion()
```

- `EsAceptable`: Es la que realiza la poda en el algoritmo, pues dado un nodo decide si seguir analizándolo en la próxima iteración o rechazarlo.

- `EsSolucion`: Decide si un nodo es una hoja del árbol de solución, o lo que es lo mismo, es una solución final del problema. Dicha solución no ha de ser necesariamente la mejor

- `Valor`: Devuelve el valor asociado al nodo.

- `NodoInicial`: Devuelve el nodo que constituye la raíz del árbol de expansión para el problema.

La función `NoHaySolucion` devuelve un nodo con un valor especial que indica que al problema no se le encontró solución.

A su vez, el Tipo de Dato Abstracto `Estructura` presenta las siguientes operaciones elementales:

- `Adicionar`: Adiciona un nodo en la estructura.

- `Extraer`: Extrae un nodo de la estructura.

- `EstaVacia`: Devuelve verdadero si la estructura no contiene ningún nodo almacenado, y falso en caso contrario.

Un segundo esquema de Ramificación y Poda (ver Algoritmo 7) es el que recorre el árbol de soluciones hasta encontrar una mejor solución. Al igual que en el algoritmo anterior, se almacena un nodo inicial en la estructura y de ahí se realizan operaciones de extracción desde la `Estructura`, y ramificaciones del nodo extraído. Aquí es necesario tener dos variables globales (`solucion` y `valorSolucion`) que serían actualizadas cada vez que se encuentra una nueva solución. Cuando se vacía la `Estructura` termina el método y se retorna la mejor solución encontrada.

Algoritmo 7 Algoritmo General de Ramificación y Poda para encontrar la mejor solución

```
Variables:
    E: Estructura;
    N, solucion : Nodo;
    hijos : ARREGLO [1..MAXHIJOS] DE Nodo;
    nHijos : ENTERO;
    valorSolucion : REAL;
 1: solucion ← NoHaySolucion(); valorSolucion ← MAX
 2: Adicionar(E, NodoInicial()) {Adiciona un nodo inicial a la Estructura E}
 3: mientras EstaVacia(S) hacer
 4:   N ← Extraer(E) {Se extrae de E un nodo y se asigna a N}
 5:   [hijos, nHijos] ← Expandir(N);
 6:   para i = 1, ..., nHijos hacer
 7:     si EsAceptable(hijos[i]) entonces
 8:       si EsSolucion(hijos[i]) entonces
 9:         si Valor(hijos[i]) < valorSolucion entonces
10:           valorSolucion ← Valor(hijos[i])
11:           solucion ← hijos[i]
12:         fin si
13:       sino
14:         Adicionar(E, N) {Adiciona N a la Estructura de Datos Lineal E}
15:       fin si
16:     fin si
17:   fin para
18: fin mientras
19: retornar solucion
```

En lo adelante, si se quiere resolver un problema de optimización, sólo es necesario especificar cada una de las operaciones definidas para cada tipo de datos, además de especificar la información que debe almacenar el Tipo de Datos Abstracto `Nodo`.

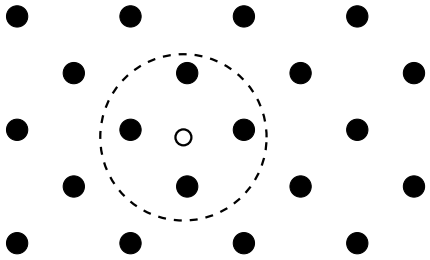


Figura 3.1: Idea del *Sphere-Decoding*

3.4.2 Método Sphere-Decoding

Los métodos de decodificación esférica (SD - *Sphere-Decoding*) son métodos que siguen un esquema de Ramificación y Poda, y alcanzan la solución ML al problema (3.8). La idea básica del algoritmo *Sphere-Decoding* es intentar buscar sólo aquellos puntos de la retícula que están dentro de la esfera con centro en el vector dado x, y un radio r inicialmente escogido (ver Figura 3.1). Claramente, el punto más cercano a x dentro de la esfera, es el punto más cercano a x en toda la retícula. Reducido el espacio de búsqueda se reduce por tanto la complejidad computacional.

La idea entonces se resume en encontrar todos los vectores $s \in {\mathcal{A}_L}^m$ tales que

$$r^2 \geq \|x - Hs\|^2 \tag{3.27}$$

y luego se selecciona el que minimiza la función objetivo.

Determinar los puntos de una retícula que se encuentren en el interior de una hiper-esfera m-dimensional es una tarea difícil. Sin embargo, los métodos SD proponen una idea eficiente para resolver el problema. Parten del caso base $m = 1$, donde es trivial pues la esfera es unidimensional, y determinar los puntos en su interior es determinar los puntos en el interior de un intervalo. Luego se usa esta observación para ir de la dimensión k a la $k+1$. Suponiendo que se han determinado todos los puntos k-dimensionales que están en una esfera de radio r, para cada uno de esos puntos el conjunto de valores admisibles en la coordenada $(k+1)$-dimensional que se encuentran en la esfera de mayor dimensión y el mismo radio r forman un intervalo. Lo anterior significa que se puede determinar todos los puntos solución en una esfera de dimensión m y radio r determinando sucesivamente todos los puntos solución

en esferas de dimensiones menores $1, 2, \ldots, m$ y el mismo radio r.

Para subdividir el problema en estos diversos subproblemas, se realiza previamente la descomposición QR de la matriz H, donde R es una matriz triangular de dimensiones $m \times m$ y elementos de la diagonal todos positivos, y $Q = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}$ es una matriz ortogonal $n \times n$. Las matrices Q_1 y Q_2 representan las primeras m y las últimas $n-m$ columnas ortogonales de Q. La condición (3.27) puede por tanto escribirse como

$$r^2 \geq \left\| x - \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R \\ 0 \end{bmatrix} s \right\|^2 = \left\| Q_1^T x - Rs \right\|^2 + \left\| Q_2^T x \right\|^2$$

o lo que es lo mismo

$$r^2 - \left\| Q_2^T x \right\|^2 \geq \left\| Q_1^T x - Rs \right\|^2 \tag{3.28}$$

Definiendo $y = Q_1^T x$ y $r'^2 = r^2 - \left\| Q_2^T x \right\|^2$ se reescribe (3.28) como

$$r'^2 \geq \| y - Rs \|^2 \tag{3.29}$$

Luego, teniendo en cuenta que R es triangular superior, la desigualdad (3.29) puede ser nuevamente escrita como

$$r'^2 \geq (y_m - R_{m,m} s_m)^2 + \| y_{1:m-1} - R_{1:m-1,1:m} s_{1:m} \|^2 \tag{3.30}$$

De esta última condición, se desprende que una condición necesaria para que Hs esté dentro de la esfera, es que $r'^2 \geq (y_m - R_{m,m} s_m)^2$, o lo que es lo mismo, que la componente s_m, pertenezca al intervalo

$$\left\lceil \frac{-r' + y_m}{R_{m,m}} \right\rceil_{\mathcal{A}} \leq s_m \leq \left\lfloor \frac{r' + y_m}{R_{m,m}} \right\rfloor_{\mathcal{A}} \tag{3.31}$$

donde $\lceil \xi \rceil_{\mathcal{A}}$ denota el menor elemento de la constelación $\mathcal{A}$ mayor o igual que ξ, y $\lfloor \xi \rfloor_{\mathcal{A}}$ denota el mayor elemento menor o igual que ξ. Luego, para cada valor de s_m dentro del intervalo (3.31), se determina el intervalo donde estarán los valores de s_{m-1} mediante

$$\left\lceil \frac{-r'_{m-1} + y_{m-1|m}}{R_{m-1,m-1}} \right\rceil_{\mathcal{A}} \leq s_{m-1} \leq \left\lfloor \frac{r'_{m-1} + y_{m-1|m}}{R_{m-1,m-1}} \right\rfloor_{\mathcal{A}} \tag{3.32}$$

donde $r'^2_{m-1} = r'^2 - (y_m - R_{m,m}s_m)^2$ y $y_{m-1|m} = y_{m-1} - R_{m-1,m}s_m$. El algoritmo continúa de esa misma forma para determinar s_{m-2} y así sucesivamente hasta determinar s_1. En caso de no encontrarse ninguna solución, el radio r debe aumentarse y se ejecutaría nuevamente el algoritmo.

La búsqueda en el SD pertenece a la clase de métodos Ramificación y Poda, específicamente al esquema general para encontrar la mejor solución, descrito en el algoritmo 7. En el método SD los nodos del nivel k son los puntos de la retícula que están dentro de la esfera de radio r'_{m-k+1} y dimensión $m-k+1$. Las hojas del árbol serían las soluciones de (3.27). Para adecuar el método *Sphere-Decoding* a un esquema *Branch and Bound* sólo es necesario que cada nodo contenga la siguiente información:

- Nivel del árbol al cual pertenece: $m - k + 1$
- Valor de $y_{k|k+1}$
- Valor de r'_k
- Componentes del vector $\hat{s}$ hasta ese momento determinadas: $\hat{s}_m, \ldots, \hat{s}_k$

donde $k = m, m-1, \ldots, 1$. La ramificación de un nodo de nivel $m - k + 1$ (dada en el algoritmo 7 por la rutina `Expandir`) generaría tantos nodos como elementos tiene el alfabeto o constelación, y la poda (condicionada en el algoritmo 7 mediante la rutina `EsAceptable`) se realizaría aceptando sólo aquellos nodos cuya componente $\hat{s}_{k-1}$ se encuentre en el intervalo

$$\left[\left\lceil \frac{-r'_{k-1} + y_{k-1|k}}{R_{k-1,k-1}} \right\rceil_{\mathcal{A}}, \left\lfloor \frac{r'_{k-1} + y_{k-1|k}}{R_{k-1,k-1}} \right\rfloor_{\mathcal{A}}\right] \tag{3.33}$$

Todo nodo cuyo nivel sea m es un nodo solución. Precisamente para los nodos que representan una posible solución (un punto dentro de la hiper-esfera) estaría definida la rutina `Valor` la cual retornaría el valor de $\|x - H\hat{s}\|^2$.

Dado que el recorrido que se sigue en el SD es un recorrido en profundidad, buscando la mejor de todas las soluciones posibles, la estructura que se usaría para almacenar los nodos sería una estructura LIFO (*Last-In First-Out*) o pila. La estructura se inicializaría con un nodo especial N_0 cuyo nivel en el árbol sería 0, y el valor de $y_{m+1|m+2}$ sería la última componente del vector y, el valor de r_{m+1} podría ser el radio inicial y el vector $\hat{s}$ no tendría componentes definidas.

La complejidad computacional de los métodos SD se analiza amplia y detalladamente por Hassibi y Vikalo en [62] y [131]. En tales publicaciones se demuestra que el método SD, para grandes dimensiones y amplios rangos de SNR, tiene una complejidad computacional polinomial, cercana a la complejidad cúbica. Lo analizado en estos trabajos tiene en cuenta el funcionamiento del SD propuesto por Fincke-Pohst [41], pero en la última década ha emergido un número considerable de investigaciones sobre los métodos SD, en las cuales se trata de optimizar algunas cuestiones que influyen en el rendimiento de estos métodos, como es la selección del radio para la búsqueda y el orden de decodificación de los símbolos.

En las siguientes sub-secciones se abordan algunos de los más significativos de estos avances.

3.4.3 Preprocesado y ordenación de la decodificación

La complejidad de los decodificadores en esfera depende críticamente de la etapa de preprocesado y el orden en el cual se considera a las componentes de la señal transmitida s. El preprocesado y ordenación estándar consiste en el cálculo de la factorización QR sobre la matriz del canal H y la ordenación de los componentes de la forma natural pero empezando por el último, dada por $s_m, s_{m-1}, \ldots, s_1$.

Sin embargo, con la aplicación de algoritmos alternativos de preprocesado/ordenación se puede conseguir una complejidad esperada menor en algunos casos. El preprocesado y consiste en encontrar una matriz de permutación P, cuyas componentes todas son enteras al igual que las de P^{-1}, de modo que permita modificar el sistema de la siguiente forma:

$$x = Hs + v = HPP^{-1}s + v = \tilde{H}z + v \tag{3.34}$$

Una primera propuesta de preprocesado y ordenación plantea la ordenación de las columnas de la matriz H según las longitudes de sus vectores columnas. Es decir, se trata de encontrar una matriz de permutación de columnas P a la matriz H tal que la matriz $\tilde{H} = HP$ cumpla que

$$\left\|\tilde{\mathrm{h}}_1\right\| \leq \left\|\tilde{\mathrm{h}}_2\right\| \leq \ldots \leq \left\|\tilde{\mathrm{h}}_m\right\| \tag{3.35}$$

Esta ordenación puede hacerse efectuando una ordenación ascendente de las columnas, o reduciendo la

matriz H mediante el método LLL descrito en la sección 3.2.3. Las componentes de x serían permutadas acorde a una permutación p correspondiente a la permutación de columnas P, es decir, las componentes de x serían procesadas en el orden $x_{p(m)}, x_{p(m-1)}, \ldots, x_{p(1)}$. Las cotas superiores e inferiores que se van calculando recursivamente en el algoritmo SD según (3.33) implican que la expansión de $x_{p(i)}$ depende de la expansión de $x_{p(i+1)}, \ldots, x_{p(m)}$ de la misma forma que la decisión de $x_{p(i)}$ depende de la decisión de $x_{p(1)}, \ldots, x_{p(i-1)}$. Eligiendo una permutación p que se corresponda con una permutación P que ordene de manera creciente las columnas de H, la expansión de $x_{p(i)}$ se verá reducida con alta probabilidad, de modo que la complejidad esperada del método también se reducirá.

Si se quiere resolver un problema CVP derivado de un problema de decodificación en un sistema MIMO, donde las señales se forman a partir de una constelación $\mathcal{A}$, la reducción LLL no es recomendable para realizar esta ordenación de las columnas. Este método tiene el inconveniente de que no sólo cambia el orden de las columnas del canal H y de los símbolos del vector de señal recibido, sino que también cambia la constelación durante la transmisión. El cambio de constelación produce un aumento del número de símbolos reales que la representan de forma equivalente y por tanto un aumento del tamaño del árbol de decodificación. Además la nueva constelación es variable según la matriz de permutación. Todo ello significa que no compensa el coste invertido en realizar este preprocesado y deshacerlo, frente a la mejora conseguida en el tiempo necesario para detectar.

Otra propuesta de preprocesado y ordenamiento es la conocida como V-BLAST ZF-DFE [46], que consiste en encontrar una permutación P que maximice el mínimo de los elementos $R_{1,1}, R_{2,2}, \ldots, R_{m,m}$ de la diagonal de la matriz R, de la descomposición QR de la matriz $\tilde{H} = HP$. Nótese por (3.33) existe una relación inversa entre las cotas de los intervalos que se generan en el SD y los valores de los elementos de la diagonal de R. Esto significa que si se intenta maximizar los elementos $R_{1,1}, R_{2,2}, \ldots, R_{m,m}$, se reducen los intervalos para cada componente $\hat{s}_k$ y con ello se reduce el árbol de búsqueda.

Estas dos propuestas de preprocesado y ordenamiento se basan sólo en la matriz de canal H. Su y Wassell en [113] muestran que un ordenamiento óptimo no depende sólo de la matriz H sino también del vector recibido x. Los resultados experimentales logrados en este trabajo muestran que el esquema de preprocesado y ordenamiento que se propone reduce considerablemente la cantidad de nodos expandidos por el SD, logrando menores índices que la técnica V-BLAST ZF-DFE. Sin

embargo, dado que la propuesta de ordenamiento depende de la señal recibida, dicho ordenamiento debe calcularse por cada señal que se recibe en el sistema MIMO. Como por cada matriz de canal se envían y decodifican varias señales, es deseable que la descomposición QR de la matriz de canales que se usa en el método SD se realice una sola vez (en la detección de la primera señal), y pueda reutilizarse en el resto de las detecciones. El incoveniente que presenta la propuesta de Karen y Wassell es que la QR de la matriz de canales debe calcularse por cada señal a decodificar.

3.4.4 Selección de los radios iniciales

El costo computacional del SD depende en gran medida también de la elección del radio inicial r en (3.27). Si r es muy grande, la cantidad de nodos en cada nivel del árbol sería considerable y la complejidad de la búsqueda se mantendría exponencial, mientras que si por otro lado, r es muy pequeño, puede suceder que no hayan puntos dentro de la esfera.

Un buen candidato para estimar r es el denominado radio de cobertura de la retícula, definido como el menor radio de esferas centradas en los puntos de la retícula que cubra todo el espacio m-dimensional. Este es claramente el menor radio que garantiza la existencia de un punto dentro de la esfera con centro en un vector cualquiera x. El problema con esta elección de r es que determinar el radio de cobertura de una retícula dada ya es en sí NP-*complejo* [90].

Otra de las elecciones para el radio es la distancia entre el vector x y el punto de Babai (definido en la sección 3.3.1), pues este radio garantiza la existencia de al menos un punto (la solución del ZF). Sin embargo, en varios casos esta elección puede producir muchos puntos dentro de la esfera. En [86] proponen que el radio inicial se obtenga mediante $r = \|x - Hs_{\text{MMSE}}\|$, donde s_{MMSE} es la solución obtenida por el método MMSE (ver sección 3.3.3). Un inconveniente que puede presentar tomar el radio inicial como la distancia $\|x - H\hat{s}\|$, donde $\hat{s}$ es la solución de algún método heurístico (como ZF o MMSE), es que el proceso de aproximar un valor continuo a un elemento de la constelación para transformar la solución numérica en un vector que pertenezca a la constelación $\mathcal{A}^m$, puede provocar que la distancia entre x y la solución obtenida sea mayor y por tanto la esfera contendría dentro muchos puntos. Como ventaja tienen que son muy seguros, pues garantizan al menos un punto de la retícula dentro de la hiper-esfera.

Si se conoce que el problema de mínimos cuadrados es formulado para decodificar una señal recibida en un sistema MIMO, se puede aprovechar la varianza σ^2 del ruido para estimar el radio inicial. En [62] se propone que el radio inicial se obtenga mediante

$$r^2 = \alpha n \sigma^2 \tag{3.36}$$

donde el valor de α es tomado de modo tal que la probabilidad de encontrar un punto dentro de la hiper-esfera

$$\int_0^{\alpha n/2} \frac{\lambda^{n/2-1}}{\Gamma(n/2)} e^{-\lambda} d\lambda \tag{3.37}$$

sea bastante cercana a 1.

En la variante del SD donde se usa la estrategia de numeración Schnorr-Euchner no es imprescindible definir un radio inicial de búsqueda. Esta variante, siempre que encuentra un punto dentro de la hiper-esfera, actualiza el radio de búsqueda con la distancia $\|x - \hat{x}\|$, donde $\hat{x}$ es el punto encontrado dentro de la hiper-esfera. Es por ello que se puede inicializar $r = \infty$. Sin embargo, como bien se demuestra en [31], si se inicializa $r = \infty$ el punto que primeramente encuentra el método es el punto solución del ZF, provocando que el radio se actualice con la distancia $\|x - Hs_{\mathrm{ZF}}\|$.

En [138] se propone una estrategia para reducir la búsqueda cuando se incrementa el radio luego de no tener éxito en un intento previo, mientras que en [13] se presenta una variante de aceleración de la reducción del radio durante la búsqueda.

A pesar de estos resultados en la gestión de los radios de búsqueda en los métodos SD, se considera que aún la selección del radio inicial es un problema que merece seguirse investigando.

3.4.5 Sphere-Decoding Automático

En la sección 3.4.2 se vio que el algoritmo *Sphere-Decoding* realiza un recorrido en profundidad de izquierda a derecha en el árbol buscando el nodo que represente la mejor solución al problema. Como se vio en la sección 3.4.4, una de las desventajas del esquema SD es que su complejidad depende en gran medida de la elección del radio inicial r.

El algoritmo *Automatic Sphere-Decoding* (ASD) es otro tipo de búsqueda de la mejor solución sin

usar un radio inicial. Este algoritmo garantiza que el primer nodo visitado del último nivel, sea el nodo que contiene la solución óptima.

Este algoritmo se basa en asociarle a cada nodo un peso determinado dentro de la búsqueda. El peso de un nodo cuyo nivel es $m - k + 1$ se calcula mediante

$$\sqrt{\sum_{i=k}^{m} d_i^2} \tag{3.38}$$

donde el vector $d = (d_1, d_2, \ldots, d_m)^t$ viene dado por

$$d = y - R\hat{s} \tag{3.39}$$

siendo la matriz R obtenida de la descomposición QR de la matriz de canales H, el vector $y = Q_1^T x$, y el vector $\hat{s}$ es el correspondiente al nodo.

Claramente, el peso σ_* asociado al nodo de nivel m que contiene el vector solución $\hat{s}$, es precisamente la distancia que existe entre el vector x y el vector más cercano a él en la retícula con base H. Por otro lado, según (3.38) es evidente que el peso de un nodo será menor que el peso de sus correspondientes nodos hijos. Es por ello que durante la búsqueda por el árbol, no se deberían seleccionar nodos cuyos pesos sean mayores que σ_* pues no llegarían a la solución del problema.

Los métodos SD, tanto el de Fincke y Pohst como el de Schnorr y Euchner, pueden seleccionar durante la búsqueda nodos cuyos pesos son menores que el peso de la solución, pues basan su decisión en un radio de búsqueda. El ASD está diseñado para encontrar la solución sin expandir nodos cuyos pesos son mayores que σ_* y sin tener un conocimiento previo de la solución del problema.

El método ASD usa una lista de nodos cuyo primer elemento siempre será el de menor peso. La lista puede ser inicializada con el mismo nodo inicial definido en la sección 3.4.2 que se usaría en el SD. Luego, el procedimiento es similar: selecciona el primer elemento de la estructura, genera sus nodos hijos y los incluye en la lista. Cuando se adicionan los nodos a la lista, es necesario determinar el nodo de menor peso para que este ocupe la primera posición. Como el ASD no usa radio de búsqueda, entonces la expansión de un nodo debe generar siempre tantos nodos como cantidad de símbolos tiene

la constelación $\mathcal{A}$. La ejecución del método terminaría cuando haya extraído de la lista un nodo que pertenezca al último nivel del árbol, o sea, un nodo que sea hoja del árbol. Cuando el algoritmo extrae de la lista un nodo hoja, puede dar por terminada la búsqueda pues en ese momento el resto de los nodos de la lista tendrán pesos mayores (dada la condición de que la lista siempre mantendrá en la primera posición el de menor peso), y transitivamente aquellos que no sean hojas entonces no podrían generar nodos con pesos menores al nodo encontrado. Es por ello que el primer nodo hoja extraído de la lista es el nodo que contiene la solución del problema. El método se puede ver expresado formalmente en [112].

La búsqueda en el ASD también pertenece a la clase de métodos Ramificación y Poda, con la particularidad de que no realiza poda en la ramificación de un nodo, específicamente al esquema general para encontrar la primera solución presentado en el algoritmo 6. La estructura que se usaría sería una cola de prioridades o *heap* que es la que garantizaría que en la primera posición de la estructura siempre se encuentre el nodo con menor valor en la función de coste `Valor`. La función de coste evidentemente sería el peso definido por (3.38). Dado que en este caso no se realiza poda la rutina `EsAceptable` siempre retornaría `true`. La estructura del nodo sería la misma usada en la implementación del método SD, sólo que el campo para almacenar el valor del radio carecería de uso.

4

Métodos paralelos de optimización basados en búsqueda directa

En este capítulo se abordan los métodos de optimización basados en Búsqueda Directa. En una primera sección se da un repaso al estado del arte en el campo de la optimización mediante los métodos de Búsqueda Directa, donde se define lo que es la búsqueda directa, se describen las distintas clases de métodos, se describe el framework GSS (*Generating Set Search*) y se numeran los últimos avances de estos métodos en el campo de la computación paralela. En una segunda sección se describen los distintos algoritmos de búsqueda directa diseñados e implementados en el trabajo. En una tercera sección se describe la paralelización de los algoritmos secuenciales de mejores resultados. Finalmente en las siguientes secciones se exponen los resultados de estos métodos en dos problemas de optimización. El primer problema es el Problema Inverso Aditivo de Valores Singulares (PIAVS) el cual es un problema continuo, y el segundo problema es el Problema del Vector más Cercano en Retículas (cuyas siglas son CVP por su nombre en inglés *Closest Vector Point*)

4.1 Estado del arte

4.1.1 ¿Qué son los métodos de búsqueda directa?

En primer capítulo los métodos de búsqueda directa se introdujeron como métodos de optimización que no utilizan derivadas explícitamente. Sin embargo, como se argumenta en [123], esta característica no ofrece una definición completa de lo que es la Búsqueda Directa. El término "búsqueda directa" en el campo de la optimización fue acuñado en 1961 por Hooke y Jeeves en [64]. Hooke y Jeeves pretendieron exponer que los métodos de búsqueda directa son aplicables en situaciones en las cuales no se puede hacer más que comparar valores de la función objetivo. Además de eso, pretendieron distinguir los métodos de búsqueda directa de los métodos tipo Newton y los métodos de descenso suficiente. Más tarde, Torczon en [120] sentenció:

> "Los métodos de búsqueda directa se caracterizan por el hecho de que el proceso de toma de decisiones se basa sólo en la información que ofrece la evaluación de la función; estos algoritmos ni requieren ni estiman información de las derivadas para determinar una dirección de descenso."

También Wright en [136] ofrece una definición de búsqueda directa en la cual deja claro que estos métodos no intentan usar o aproximar derivadas. Trosset en [124] da una definición de la búsqueda directa más elaborada donde se acomodan variantes estocásticas de la búsqueda directa. En [83] se argumenta que como los métodos de Newton se clasifican como métodos de "segundo orden" por disponer de la primera y la segunda derivadas de la función, además de usar el polinomio de Taylor de segundo orden para construir aproximaciones de la función; y que como los métodos de máximo descenso se clasifican de "primer orden" por disponer de la primera derivada y usar polinomios de Taylor de primer orden para aproximar la función, entonces los métodos de búsqueda directa se clasifican como métodos de "orden cero", pues no disponen de ninguna derivada y no construyen aproximaciones de la función.

Sin embargo, la definición precisa de la búsqueda directa no es un tema cerrado, como se concluye en [74].

4.1.2 Tipos de métodos de búsqueda directa

La época dorada de los métodos de búsqueda directa fue el período 1960-1971. Los algoritmos desarrollados en ese período fueron ampliamente usados, y sentaron las bases de las variantes desarrolladas posteriormente. Luego Swann en [115] resumió el estado de la búsqueda directa y concluyó que estos métodos a pesar de ser desarrollados a base de heurísticas y de no existir demostraciones teóricas sobre su convergencia, en la práctica han demostrado ser robustos aunque a veces muy lentos. Sin embargo, el hecho de no existir una teoría que los sustenten hizo que la comunidad científica relegara la búsqueda directa a un segundo plano por varios años. Es curioso que, como se verá más adelante, a partir de 1971 comenzaron a aparecer los primeros resultados referentes a la convergencia de la búsqueda directa.

En esta sección se analiza en síntesis el desarrollo y evolución de la búsqueda directa. Para una mejor organización, fueron divididos en tres categorías:

- Métodos simplex.
- Métodos de búsqueda multidireccional.
- Métodos de búsqueda por patrones.

En cada sección dedicada a cada una de las categorías, se explica en qué consisten los métodos, cuál ha sido su desarrollo, sus ventajas y desventajas.

Métodos simplex

El primero de los métodos simplex fue publicado en 1962 por Spendley, Hext y Himsworth [110]. Los primeros métodos de búsqueda directa surgidos hasta ese momento requerían desde $2n$ hasta 2^n evaluaciones de la función objetivo, donde n es la dimensión del espacio, para completar la búsqueda de un punto que provocara un descenso en la función. El aporte en este trabajo es que sólo serían necesarios $n+1$ valores de la función objetivo para determinar un descenso. Esta observación tiene cierto sentido, dado que en efecto son $n+1$ evaluaciones de $f(x)$ las que se necesitan para estimar $\nabla f(x)$.

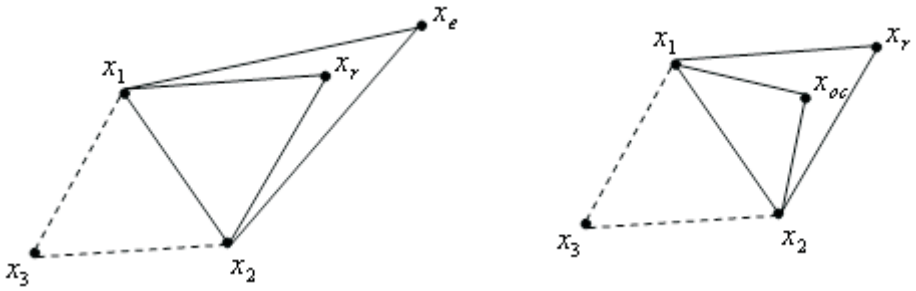
Figura 4.1: El simplex original, la reflexión de un vértice a través del punto medio del arco opuesto, y el simplex resultante.

Un simplex es una figura geométrica de dimensión n constituido por $n + 1$ puntos de $\mathbb{R}^n$. Por ejemplo un triángulo en $\mathbb{R}^2$ o un tetraedro en $\mathbb{R}^3$. Un simplex no sólo tiene la propiedad de definir el espacio, sino que además tiene la propiedad de que si se sustituye un vértice cualquiera reflejándolo a través del centro de gravedad de los puntos restantes, entonces el resultado también es un simplex (Figura 4.1).

El método propuesto en [110] toma un simplex inicial, identifica el "peor" de los vértices (el punto para el cual la función toma el mayor valor) y lo refleja a través del centro de gravedad de los puntos restantes, basándose en la conclusión lógica de que el "mejor" está en la dirección opuesta de este punto. Si el vértice reflejado sigue siendo el peor vértice, se toma el segundo peor vértice y se repite el proceso. Cada iteración consiste en construir un nuevo simplex, realizando reflexiones desde el peor de los puntos.

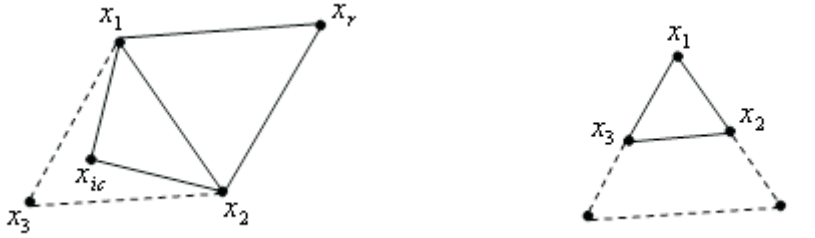
La cuestión difícil de responder en este método es cómo saber si se está en o cerca del mínimo de la función. En el propio artículo [110] se ilustra en dos dimensiones una secuencia cíclica de simplex, que podría ser interpretada como que indica que se identificó la vecindad del mínimo. Spendley, Hext y Himsworth sugieren dos alternativas para cuando el mejor punto no ha sufrido ninguna modificación en un buen número de iteraciones. La primera es reducir las longitudes de los arcos adyacentes al mejor punto, y la segunda es utilizar un método de optimización de primer o segundo orden para acelerar la convergencia local.

En 1965 fue publicado el método Nelder-Mead [94]. Este método es una versión mejorada del



(a) Expansión

(b) Contracción por fuera



(c) Contracción por dentro

(d) Recorte de distancias

Figura 4.2: Operaciones en el método Nelder-Mead

método simplex de Spendley, Hext y Himsworth, pues incorpora otros movimientos además de la reflexión para acelerar la búsqueda. Las acciones que incorporan son la contracción y la expansión, las cuales suplementan la reflexión (Fig. 4.2).

El método Nelder-Mead ha tenido una gran popularidad en la comunidad científica, especialmente en la dedicada a la ingeniería química y la medicina [134]. De todos los métodos de Búsqueda Directa es uno de los que más se han incorporado en librerías numéricas para paquetes de software.

Sin embargo, no es un método del todo robusto. Cuando el método trabaja bien puede encontrar la solución realizando muchas menos evaluaciones de la función objetivo que otros métodos directos, pero también puede fallar. En [134] se reportan casos donde el método ha convergido a puntos que no son estacionarios. En [76] se demuestra la convergencia para dimensión 1, y se obtienen resultados limitados en la convergencia en dos dimensiones. Además, en [88] se construye una familia de funciones en $\mathbb{R}^2$ para la cual el algoritmo Nelder-Mead no converge a puntos estacionarios, incluso dicha familia de funciones fue parametrizada para ser estrictamente convexa.

A pesar del hecho de que los métodos simplex son de los más populares y más usados de los métodos directos, no existen resultados satisfactorios sobre su convergencia a un punto estacionario. Por encima de eso, los ejemplos de McKinnon en [88] indican que no será posible probar la convergencia global del método Nelder-Mead para problemas de gran dimensión.

Métodos de búsqueda multidireccional

Los métodos de búsqueda multi-direccional son publicados por primera vez en 1989 por Torczon en [120]. En el método Nelder-Mead se realiza la búsqueda en una sola dirección, que es determinada por el punto en el cual la función objetivo alcanza el mayor valor y el centro del resto de los puntos. Los métodos de búsqueda multi-direccional incorporan a la estructura de los métodos simplex la idea de buscar en n direcciones distintas.

Al igual que los métodos simplex, forman un simplex inicial de $n+1$ puntos del espacio $\mathbb{R}^n$, que son ordenados ascendentemente según el valor de la función correspondiente. Las n aristas que conectan al mejor de los vértices con el resto definen n direcciones de búsquedas linealmente independientes. Las acciones de reflexión, expansión y contracción se realizan de forma distinta a los métodos simplex.

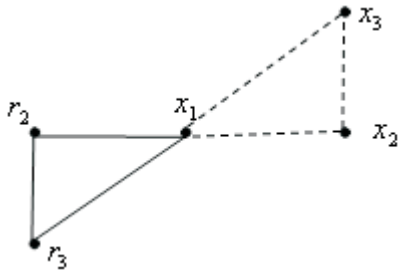
La reflexión consiste en reflejar cada uno de los vértices $x_2, \ldots, x_{n+1}$ a través de x_1 (Fig. 4.3(a)). La expansión consiste en expandir cada uno de los vértices reflejados en la misma dirección (Fig. 4.3(c)). Y la contracción es similar a la acción de "recortar distancias" en el Nelder-Mead (Fig. 4.3(b)).

Dos años más tarde, en 1991, se demostró la convergencia de la búsqueda multi-direccional [121].

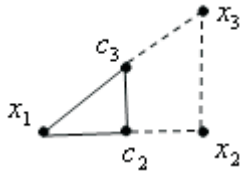
Métodos de búsqueda por patrones

Los métodos de búsqueda por patrones parten de un punto inicial x_0, una longitud de paso inicial Δ_0 y un conjunto de vectores que denominan patrones $\mathcal{D} = \{d_1, \ldots, d_p\};\ 1 \leq i \leq p, d_i \in \mathbb{R}^n$. En cada iteración k se calcula $f(x_k + \Delta_k d_i)$ para todos los p vectores del conjunto $\mathcal{D}$, hasta encontrar un vector d_i tal que $f(x_k + \Delta_k d_i) < f(x_k)$. Si ninguno de los vectores de $\mathcal{D}$ redujo la función objetivo se contrae el paso, de lo contrario, si al menos un vector logra reducir la función objetivo, se puede expandir el paso. El método termina cuando la longitud del paso sea suficientemente pequeña.

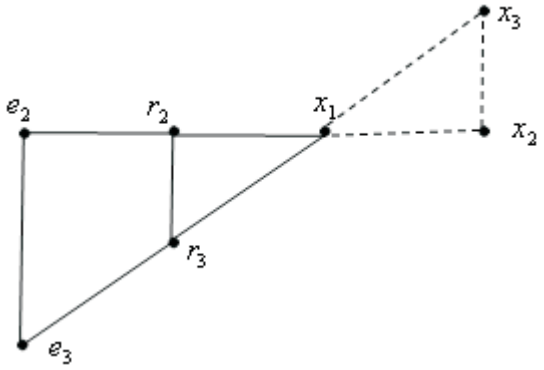
Para que los vectores de $\mathcal{D}$ puedan abarcar todo el espacio $\mathbb{R}^n$ deben formar una base generadora



(a) Reflexión

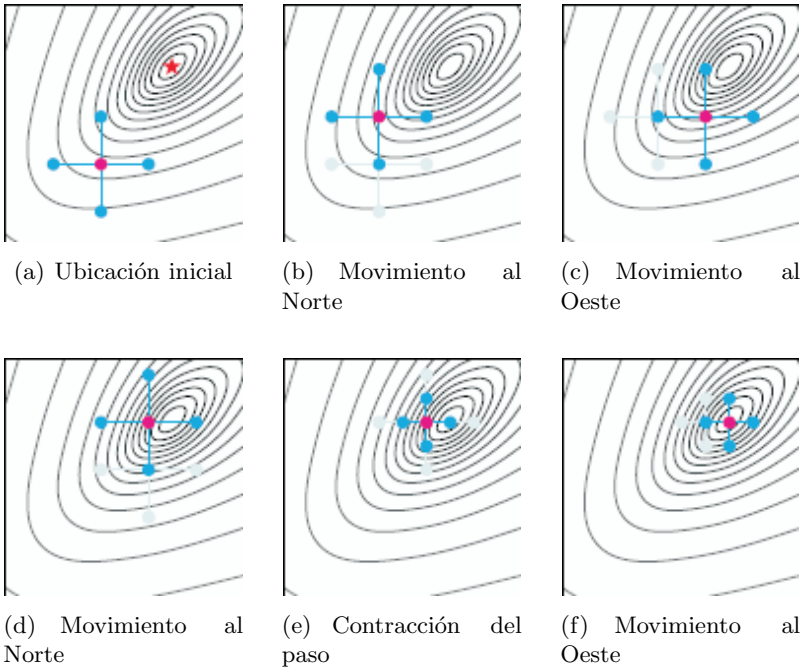


(b) Contracción



(c) Expansión

Figura 4.3: Operaciones en la búsqueda multidireccional



(a) Ubicación inicial (b) Movimiento al Norte (c) Movimiento al Oeste

(d) Movimiento al Norte (e) Contracción del paso (f) Movimiento al Oeste

Figura 4.4: Ejemplo de una búsqueda por patrones

del espacio, esto es que cualquier vector $v \in \mathbb{R}^n$ puede ser escrito como una combinación lineal con coeficientes no negativos de los vectores de $\mathcal{D}$, o sea, o sea, que existen $\alpha_1,\ \alpha_2, \ldots,\ \alpha_p$ tal que $v = \alpha_1 d_1 + \ldots + \alpha_p d_p$.

Este algoritmo sencillo se puede comprender aún más con el siguiente ejemplo tomado de [73]. Aquí se toman las direcciones:

$$\mathcal{D} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right\}$$

las cuales se pueden ver como homólogas a Norte, Sur, Este y Oeste. En la figura 4.4 se ilustran la situación inicial y las 5 primeras iteraciones del método. Con una estrella roja se señala el punto óptimo, con un punto magenta se señala la ubicación del punto de cada iteración, en azul oscuro las posibles trayectorias a tomar en la actual iteración y en azul claro las posibles trayectorias a tomar en la iteración anterior.

Nótese que ha medida que el punto de la iteración se acerca al mínimo de la función, el algoritmo reduce la longitud del paso.

A esta clase de algoritmos pertenece el propuesto por Hooke y Jeeves en [64]. En 1971, Céa demostró que el algoritmo de Hooke y Jeeves es globalmente convergente [23]. En ese mismo año Polak [101] demostró la convergencia global del algoritmo aplicado por Fermi y Metropolis descrito por Davidon en [33]. En ambos métodos las direcciones que se toman son las $2n$ direcciones coordenadas. En [122] se demostró la convergencia global de la generalización del método de búsqueda por patrones.

4.1.3 Métodos GSS

En [73] se logró formalizar una clase de algoritmos de búsqueda directa, que agrupa una gran variedad de algoritmos que habían aparecido en la literatura. A esta clase se le llamó *Generating Set Search Methods* (métodos GSS). El nombre se debe a que en la optimización sin restricciones es crucial que el conjunto de direcciones de búsqueda constituya una base generadora del espacio $\mathbb{R}^n$, es decir, que todo vector en $\mathbb{R}^n$ puede expresarse como una combinación lineal de estas direcciones con coeficientes no negativos.

Los métodos GSS agrupan la búsqueda multidireccional, los métodos de búsqueda por patrones, así como las extensiones introducidas por Lewis y Torczon en [82]. Todos estos métodos implementan sólo la condición de decrecimiento simple $f\left(x^{(k+1)}\right) < f\left(x^{(k)}\right)$. Los métodos GSS incluyen además métodos basados en condiciones de decrecimiento suficiente similares a

$$f\left(x^{(k)} + \alpha^{(k)} d^{(k)}\right) \leqslant f\left(x^{(k)}\right) + c_1 \alpha^{(k)} \nabla f\left(x^{(k)}\right) d^{(k)}$$

pero en las que no se usan derivadas. Entre estos algoritmos se incluyen aquellos presentados por Lucidi y Sciandrone [87] y García-Palomares y Rodríguez [48].

En el algoritmo 8 se ofrece la formalización de los métodos GSS como aparece en [73]

En este método las direcciones son las pertenecientes al conjunto $\mathcal{D}_k$, el cual contiene un conjunto generador $\mathcal{G}_k$ para $\mathbb{R}^n$. Existe la posibilidad adicional de incluir direcciones de búsquedas en $\mathcal{H}_k$. Este conjunto puede ser vacío, pero está abierta la posibilidad de incluirle direcciones que denoten heurísticas diseñadas para acelerar la búsqueda. El conjunto de direcciones $\mathcal{D}_k$ puede cambiar en cada iteración. Sin embargo, la medida del coseno para el conjunto $\mathcal{D}_k$, denotada por $\kappa\left(\mathcal{D}_k\right)$ debe estar acotada inferiormente. La medida del coseno se calcula de la forma siguiente:

Algoritmo 8 Método GSS

Entrada: Sean

$f : \mathbb{R}^n \to \mathbb{R}$ la función objetivo.
$x^{(0)} \in \mathbb{R}^n$ la aproximación inicial de la solución.
$\Delta_{tol} > 0$ la tolerancia de la longitud del paso.
$\Delta_0 > \Delta_{tol}$ el valor inicial de la longitud del paso.
$\theta_{\text{máx}} < 1$ la cota superior del parámetro de contracción.
$\rho : [0, +\infty) \to \mathbb{R}$

Salida: Mínimo de la función f

1: **para** $k = 0, 1, \ldots$ **hacer**
2: Sea $\mathcal{D}_k = \mathcal{G}_k \cup \mathcal{H}_k$
3: **si** existe $d^{(k)} \in \mathcal{D}_k$ tal que $f\left(x^{(k)} + \Delta_k d^{(k)}\right) < f\left(x^{(k)}\right) - \rho\left(\Delta_k\right)$ **entonces**
4: $x^{(k+1)} \leftarrow x^{(k)} + \Delta_k d^{(k)}$
5: $\Delta_{k+1} \leftarrow \phi_k \Delta_k$ donde $\phi_k \geq 1$ {*opcionalmente se expande el paso*}
6: **sino**
7: $x^{(k+1)} \leftarrow x^{(k)}$
8: $\Delta_{k+1} \leftarrow \theta_k \Delta_k$ donde $0 < \theta_k \leq \theta_{\text{máx}} < 1$ {*contrae el paso*}
9: **fin si**
10: **si** $\Delta_{k+1} < \Delta_{tol}$ **entonces**
11: **terminar**
12: **fin si**
13: **fin para**

$$\kappa\left(\mathcal{G}\right) = \min_{v \in \mathbb{R}^n} \max_{d \in \mathcal{G}} \frac{v^t d}{\|v\| \|d\|}$$

y es necesario que esté acotada inferiormente para evitar que las direcciones lleguen a ser muy malas, es decir, que con ellas el decrecimiento que se pueda obtener sea muy poco. Una de las razones por la cual una búsqueda con $\mathcal{D} = \{e_1, \ldots, e_n, -e_1, \ldots, -e_n\}$, donde e_i es la i-ésima dirección coordenada, se deteriora a medida que n crece, es porque $\kappa\left(\mathcal{D}\right) = \frac{1}{\sqrt{n}}$.

El método GSS ofrece flexibilidad en la medida de actualizar la longitud del paso en los pasos 2 y 3. Aquí se puede contraer o expandir el paso en proporciones distintas en cada iteración. No obstante, es necesario que θ_k esté acotado inferiormente por 0 y superiormente por $\theta_{\text{máx}} < 1$, para que el método pueda converger.

Un aspecto de interés es el criterio usado para aceptar una iteración. La introducción de una función $\rho\left(\cdot\right)$ permite usar el criterio de decrecimiento simple, o el criterio de decrecimiento suficiente. Se debe usar una función que satisfaga la condición $\rho(t)/t \to 0$ a medida que $t \to 0$ (una opción es $\rho\left(t\right) = \alpha t^2$ para algún $\alpha > 0$). Esta condición es una de las técnicas de globalización que se pueden

aplicar a los métodos GSS. Las estrategias de globalización en los métodos GSS son técnicas para garantizar que

$$\lim_{k \to +\infty} \Delta_k = 0$$

Usar un criterio de decrecimiento suficiente evita que el algoritmo escoja una dirección en la cual no se descienda mucho, por lo que puede acelerar la búsqueda del óptimo.

Los métodos GSS capturan la esencia de los clásicos métodos de búsqueda de patrones, mientras que abarca muchas de las más recientes ideas encontradas en la literatura de optimización no lineal. En [73] se analizan estrategias de globalización, que posibilitan que estos métodos converjan globalmente.

4.1.4 Búsqueda directa en entornos paralelos

El interés por los métodos de búsqueda directa renace en la comunidad científica cuando, a principios de la década de los 90, se publican trabajos sobre el tema en el contexto de la computación paralela. En [120], Torczon plantea que los métodos de búsqueda multi-direccional que proponía son inherentemente paralelos, pues las búsquedas en cada dirección son independientes entre sí y se pueden desarrollar de forma paralela. Sin embargo, en dicho trabajo el tema de la paralelización del algoritmo sólo se tocó entre las líneas futuras. En el año siguiente, 1991, se reportó en [34] una paralelización de la búsqueda multi-direccional, y se trazaron estrategias para balancear la carga en los procesadores. Sin embargo, los análisis se hicieron bajo la precondición de que el número de procesadores sea mayor que el tamaño del problema, y para los experimentos se probó el método para una función de dos variables.

La paralelización de métodos de búsqueda por patrones ha sido abordada en [65] y [74]. En el primero de los trabajos plantean un algoritmo paralelo síncrono, en el cual se distribuyen las direcciones por los procesadores de tal forma que, en cada iteración, cada procesador realiza la búsqueda a través de las direcciones que le fueron asignadas. Luego se realiza una reducción para determinar el punto en el cual la función alcanza el menor valor. Realizada la reducción se determina si hubo éxito o no en las búsquedas, y en dependencia de ello se actualiza la longitud del paso. Plantean que tal algoritmo puede provocar desequilibrio en la carga de trabajo de los procesadores, especialmente si

se trabaja en máquinas paralelas heterogéneas; y por tanto plantean un algoritmo paralelo asíncrono. Realizan las experimentaciones sobre varios problemas con determinadas dimensiones prefijadas. Los resultados que presentan son comparaciones entre el algoritmo paralelo síncrono y el asíncrono, donde el asíncrono alcanza mejores tiempos generalmente. Sin embargo, el trabajo no reporta los tiempos del algoritmo secuencial para tales problemas, por lo que no se puede saber cuánto se ganó en velocidad con los algoritmos paralelos presentados. En [74] se demuestra la convergencia global del algoritmo paralelo asíncrono de la búsqueda por patrones.

4.1.5 Ventajas y desventajas de la búsqueda directa

Los métodos de búsqueda directa fueron populares en los primeros años de la optimización numérica, debido en gran medida a que son fáciles de implementar, y que no requieren derivadas de la función objetivo. Sin embargo, el amplio desarrollo de los métodos basados en derivadas, así como la disponibilidad de implementaciones sofisticadas de estos métodos, incluyendo estrategias de globalización por búsqueda lineal o región de confianza, y opciones para generar aproximaciones del Gradiente y del Hessiano, hacen que en la actualidad la primera elección para resolver un problema de optimización sin restricciones para el cual se tiene información de la primera derivada, es algún método basado en gradientes. Si se tiene además información de la segunda derivada, la opción más usada es algún método Standard tipo Newton.

No obstante, los métodos de búsqueda directa aún tienen su aplicación en problemas de optimización que no se pueden resolver por métodos basados en derivadas, pues la función objetivo a optimizar, o bien no se tiene o es muy costoso obtener información de sus derivadas, o no está expresada de forma algebraica o analítica, o no es de naturaleza numérica. En algunos problemas de control la función objetivo es tan costosa que determinar el gradiente mediante el método de diferencias finitas no es nada práctico.

Hay que destacar que los métodos GSS poseen convergencia global a un punto estacionario, mientras que con los métodos basados en derivadas, en algunos problemas, se debe procurar que el punto inicial se encuentre cerca del mínimo pues en otro caso el método puede no converger.

Por otro lado, existen dos cuestiones que inevitablemente están presentes en cualquier método de

búsqueda directa. La primera es que los métodos de búsqueda directa son asintóticamente más lentos que los métodos de máximo descenso. Y la segunda es que a medida que el tamaño del problema crece, la ejecución de la búsqueda directa se deteriora

4.2 Algoritmos de búsqueda directa para problemas de optimización continua

En esta sección se describen y se analizan los métodos de búsqueda directa a los cuales se les desarrolló rutinas secuenciales. Todos los algoritmos pertenecen a la clase de métodos GSS. En una primera sección se realiza la descripción y en una segunda sección se realizan los análisis experimentales de los métodos aplicados a un problema de optimización continua.

4.2.1 Descripción de los métodos

El primer método que se presentará es un método GSS cuyo conjunto de direcciones siempre será el de las direcciones coordenadas. Se toma este método como punto de partida porque los otros métodos son resultado de aplicar diferentes variantes en algunas de sus acciones, bien sea en la elección del conjunto de direcciones de búsqueda y su correspondiente actualización en cada iteración, o en la estrategia de expansión y contracción del paso de búsqueda. Se incluyen algunas variantes que se proponen en trabajos anteriores, además de las diseñadas como parte de este trabajo.

Algoritmo gssG1

Este primer algoritmo es un método GSS cuyo conjunto de direcciones siempre será el conjunto, donde mediante e_i se denotan aquellos vectores de $\mathbb{R}^n$ cuyas componentes son todas nulas excepto la i-ésima que tiene valor unitario. Los parámetros de contracción y expansión tampoco se alteran en cada iteración, aunque pueden ser cualquier par de valores que cumplan las restricciones dadas en la formalización de los métodos GSS.

Dado que el algoritmo es iterativo, se calcula la complejidad temporal de cada iteración y el tiempo de ejecución del algoritmo quedaría expresado en función de la cantidad de iteraciones y el tiempo de

Algoritmo 9 gssG1

Entrada: Sean:

$f : \mathbb{R}^n \to \mathbb{R}$ la función objetivo.
$x^{(0)} \in \mathbb{R}^n$ la aproximación inicial de la solución.
$\Delta_{tol} > 0$ la tolerancia de la longitud del paso.
$\Delta_0 > \Delta_{tol}$ el valor inicial de la longitud del paso.
$\theta < 1$ el parámetro de contracción del paso.
$\phi \geq 1$ el parámetro de expansión del paso.
$\rho : [0, +\infty) \to \mathbb{R}$ una función continua decreciente
$\mathcal{D}_{\oplus} = \{\pm e_i : 1 \leq i \leq n\}$ el conjunto de direcciones

Salida: Mínimo de la función f

1: **para** $k = 0, 1, \ldots$ **hacer**
2: **si** existe $d^{(k)} \in \mathcal{D}_{\oplus}$ tal que $f\left(x^{(k)} + \Delta_k d^{(k)}\right) < f\left(x^{(k)}\right) - \rho\left(\Delta_k\right)$ **entonces**
3: $x^{(k+1)} \leftarrow x^{(k)} + \Delta_k d^{(k)}$
4: $\Delta_{k+1} \leftarrow \phi \Delta_k$
5: **sino**
6: $x^{(k+1)} \leftarrow x^{(k)}$
7: $\Delta_{k+1} \leftarrow \theta \Delta_k$
8: **fin si**
9: **si** $\Delta_{k+1} < \Delta_{tol}$ **entonces**
10: **terminar**
11: **fin si**
12: **fin para**

ejecución por iteración. En cada iteración se realiza una exploración por cada una de las direcciones, y en cada exploración se debe evaluar la función una vez. En el caso peor, que sucede cuando no se encuentra descenso en ninguna de las direcciones, se evalúa la función $2n$ veces. Además de eso, se efectúan $2n$ sumas de dos vectores de longitud n. Por tanto, siendo T_f el tiempo de ejecución del algoritmo que evalúa la función, y siendo k la cantidad de iteraciones del método, el tiempo de ejecución del método **gssG1** , dado en Flops, es:

$$T = k \cdot 2n \cdot (T_f + n) F \tag{4.1}$$

Algoritmo gssG2

Puede suceder que el punto inicial $x^{(0)}$ esté bastante alejado del óptimo de tal forma que, siendo x^* el óptimo, $x_i^* = x_i^{(0)} + \varepsilon_i \ ; \ 1 \leq i \leq n$ o $x_i^* = x_i^{(0)} - \varepsilon_i \ ; \ 1 \leq i \leq n$, donde los ε_i son valores reales positivos no necesariamente distintos. En esta situación, acercarse a x^* partiendo de $x^{(0)}$ mediante las

direcciones coordenadas puede resultar lento, pues a través de ellas sólo se puede aumentar o disminuir el valor de una componente del punto $x^{(k)}$.

La idea de este algoritmo es agregarle al conjunto de direcciones las direcciones $(1, 1, \ldots, 1)$ y $(-1, -1, \ldots, -1)$, pues son direcciones que, para situaciones como la descrita anteriormente, pueden acercar en pocas iteraciones el punto de la iteración $x^{(k)}$ al óptimo de la función. De modo que el algoritmo **gssG2** es similar al **gssG1** sólo que ahora, siendo $\mathcal{D}_{\oplus} = \mathcal{G}_{\oplus} \cup \mathcal{H}_{\oplus}$, $\mathcal{G}_{\oplus} = \{\pm e_i : 1 \leq i \leq n\}$ y $\mathcal{H}_{\oplus} = \{(1,\ 1, \ldots,\ 1),\ (-1,\ -1, \ldots,\ -1)\}$. En este algoritmo, las primeras direcciones con las que se exploraría serían con las de $\mathcal{H}$, si no se obtiene éxito con ninguna de ellas, se procede a explorar con las direcciones de $\mathcal{G}$.

Dado que la única diferencia que tiene **gssG2** con **gssG1** es que cuenta con $2n + 2$ direcciones, entonces su tiempo de ejecución es:

$$T = k \cdot (2n + 2)\,(T_f + n)\,F \tag{4.2}$$

Algoritmo gssEM

Este algoritmo se basa en una estrategia llamada Movimientos Exploratorios (*Exploratory Moves*), que se propone en [73], cuyo objetivo es disminuir el número de iteraciones del algoritmo.

La estrategia de Movimientos Exploratorios parte del hecho de que en la iteración $k - 1$ hubo éxito en la búsqueda a través de las direcciones. En ese caso, en la k-ésima iteración se realiza una búsqueda partiendo del punto $x_p = x^{(k)} + \left(x^{(k)} - x^{(k-1)}\right)$ en lugar de partir de $x^{(k)}$. La idea es que como el paso $x^{(k)} - x^{(k-1)}$ provocó un descenso en f en la iteración anterior, es posible que en una posterior iteración, dicho paso mejore nuevamente el valor de la función. Por tanto, aún cuando $f(x_p) \geq f\left(x^{(k)}\right)$ se realiza una búsqueda partiendo de x_p. Si en dicha búsqueda se encuentra un punto x_+ tal que $f(x_+) < f\left(x^{(k)}\right)$ entonces $x^{(k+1)} = x_+$. Si no se encontró ningún punto que disminuyera el valor de f, entonces la búsqueda se realiza a partir de $x^{(k)}$, procediendo de la forma análoga a los algoritmos anteriores.

Las direcciones que se tomaron para este algoritmo, son las mismas que las que se tomaron para **gssG2**.

El algoritmo 11 presenta los pasos de este esquema. Previamente se formaliza una subrutina con el algoritmo 4.2.1 que será usada varias veces en el algoritmo 11.

Algoritmo 10 Subrutina Search(f, x_p, $x^{(k)}$, $\mathcal{D}$, Δ_k, ρ, $x^{(k+1)}$, S_k)

1: **si** existe $d^{(k)} \in \mathcal{D}$ tal que $f\left(x_p + \Delta_k d^{(k)}\right) < f\left(x^{(k)}\right) - \rho\left(\Delta_k\right)$ **entonces**
2: $S_k \leftarrow$ **true**
3: $x^{(k+1)} = x_p + \Delta_k d^{(k)}$
4: **sino**
5: $x^{(k+1)} \leftarrow x^{(k)}$
6: $S_k \leftarrow$ **false**
7: **fin si**

Para calcular el tiempo de ejecución de cada iteración se debe tener en cuenta el caso peor. El caso peor sucede cuando en la búsqueda partiendo por x_p no se tenga éxito, y además tampoco se tenga éxito en la búsqueda partiendo por $x^{(k)}$. En este caso, el tiempo de ejecución sería el doble del tiempo de ejecución del algoritmo **gssG2**.

$$T = k \cdot (4n + 4)\left(T_f + n\right) F \tag{4.3}$$

No obstante, a pesar de que se invierte más cálculo, esta estrategia puede disminuir la cantidad de iteraciones del algoritmo.

Algoritmo gssDPCyc

Este algoritmo mantiene las $2n + 2$ direcciones que usa **gssG2**, pero incorpora una nueva estrategia para expandir la longitud del paso, y una estrategia para modificar el orden de las direcciones en el conjunto $\mathcal{D}$.

En cada iteración, el algoritmo realiza una búsqueda a partir de $x^{(k)}$. Supongamos que se tuvo éxito en la búsqueda y que $d^{(k)}$ es la dirección de descenso que se encontró. Supongamos además que en la iteración anterior también hubo éxito en la búsqueda y que $d^{(k-1)}$ es la dirección de descenso de la iteración $k - 1$. En ese caso el algoritmo chequea si $d^{(k)} = d^{(k-1)}$. De cumplirse esa condición, la longitud del paso para la iteración $k + 1$ se actualizaría como $\Delta_{k+1} \leftarrow 2\theta\Delta_k$. Esto se hace basándose

Algoritmo 11 gssEM

Entrada: Sean:

$f : \mathbb{R}^n \to \mathbb{R}$ la función objetivo.
$x^{(0)} \in \mathbb{R}^n$ la aproximación inicial de la solución.
$\Delta_{tol} > 0$ la tolerancia de la longitud del paso.
$\Delta_0 > \Delta_{tol}$ el valor inicial de la longitud del paso.
$\theta < 1$ el parámetro de contracción del paso.
$\phi \geq 1$ el parámetro de expansión del paso.
$\rho : [0, +\infty) \to \mathbb{R}$ una función continua decreciente
$\mathcal{D} = \{(1,\ 1, \ldots,\ 1),\ (-1,\ -1, \ldots,\ -1)\} \cup \{\pm e_i : 1 \leq i \leq n\}$ el conjunto de direcciones.
S_k la variable que indica si en la iteración k hubo éxito. Inicialmente $S_{-1} =$ **false**.

Salida: Mínimo de la función f

1: **para** $k = 0, 1, \ldots$ **hacer**
2: **si** $S_{k-1} =$ **false entonces**
3: Ir a 16
4: **fin si**
5: **si** $S_{k-1} =$ **true entonces**
6: $x_p = x^{(k)} + \left(x^{(k)} - x^{(k-1)}\right)$
7: Search(f, x_p, $x^{(k)}$, $\mathcal{D}$, Δ_k, ρ, $x^{(k+1)}$, S_k)
8: **si** $S_{k-1} =$ **false entonces**
9: Ir a 16
10: **sino**
11: $\Delta_{k+1} \leftarrow \theta\Delta_k$
12: Ir a 15
13: **fin si**
14: **fin si**
15: Search(f, $x^{(k)}$, $x^{(k)}$, $\mathcal{D}$, Δ_k, ρ, $x^{(k+1)}$, S_k)
16: **si** $S_{k-1} =$ **true entonces**
17: $\Delta_{k+1} \leftarrow \phi\Delta_k$
18: **sino**
19: $\Delta_{k+1} \leftarrow \theta\Delta_k$
20: **fin si**
21: **si** $\Delta_{k+1} < \Delta_{tol}$ **entonces**
22: **terminar**
23: **fin si**
24: **fin para**

en que si $d^{(k)} = d^{(k-1)}$, indica que $d^{(k)}$ es una dirección correcta de búsqueda y que los pasos hasta ahora resultaban cortos, por tanto se duplica el paso avanzar más en la próxima iteración.

Además de duplicar la medida de expansión de la longitud del paso, es deseable que la búsqueda en la próxima iteración comience con la dirección con la cual se tuvo éxito, por eso se rotan las direcciones del conjunto $\mathcal{D}_k$ de modo que la primera dirección del conjunto $\mathcal{D}_{k+1}$ sería $d^{(k)}$.

Se prefirió la idea de rotar las direcciones en lugar de simplemente colocar a $d^{(k)}$ como primera dirección de $\mathcal{D}_{k+1}$. Sea i la posición que ocupa $d^{(k)}$ en $\mathcal{D}_k$, denotemos a $d^{(k)}$ como $d_i^{(k)}$. Como la búsqueda se realiza según el orden en que están las direcciones en $\mathcal{D}_k$ entonces en las direcciones $d_1^{(k)}, \ldots, d_{i-1}^{(k)}$ no se alcanzó éxito. Si no se alcanzó éxito en dichas direcciones es poco probable que se tenga éxito en alguna de ellas en la siguiente iteración, por tanto pasan a ser las últimas opciones para la búsqueda, mientras que las direcciones $d_i^{(k)}, \ldots, d_{2n+2}^{(k)}$ pasan a ser las primeras en explorar en la iteración $k+1$. Esta estrategia de rotar las direcciones persigue el objetivo de incrementar las probabilidades de que se tenga éxito en la primera dirección del conjunto $\mathcal{D}_k$, y de esta forma disminuye el tiempo promedio de una iteración.

El caso peor de este algoritmo es similar al del método **gssG2**, por tanto la expresión del tiempo de ejecución es análoga. Sin embargo se puede prever que el tiempo promedio es menor, por lo que en la práctica puede arrojar mejores tiempos de ejecución.

Algoritmo gssUM

El algoritmo **gssDPCyc**, cuando encuentra éxito en una dirección determinada, duplica la longitud del paso y en la siguiente iteración comienza la búsqueda por dicha dirección. La estrategia que se propone en este algoritmo es, luego de encontrar una dirección de descenso, realizar una minimización de la función pero respecto a la longitud del paso, o lo que es lo mismo, siendo $d^{(k)}$ la dirección de descenso y $x^{(k)}$ el punto de la iteración k-ésima, el algoritmo **gssUM** minimiza la función unidimensional $g(\Delta) = f\left(x^{(k)} + \Delta d^{(k)}\right)$. Luego, encontrado el valor Δ^* que minimiza a g, el punto para la próxima iteración sería $x^{(k+1)} = x^{(k)} + \Delta^* d^{(k)}$.

La minimización unidimensional se realiza mediante el método *Golden Section Search*, haciendo

Algoritmo 12 gssDPCyc

Entrada: Sean:

$f : \mathbb{R}^n \to \mathbb{R}$ la función objetivo.
$x^{(0)} \in \mathbb{R}^n$ la aproximación inicial de la solución.
$\Delta_{tol} > 0$ la tolerancia de la longitud del paso.
$\Delta_0 > \Delta_{tol}$ el valor inicial de la longitud del paso.
$\theta < 1$ el parámetro de contracción del paso.
$\phi \geq 1$ el parámetro de expansión del paso.
$\rho : [0, +\infty) \to \mathbb{R}$ una función continua decreciente
$\mathcal{D}_0 = \{(1,\ 1, \ldots,\ 1),\ (-1,\ -1, \ldots,\ -1)\} \cup \{\pm e_i : 1 \leq i \leq n\}$

Salida: Mínimo de la función f

1: **para** $k = 0, 1, \ldots$ **hacer**
2: **si** existe $d^{(k)} \in \mathcal{D}_{\oplus}$ tal que $f\left(x^{(k)} + \Delta_k d^{(k)}\right) < f\left(x^{(k)}\right) - \rho\left(\Delta_k\right)$ **entonces**
3: $x^{(k+1)} \leftarrow x^{(k)} + \Delta_k d^{(k)}$
4: **si** $d^{(k)} = d^{(k-1)}$ **entonces**
5: $\Delta_{k+1} \leftarrow 2\phi\Delta_k$
6: **sino**
7: $\Delta_{k+1} \leftarrow \phi\Delta_k$
8: **fin si**
9: Sea i la posición de $d^{(k)}$ en $\mathcal{D}_k$, $\mathcal{D}_{k+1} \leftarrow \left\{d_i^{(k)}, \ldots, d_{2n+2}^{(k)}, d_1^{(k)}, \ldots, d_{i-1}^{(k)}\right\}$
10: **sino**
11: $x^{(k+1)} \leftarrow x^{(k)}$
12: $\Delta_{k+1} \leftarrow \theta\Delta_k$
13: **fin si**
14: **si** $\Delta_{k+1} < \Delta_{tol}$ **entonces**
15: **terminar**
16: **fin si**
17: **fin para**

previamente un *bracketing*. Se usan para ello rutinas como las que se proponen en [102]. Para hacer un *bracketing* a un mínimo en una función f, es necesario al menos dos valores reales a y b tal que $b > a$ y $f(a) \geq f(b)$, pues luego se puede calcular un tercer valor c para el cual $c > b$ y $f(c) \geq f(b)$. Como $d^{(k)}$ es una dirección de descenso entonces se cumple que $f\left(x^{(k)} + \Delta_k d^{(k)}\right) < f\left(x^{(k)}\right)$, o dicho en términos de la función g, se cumple que $g(\Delta_k) < g(0)$. Por lo que para hacer el *bracketing* al mínimo de g se toman $a = 0$ y $b = \Delta_k$.

Hecha la minimización unidimensional respecto a la longitud del paso, y actualizado el punto $x^{(k+1)}$, en la próxima iteración no tendría sentido comenzar a buscar por $d^{(k)}$. Por tanto, en **gssUM** se rotan las direcciones al igual que en **gssDPCyc**, pero en este caso la dirección $d_{i+1}^{(k)}$ sería la primera y $d_i^{(k)}$ la última, siendo i la posición que ocupa $d^{(k)}$ en el conjunto $\mathcal{D}_k$.

En la presentación del algoritmo se supone que existe una rutina **GoldenSearch** con el siguiente prototipo:

Subrutina GoldenSearch(f, $x^{(k)}$, $d^{(k)}$, Δ_k, tol, Δ^*)

que realiza la minimización unidimensional.

Este algoritmo añade la complejidad adicional del método *Golden Search*, por lo que el tiempo de ejecución por cada iteración aumenta. El método *Golden Search* primero realiza un *bracketing* al mínimo, y luego procede a buscar el mínimo. Tanto el *bracketing* como la búsqueda del mínimo son procesos iterativos en cuyas iteraciones el costo mayor lo lleva la evaluación de la función g. Por tanto, considerando que el caso peor en este algoritmo es que se tenga éxito en la última dirección del conjunto, y denotando por k_1 la cantidad de iteraciones consumidas por el *bracketing*, y por k_2 las iteraciones consumidas por el *Golden Search*, se puede aproximar el tiempo de ejecución del método **gssUM** a la siguiente expresión:

$$T = k \cdot ((2n+2)(T_f + n) + (k_1 + k_2) T_g) F \tag{4.4}$$

Como $T_g = T_f + n$ entonces:

Algoritmo 13 gssUM

Entrada: Sean:

$f : \mathbb{R}^n \to \mathbb{R}$ la función objetivo.
$x^{(0)} \in \mathbb{R}^n$ la aproximación inicial de la solución.
$\Delta_{tol} > 0$ la tolerancia de la longitud del paso.
$\Delta_0 > \Delta_{tol}$ el valor inicial de la longitud del paso.
$\theta < 1$ el parámetro de contracción del paso.
$\phi \geq 1$ el parámetro de expansión del paso.
$\rho : [0, +\infty) \to \mathbb{R}$ una función continua decreciente
$\mathcal{D}_0 = \{(1,\ 1, \ldots,\ 1),\ (-1,\ -1, \ldots,\ -1)\} \cup \{\pm e_i : 1 \leq i \leq n\}$

Salida: Mínimo de la función f

1: **para** $k = 0, 1, \ldots$ **hacer**
2: **si** existe $d^{(k)} \in \mathcal{D}_{\oplus}$ tal que $f\left(x^{(k)} + \Delta_k d^{(k)}\right) < f\left(x^{(k)}\right) - \rho\left(\Delta_k\right)$ **entonces**
3: GoldenSearch(f, x^k, d^k, Δ_k, tol, Δ^*)
4: $x^{(k+1)} \leftarrow x^{(k)} + \Delta^* d^{(k)}$
5: $\Delta_{k+1} \leftarrow \phi \Delta_k$
6: Sea i la posición de $d^{(k)}$ en $\mathcal{D}_k$, $\mathcal{D}_{k+1} \leftarrow \left\{d_i^{(k)}, \ldots, d_{2n+2}^{(k)}, d_1^{(k)}, \ldots, d_{i-1}^{(k)}\right\}$
7: **sino**
8: $x^{(k+1)} \leftarrow x^{(k)}$
9: $\Delta_{k+1} \leftarrow \theta \Delta_k$
10: **fin si**
11: **si** $\Delta_{k+1} < \Delta_{tol}$ **entonces**
12: **terminar**
13: **fin si**
14: **fin para**

$$T = k \cdot (2n + 2 + k_1 + k_2)\,(T_f + n)\,F \tag{4.5}$$

Algoritmo gssSum

En la sub-sección dedicada al algoritmo **gssG2** se explicó el por qué la inclusión de los direcciones $(1, 1, \ldots, 1)$ y $(-1, -1, \ldots, -1)$ en el conjunto de direcciones de búsqueda. Estas dos direcciones pueden acelerar la búsqueda siempre y cuando incrementar o decrementar cada componente del punto $x^{(k)}$ en una misma proporción conlleve a un nuevo punto que mejore el valor de f. Sin embargo, puede suceder que sólo se necesite modificar algunos de las componentes del punto, bien sean 2, 3 o hasta $n - 1$, para acercase al mínimo. En ese caso las dos direcciones incluidas no son útiles.

Se podría pensar entonces en incluir todas las direcciones compuestas por 0's y 1's y todas las direcciones compuestas por 0's y -1's, pero esa idea implicaría que se tengan 2^{n+1} direcciones de búsqueda, lo que haría al método de búsqueda directa muy costoso y para problemas medianamente grandes imposible de aplicarlo en la práctica.

En este algoritmo, **gssSum**, en cada iteración se exploran todas las direcciones, para saber cuáles son de descenso y cuáles no. Luego, siendo $d_{i_1}^{(k)}$, $d_{i_2}^{(k)}, \ldots,$ $d_{i_m}^{(k)}$ las direcciones de descenso, se construye una nueva dirección $d = d_{i_1}^{(k)} + d_{i_2}^{(k)} + \ldots + d_{i_m}^{(k)}$ con la cual probablemente se obtenga un mayor descenso. Como cabe la posibilidad de que d no sea una dirección de descenso, el algoritmo **gssSum** escoge entre d y $d_{i_1}^{(k)}$, $d_{i_2}^{(k)}, \ldots,$ $d_{i_m}^{(k)}$ la dirección que disminuye en mayor medida la función objetivo.

Para este algoritmo, no es necesario incluir las direcciones $(1, 1, \ldots, 1)$ y $(-1, -1, \ldots, -1)$, pues lo que en esencia se hace es explorar cuáles componentes del punto $x^{(k)}$ son necesarias modificar y cuáles no para acercarse al óptimo, y para ello bastan las direcciones coordenadas.

Este algoritmo supone un costo de tiempo por iteración mayor que el resto de los algoritmos vistos hasta ahora, pues siempre explora cada una de las $2n$ iteraciones. Como las $2n$ direcciones están compuestas por n pares de direcciones ortogonales, a lo sumo en cada iteración se van a encontrar n direcciones de descenso. Por lo tanto, el caso peor de este algoritmo es que en cada iteración se encontrasen n direcciones de descenso, pues se tendrían que hacer n sumas de vectores. El tiempo de

Algoritmo 14 gssSum

Entrada: Sean:

$f : \mathbb{R}^n \to \mathbb{R}$ la función objetivo.
$x^{(0)} \in \mathbb{R}^n$ la aproximación inicial de la solución.
$\Delta_{tol} > 0$ la tolerancia de la longitud del paso.
$\Delta_0 > \Delta_{tol}$ el valor inicial de la longitud del paso.
$\theta < 1$ el parámetro de contracción del paso.
$\phi \geq 1$ el parámetro de expansión del paso.
$\rho : [0, +\infty) \to \mathbb{R}$ una función continua decreciente
$\mathcal{D}_{\oplus} = \{\pm e_i : 1 \leq i \leq n\}$

Salida: Mínimo de la función f

1: **para** $k = 0, 1, \ldots$ **hacer**
2: **si** existen $d_{i_1}^{(k)}, \ldots, d_{i_m}^{(k)} \in \mathcal{D}_{\oplus}$ tal que $f\left(x^{(k)} + \Delta_k d_{i_j}^{(k)}\right) < f\left(x^{(k)}\right) - \rho\left(\Delta_k\right)$ con $1 \leq j \leq m$ **entonces**
3: $d \leftarrow d_{i_1}^{(k)} + d_{i_2}^{(k)} + \ldots + d_{i_m}^{(k)}$
4: $d^* \leftarrow \min_d f\left(x^{(k)} + \Delta_k d\right) : \left\{d, d_{i_1}^{(k)}, \ldots, d_{i_m}^{(k)}\right\}$
5: $x^{(k+1)} \leftarrow x^{(k)} + \Delta_k d^*$
6: $\Delta_{k+1} \leftarrow \phi \Delta_k$
7: **sino**
8: $x^{(k+1)} \leftarrow x^{(k)}$
9: $\Delta_{k+1} \leftarrow \theta \Delta_k$
10: **fin si**
11: **si** $\Delta_{k+1} < \Delta_{tol}$ **entonces**
12: **terminar**
13: **fin si**
14: **fin para**

ejecución quedaría expresado de la siguiente forma:

$$T = k \cdot \left(2n \cdot (T_f + n) + n^2\right) F \approx k \cdot 2n \cdot (T_f + n) F \tag{4.6}$$

4.2.2 Aplicación de los métodos al Problema Inverso Aditivo de Valores Singulares (*PIAVS*)

Los problemas inversos son de gran importancia para diferentes aplicaciones de la ciencia y la ingeniería. Un problema inverso específico es el Problema Inverso de Valores Singulares [27], y dentro de éste, el Problema Inverso Aditivo de Valores Singulares [43], el cual se define como:

Dado un conjunto de matrices reales A_0, A_1, ..., $A_n \in \mathbb{R}^{m \times n}$ $(m \geq n)$ y un conjunto de valores reales no negativos $\sigma^* = \{\sigma_1^*, \sigma_2^*, ..., \sigma_n^*\}$ tales que $\sigma_1^* \geq \sigma_2^* \geq ... \geq \sigma_n^*$, encontrar un vector $c = (c_1, c_2, ..., c_n)^t \in \mathbb{R}^n$ de modo que el conjunto de valores singulares de

$$A(c) = A_0 + c_1 A_1 + \ldots + c_n A_n$$

sea precisamente σ^*.

Este problema usualmente se formula como un sistema de ecuaciones no lineales, y se resuelve con diferentes variantes de métodos de Newton. En [43] se analizan algunas de estas variantes. Por otro lado, este problema puede verse también como un problema de optimización, pues se desea que la distancia entre los valores singulares dados σ^* y los valores singulares de $A(c)$, denotado como $\sigma(c)$, sea mínima. De modo que se define la función $f : \mathbb{R}^n \to \mathbb{R}$ como $f(c) = \|\sigma^* - \sigma(c)\|_2$, y el problema pasa a ser en encontrar un vector $c = (c_1, c_2, ..., c_n)^t \in \mathbb{R}^n$ que minimice $f(c)$.

El *PIAVS* es un problema bastante complejo, en primer lugar por el coste elevado de la evaluación de la función f, pues solamente el cálculo de los valores singulares de una matriz es aproximadamente $\frac{38}{3}n^3$; y en segundo lugar porque a medida que aumenta el tamaño del problema se hace más irregular la función, y es más complicado encontrar el mínimo.

Los primeros experimentos se realizaron con el problema particular donde las matrices presentan la siguiente definición:

$$A_0 = 0,\ A_k = \left(A_{ij}^k\right) \in \mathbb{R}^{n\times n}\ A_{ij} = \begin{cases} 1 & si\ i-j = k-1 \\ 0 & en\ otro\ caso \end{cases}$$

y el vector c^*, vector para el cual se cumple $\sigma\left(A\left(c^*\right)\right) = \sigma^*$, es aleatorio. Se escogió este caso en específico por su bajo costo en almacenamiento, pues la matriz $A\left(c\right)$ tendrá siempre la siguiente estructura

$$A\left(c\right) = \begin{bmatrix} c_1 & & & \\ c_2 & c_1 & & \\ \vdots & \vdots & \ddots & \\ c_n & c_{n-1} & \cdots & c_1 \end{bmatrix}$$

Los valores que se escogieron para los parámetros de los métodos de búsqueda directa fueron los siguientes:

- Tolerancia para la convergencia: $\Delta_{tol} = 10^{-4}$
- Longitud inicial del paso: $\Delta_0 = 16$
- Razón de expansión del paso: $\phi = 1$
- Razón de contracción del paso: $\theta = 0{,}5$
- Función para el decrecimiento suficiente: $\rho\left(t\right) = \frac{1}{2}t^2$

Para probar todos los algoritmos de búsqueda directa descritos en la sección anterior y compararlos según tiempos de ejecución, número de iteraciones para la convergencia y precisión de la solución encontrada, se escogieron problemas de tamaño 5, 10, 15, 20, 25 y 30. Para cada problema se fijó aleatoriamente un punto óptimo c^*, y los puntos iniciales $c^{(0)}$ se tomaron como $c_i^{(0)} = c_i^* + \delta_i,\ \ (i = 1:n)$, donde a δ_i se le dio los valores aleatorios entre 0,0 y 0,9.

En la figura 4.5 se muestran los promedios de las iteraciones que necesitaron los métodos de búsqueda directa para converger.

Los métodos **gssG1** y **gssG2** fueron de los de mayor cantidad de iteraciones en casi todos los casos, especialmente en los casos de mayor tamaño. Se puede observar también que en todos los casos,

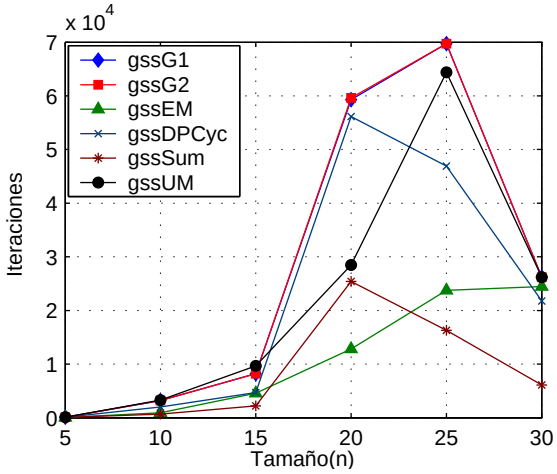


Figura 4.5: Gráfica Tamaño vs Iteraciones en los 6 métodos. Punto inicial: $c^{(0)} = c^* + (\delta_1, \ldots, \delta_n)^t$, $0 < \delta_i \leq 0{,}9$

los métodos **gssEM** y **gssSum** fueron los que menos iteraciones necesitaron para encontrar un punto mínimo, especialmente el método **gssSum** el cual a partir del tamaño $n = 25$ comienza a ampliar progresivamente su diferencia respecto al resto de los métodos.

En la figura 4.6 se muestran los promedios de las tiempos de ejecución.

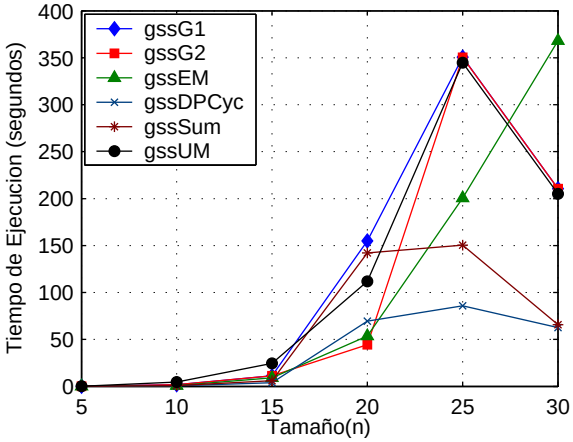


Figura 4.6: Gráfica Tamaño vs Tiempos en los 6 métodos. Punto inicial: $c^{(0)} = c^* + (\delta_1, \ldots, \delta_n)^t$, $0 < \delta_i \leq 0{,}9$

Por lo general los tiempos obtenidos son directamente proporcionales al número de iteraciones. Sin embargo sucede que el método **gssDPCyc** es el más rápido, a pesar de que tuvo mayor número de

iteraciones que los métodos **gssEM** y **gssSum**. Esto significa que la estrategia de rotar las direcciones de una iteración a otra, que aplica el método **gssDPCyc** con el objetivo de reducir el número de evaluaciones de la función objetivo por iteración, ha sido exitosa. A diferencia del **gssDPCyc**, los métodos **gssEM** y **gssSum** realizan muchas evaluaciones de la función objetivo en cada iteración, y al ser esta una función con alto costo temporal, la diferencia de los tiempos se hace más evidente.

Este Problema Inverso de Valores Singulares fue tratado como un sistema de ecuaciones no lineales, y abordado por un método de Newton con Regla de Armijo, siguiendo las ideas del método MI publicado en [43]. Para probar el método de Newton se escogió como punto inicial el mismo que se usó en los métodos de búsqueda directa en cada caso. En la figura 4.7 se muestran, por cada método incluyendo el método de Newton, los errores relativos entre los valores singulares de las matrices obtenidas y los que se tenían como dato de entrada. Se puede observar cómo en problemas cuyo tamaño es superior a 10 el error relativo obtenido con el método de Newton es mucho mayor que los obtenidos por los métodos de búsqueda directa. En esos casos, el método de Newton no logró converger según su criterio de convergencia, y los valores que se muestran en la figura son los alcanzados luego de llegar al máximo de iteraciones.

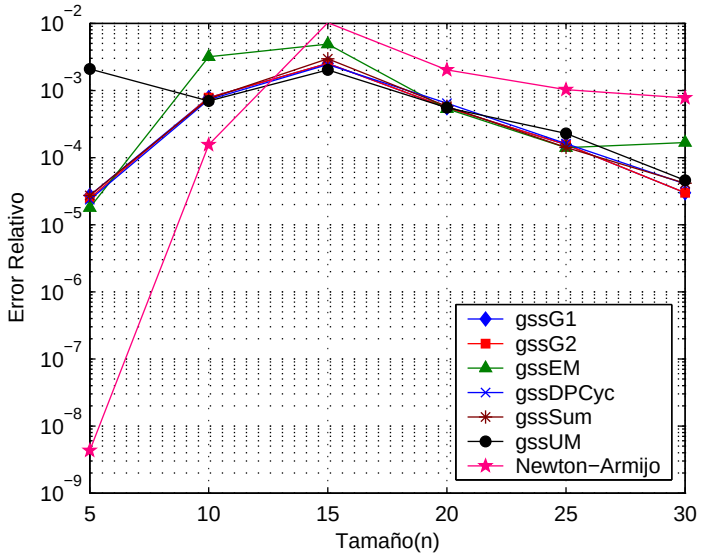


Figura 4.7: Gráfica Tamaño vs Errores Relativos en los 6 métodos + Newton-Armijo. Punto inicial: $c^{(0)} = c^* + (\delta_1, \ldots, \delta_n)^t$, $0 < \delta_i \leq 0{,}9$

Nótese también que, excepto en pocos casos, con los métodos de búsqueda directa se obtuvieron errores relativos en el orden de 10^{-3}, lo que significa que los valores singulares correspondientes a las soluciones encontradas por los métodos de búsqueda directa, son similares a los valores singulares de entrada.

De estos resultados se puede concluir que, para el problema IAVSP, los métodos **gssDPCyc** y **gssSum** fueron los de mejor rendimiento, porque alcanzaron soluciones con una precisión aceptable y fueron los que menos tiempo necesitaron para encontrarla.

A continuación se realiza una comparación de los dos métodos de mejores resultados (**gssDPCyc** y **gssSum**) con el método de Newton con Regla de Armijo para la solución de Sistemas de Ecuaciones No Lineales aplicado al mismo problema. Para este caso se usaron matrices generales aleatorias de tamaños $n = \{5, 10, 15, 20, 25, 30, 40, 50\}$. De forma similar a los experimentos anteriores, por cada problema se fijó aleatoriamente un punto óptimo c^*, y los puntos iniciales $c^{(0)}$ se tomaron como $c_i^{(0)} = c_i^* + \delta, \ \ (i = 1 : n)$, donde a δ se le dio los valores de 0,1, 1,1 y 10,1.

El método de Newton siempre convergió cuando $\delta = 0{,}1$, sin embargo cuando $\delta = 1{,}1$ sólo convergió en el caso de $n = 5$, y con $\delta = 10{,}1$ no convergió en ningún caso. Mientras, los métodos **gssDPCyc** y **gssSum** convergieron en todos los casos a óptimos relativamente cercanos a la solución del problema. Sin embargo, cuando el método de Newton converge, lo hace en un número de iteraciones mucho menor que los métodos de búsqueda directa, como se puede ver en la tabla 4.1.

n	5	10	15	20	25	30	40	50
gssDPCyc	82	630	2180	967	827	6721	7914	7178
gssSum	51	669	611	372	322	2237	1368	7292
Newton-Armijo	3	3	7	5	7	7	5	6

Tabla 4.1: Número de Iteraciones requeridas para la convergencia, $\delta = 0{,}1$

En la tabla 4.2 se muestran los tiempos de ejecución de los métodos gssDPCyc, y gssSum y Newton para el caso $\delta = 0{,}1$, cuando el método de Newton converge.

Estos resultados muestran que la aplicación de los métodos de búsqueda directa, en su variante secuencial, en problemas de alta complejidad no es práctica, dado que los tiempos de ejecución son demasiado costosos. Sin embargo, ellos muestran una propiedad muy interesante, al menos para este problema, y es que son mucho más robustos que el método de Newton cuando el punto inicial se

n	5	10	15	20	25	30	40	50
gssDPCyc	0,03	1,26	3,37	2,80	4,10	45,00	139,76	237,48
gssSum	0,02	0,50	2,96	4,12	6,77	86,21	170,34	2054,88
Newton-Armijo	0,00	0,00	0,01	0,01	0,02	0,04	0,08	0,25

Tabla 4.2: Tiempos de ejecución (segundos), $\delta = 0{,}1$

encuentra alejado de la solución.

4.3 Paralelización de los algoritmos de búsqueda directa

En la sección 4.2 se analizaron varios algoritmos de búsqueda directa y se aplicaron al Problema Inverso Aditivo de Valores Singulares. Según los resultados obtenidos, dos algoritmos fueron los más idóneos: el algoritmo **gssSum**, el cual realiza la búsqueda a través de las direcciones coordenadas y construye una nueva dirección sumando las direcciones de descenso, y el **gssDPCyc** el cual realiza la búsqueda a través de las direcciones coordenadas y las direcciones $(1, 1, \ldots, 1)$ y $(-1, -1, \ldots, -1)$, dobla la longitud del paso en caso que la dirección de descenso encontrada sea igual a la dirección encontrada en la iteración anterior, y realiza una rotación en el orden de las direcciones de modo que la dirección de descenso encontrada ocupe la primera posición en el conjunto de direcciones.

A pesar de que estos métodos fueron los de mejores resultados, sus tiempos de ejecución fueron elevados, sobre todo en los problemas más complejos con los cuales se hicieron experimentos con tamaños relativamente pequeños. En este sección se trata la paralelización de ambos métodos. Primeramente se describen y se analizan teóricamente los algoritmos paralelos diseñados, y posteriormente se muestran y se analizan los resultados experimentales obtenidos.

4.3.1 Diseño y análisis teórico de los algoritmos paralelos

Paralelización del método gssSum. Método pgssSum.

La sencillez de los métodos de búsqueda directa, en particular del método **gssSum**, unido a que las direcciones de búsquedas son independientes entre sí, hacen que su paralelización no sea un proceso difícil de resolver.

En el algoritmo **gssSum** se realizan $2n$ evaluaciones de la función objetivo, correspondientes a cada

una de las direcciones coordenadas. Estas evaluaciones no dependen entre sí, por lo que se pueden realizar de forma paralela. De modo que en el algoritmo paralelo inicialmente se distribuye por todos los procesadores el punto inicial x_0, la longitud del paso inicial Δ_0, la tolerancia para dicha longitud Δ_{tol} y las direcciones coordenadas que le corresponden. Siendo p el número de procesadores, a cada procesador le corresponden aproximadamente $2n/p$ direcciones. En cada iteración cada procesador evaluaría la función objetivo por cada dirección, y al terminar le envía a un procesador *root* las direcciones de descenso encontradas y los valores de f correspondientes. El procesador *root* se encargaría entonces de formar la nueva dirección, escoger entre esta y las enviadas cuál logra mayor descenso, actualizar el valor del punto para la próxima iteración, actualizar la longitud del paso y envíar a todos la terna $(x_{k+1},\ f(x_{k+1}),\ \Delta_{k+1})$. El algoritmo puede ser formalizado de la siguiente forma:

Algoritmo 15 pgssSum

En Paralelo : Para $pr = 0, 1, \ldots, p-1$
para $k = 0, 1, \ldots$ **hacer**
  si existen $d_{i_1}, \ldots, d_{i_{m(pr)}} \in \mathcal{D}_{pr}$ tal que $f\left(x^{(k)} + \Delta_k d_{i_j}\right) < f\left(x^{(k)}\right) - \rho(\Delta_k)$ con $1 \leq j \leq m$ **entonces**
    Enviar a P_{root} las direcciones $d_{i_1}, \ldots, d_{i_{m(pr)}}$
    Enviar a P_{root} los valores $f_{i_1}, \ldots, f_{i_{m(pr)}}$ donde $f_{i_j} = f\left(x^{(k)} + \Delta_k d_{i_j}\right)$
  fin si
  si $pr = root$ **entonces**
    Recibir los conjuntos de direcciones $\mathcal{D}_0, \ldots, \mathcal{D}_{p-1}$
    $\mathcal{D}^{(k)} \leftarrow \mathcal{D}_0 \cup \ldots \cup \mathcal{D}_{p-1}$
    si $\mathcal{D}^{(k)} \neq \emptyset$ **entonces**
      $d \leftarrow \mathcal{D}_1^{(k)} + \mathcal{D}_2^{(k)} + \ldots + \mathcal{D}_{m(k)}^{(k)}$ donde $\mathcal{D}_i^{(k)}$ es la dirección i de $\mathcal{D}^{(k)}$
      $d^* \leftarrow \min_d f(x_k + \Delta_k d) : \{d\} \cup \mathcal{D}^{(k)}$
      $x^{(k+1)} \leftarrow x^{(k)} + \Delta_k d^*$
      $\Delta_{k+1} \leftarrow \phi \Delta_k$
    sino
      $x^{(k+1)} \leftarrow x^{(k)}$
      $\Delta_{k+1} \leftarrow \theta \Delta_k$
    fin si
  fin si
  Difundir a todos desde P_{root} la terna $\left(x^{(k+1)},\ f\left(x^{(k+1)}\right),\ \Delta_{k+1}\right)$
  si $\Delta_{k+1} < \Delta_{tol}$ **entonces**
    terminar
  fin si
fin para

El tiempo aritmético de cada iteración del algoritmo es

$$T_A = \left(\frac{2n}{p} \left(T_f + n \right) + np \right) F \tag{4.7}$$

y el tiempo de comunicaciones de cada iteración, tomando como caso peor que cada procesador encuentra n/p direcciones de descenso, es:

$$\begin{aligned} T_C &= 2p \cdot \left(\frac{n}{p} \tau + \beta \right) + p \cdot \left(\left(n + 2 \right) \tau + \beta \right) \\ &\approx 2p \cdot \left(\frac{n}{p} \tau + \beta \right) + p \cdot \left(n\tau + \beta \right) \\ &= \left(p + 2 \right) n\tau + 3p\beta \approx pn\tau + 3p\beta \end{aligned} \tag{4.8}$$

Sumando ambos tiempos, y siendo k la cantidad de iteraciones del algoritmo, el tiempo paralelo viene dado por la siguiente expresión:

$$T_p = k \cdot \left(\left(\frac{2n}{p} \left(T_f + n \right) + np \right) F + pn\tau + 3p\beta \right) \tag{4.9}$$

Seguidamente se calcula el límite del *Speed-Up* del algoritmo, para estimar qué ganancia de velocidad se obtendría en un caso óptimo:

$$S_p = \frac{2n \left(T_f + n \right) F}{\left(\frac{2n}{p} \left(T_f + n \right) + np \right) F + pn\tau + 3p\beta} = \frac{\frac{2pT_f}{n} + 2p}{\frac{2T_f}{n} + 2 + \frac{p^2}{n} + \frac{p^2\tau}{n} + \frac{3p^2\beta}{n^2}} \tag{4.10}$$

$$\lim_{n \to \infty} S_p \approx \lim_{n \to \infty} \frac{\frac{2pT_f}{n} + 2p}{\frac{2T_f}{n}} = \lim_{n \to \infty} p + \frac{pn}{T_f} = p + \lim_{n \to \infty} \frac{pn}{T_f} \tag{4.11}$$

El límite de la ganancia de velocidad depende de la complejidad temporal de la función objetivo. En cualquier caso, si el costo de la función objetivo es $O\left(n \right)$ o superior, el *Speed-Up* tiene como límite a p.

Paralelización del método gssDPCyc. Método pgssDPCyc

El método **gssDPCyc** es de una sencillez similar al método **gssSum**. También este método debe hacer una búsqueda a través de varias direcciones independientes entre sí, en este caso $2n + 2$. Por tanto se puede aplicar una distribución de las direcciones similar a la realizada en el algoritmo **pgssSum**, se distribuyen las 2n+2 entre los p procesadores, de modo que cada procesador realiza la búsqueda a través de las direcciones que le fueron asignadas.

Sin embargo, a diferencia del método **gssSum** que evalúa la función en todas las direcciones, el método **gssDPCyc** no necesariamente evalúa la función en todas las direcciones, pues en cuanto encuentra una dirección de descenso detiene la búsqueda. Un algoritmo paralelo que distribuya el conjunto de direcciones equitativamente por los procesadores, y donde cada procesador realiza una búsqueda a través de sus correspondientes direcciones hasta que encuentre una de descenso, puede acusar un notable desequilibrio en la carga de trabajo de cada procesador. Esto es porque cada procesador debe reportarle a un procesador *root* sus resultados en la búsqueda, bien si tuvo éxito o si no lo tuvo. Entonces para tomar una decisión, el procesador *root* debe esperar por los resultados de cada procesador. Es muy probable que un procesador termine primero su búsqueda que otro. Puede ser incluso que un procesador realice una sola evaluación de la función, mientras que otro realice el número máximo que le corresponde. Es decir, puede pasar que mientras un procesador haya terminado su trabajo, otros estén trabajando aún.

Por tanto se deben buscar alternativas para lograr un mayor equilibrio en la carga de trabajo en los procesadores. Como se comentó en el Capítulo 2, generalmente intentar equilibrar la carga de trabajo implica aumentar el número de sincronizaciones, lo que implicaría aumentar el tiempo de comunicaciones del algoritmo. En el algoritmo paralelo **pgssDPCyc** se realiza la distribución de las direcciones explicada anteriormente, y se plantea la idea de que cada procesador, después de haber buscado en m direcciones, debe reportarle al resto los resultados de su búsqueda. De modo, que cada m evaluaciones de la función objetivo, todos los procesadores saben cuál ha tenido éxito y cuál no. En caso de que ninguno tuviera éxito, prosiguen la búsqueda en m direcciones más, y así hasta que se agoten sus direcciones.

Con el algoritmo paralelo **pgssDPCyc** se puede introducir una estrategia para acelerar la convergencia del método. En el momento en que todos reportan los resultados de sus búsquedas, puede suceder que más de un procesador haya tenido éxito, es decir, que existe más de una dirección de descenso. En ese caso se escogería aquella con la que se logra un mayor descenso. Escogida la dirección de descenso, las actualizaciones del punto para la próxima iteración y de la longitud del paso las haría el procesador a quien le corresponde dicha dirección. El mismo procesador es quien difundiría a todos la terna $\left(x^{(k+1)},\ f\left(x^{(k+1)}\right),\ \Delta_{k+1}\right)$.

Para que las direcciones se exploren en el mismo orden a como se hace en el algoritmo secuencial, la distribución de las direcciones ha de ser cíclica a través de los procesadores. El algoritmo 16 es la formalización de la versión paralela **pgssDPCyc**.

Para calcular el tiempo aritmético por iteración se supone el caso peor, que es cuando ningún procesador encuentra una dirección de descenso. En ese caso

$$T_A \approx \frac{2n+2}{p}\left(T_f + n\right)F \tag{4.12}$$

Para calcular el tiempo de comunicaciones por iteración también se toma el mismo caso peor. En ese caso, en cada iteración se realizan $(2n+2)/mp$ sincronizaciones, que son $(2n+2)/mp$ mensajes de dos palabras desde todos a hacia todos. Por tanto, la expresión del tiempo de comunicaciones por iteración es:

$$\begin{aligned} T_C &= \frac{2n+2}{mp}\cdot\left(2\tau+\beta\right) + p\cdot\left(\left(n+2\right)\tau+\beta\right) \\ &\approx \frac{2n+2}{mp}\cdot\left(2\tau+\beta\right) + p\cdot\left(n\tau+\beta\right) \end{aligned} \tag{4.13}$$

Es necesario destacar que el paso de la línea 15 del algoritmo no requiere de comunicación alguna, pues cada procesador, sabiendo cuál es la dirección de éxito de la iteración actual, puede saber cuáles direcciones le tocan en la iteración siguiente.

Finalmente:

$$T_p = k\cdot\left(\frac{2n+2}{p}\left(T_f+n\right)F + \frac{2n+2}{mp}\left(2\tau+\beta\right) + p\left(n\tau+\beta\right)\right) \tag{4.14}$$

Algoritmo 16 pgssDPCyc

En Paralelo : Para $pr = 0, 1, \ldots, p-1$
para $k = 0, 1, \ldots$ **hacer**
  $S_k \leftarrow false$; $i \leftarrow 0$
  mientras $i < (2n+2)/p$ **hacer**
    si existe j tal que $i \leq j \leq i+m$ y $d_j \in \mathcal{D}_{pr}$ cumpla con $f\left(x^{(k)} + \Delta_k d_j\right) < f\left(x^{(k)}\right) - \rho\left(\Delta_k\right)$ **entonces**
      $S_k \leftarrow true$
    fin si
    Difundir a todos desde todos S_k y j
    Determinar la mejor dirección de descenso y su procesador correspondiente. Sea P_s tal procesador.
    si hubo éxito **entonces**
      $S_k \leftarrow true$
      Ir a 16
    fin si
    $i \leftarrow i + m$
  fin mientras
  si $S_k = true$ **entonces**
    si $pr = s$ **entonces**
      $x^{(k+1)} \leftarrow x^{(k)} + \Delta_k d_j$
      si $d_j = d^{(k-1)}$ **entonces**
        $\Delta_{k+1} \leftarrow 2\phi\Delta_k$
      sino
        $\Delta_{k+1} \leftarrow \phi\Delta_k$
      fin si
      Difundir a todos desde P_s la terna $\left(x^{(k+1)},\ f\left(x^{(k+1)}\right),\ \Delta_{k+1}\right)$
    sino
      $x^{(k+1)} \leftarrow x^{(k)}$
      $\Delta_{k+1} = \theta\Delta_k$
    fin si
  fin si
  Rotar y distribuir cíclicamente las direcciones.
  si $\Delta_{k+1} < \Delta_{tol}$ **entonces**
    terminar
  fin si
fin para

Según la expresión del tiempo paralelo, los menores tiempos se lograrían mientras el valor de m sea mayor, pues disminuiría el tiempo de comunicaciones. Sin embargo, a medida que m aumenta, el tiempo aritmético promedio aumenta, e incrementa además el desequilibrio de la carga en los procesadores.

Es de suponer que la cantidad de iteraciones del algoritmo paralelo sea distinta a la del algoritmo secuencial. Por tanto, para este caso sólo se calcula el *Speed-Up* por iteración.

$$\begin{aligned} S_p &= \frac{(2n+2)(T_f+n)F}{\frac{2n+2}{p}(T_f+n)F + \frac{2n+2}{mp}(2\tau+\beta) + p(n\tau+\beta)} \\ &\approx \frac{2n(T_f+n)F}{\frac{2n}{p}(T_f+n)F + \frac{2n}{mp}(2\tau+\beta) + p(n\tau+\beta)} \\ &= \frac{\frac{2pT_f}{n} + 2p}{\frac{2T_f}{n} + 2 + \frac{2}{nm}(2\tau+\beta) + \frac{p^2}{n}\left(\tau+\frac{\beta}{n}\right)} \end{aligned} \tag{4.15}$$

El cálculo del límite del *Speed-Up* resulta similar al límite del Speed-Up del algoritmo **pgssSum**.

Paralelización asíncrona del método gssDPCyc. Método pgssDPCycMS

En este trabajo se diseñó un algoritmo paralelo asíncrono del método secuencial **gssDPCyc**. El algoritmo presenta un esquema *master-slave*. En este esquema un procesador *master* es el que lleva todo el control del proceso de búsqueda, y la función del resto de los procesadores, llamados *slaves*, es evaluar la función objetivo y enviar el resultado al *master*. Cada procesador *slave* tiene un conjunto de direcciones de búsqueda, evalúa la función para cada dirección, y luego de cada evaluación envía el resultado al *master* a través de un mensaje no bloqueante. El mensaje no bloqueante permite que el procesador *slave* continúe su ejecución aún cuando no haya llegado el mensaje enviado a su destino. Por su lado el procesador master va recibiendo los valores de la función en cada dirección, y en cuanto detecte un descenso en la función envía a los procesadores *slaves* un mensaje para que detengan sus evaluaciones. Es el procesador *master* quien actualiza el valor del punto para la próxima iteración y el valor de la longitud del paso, y envía tales parámetros a los procesos *slaves*, además de cuál fue la

dirección de éxito, para que puedan realizar la rotación cíclica de las direcciones de búsqueda.

Con este algoritmo no se puede aplicar ninguna estrategia para acelerar la búsqueda parecida a la aplicada en el algoritmo **pgssDPCyc**, pues sólo recibe a la vez un valor de la función. Por tanto, tiene un esquema de ejecución similar al algoritmo secuencial. Sin embargo, no se puede esperar que converja en el mismo número de iteraciones. La particularidad de que los mensajes sean no bloqueantes y que el *master* siempre está a la espera del primer mensaje que le llegue, no importa de cuál procesador sea, hace que a los efectos del *master* las direcciones no se evalúan siempre en el orden preestablecido. Por eso puede suceder, con significativa probabilidad, que el número de iteraciones del algoritmo paralelo sea distinto al del algoritmo secuencial.

Una cuestión que se debe tener en cuenta es que cuando el procesador *master* recibe un valor de la función de un procesador *slave*, en el mensaje se debe contener también a qué iteración corresponde esa evaluación, pues puede suceder que sea un mensaje de la iteración anterior que el *master* no recibió. Para que un procesador *slave* pueda enviar el número de la iteración conjuntamente con el valor de la función, el procesador *master* debe enviárselo previamente, preferentemente en el mismo mensaje en el cual se envían $\left(x^{(k+1)},\ \Delta_{k+1},\ d_k^*\right)$

En el algoritmo 17 se formalizan los pasos que ejecutaría el procesador *master* y en el algoritmo 18 se formalizan los pasos que ejecutarían los procesadores *slaves*. En ambos algoritmos se ha supuesto, sin pérdida de generalidad, que el procesador 0 es el procesador *master*.

Un inconveniente que presenta este algoritmo es que tiene un procesador menos para hacer la búsqueda y realizar los cálculos de la función. Sin embargo añade la ventaja de solapar tiempos aritméticos con tiempos de comunicaciones.

4.3.2 Resultados experimentales de los métodos paralelos

Los métodos paralelos fueron desarrollados en el entorno paralelo MPI (*Message Passing Interface*) [109]. El elevado coste de la función objetivo en este problema condicionó que se tomaran los tamaños n: 72, 96, 120, 144 para realizar las experimentaciones secuenciales en un tiempo razonable. Los experimentos se realizaron en el cluster Kefren descrito en el capítulo 2.

Algoritmo 17 pgssDPCycMS: Algoritmo del *Master*

1: **para** $k = 0, 1, \ldots$ **hacer**
2: Enviar a todos $\left(x^{(k)},\ \Delta_k,\ d^*_{k-1},\ k\right)$
3: $S_k \leftarrow false$; $i \leftarrow 0$
4: **mientras** $i < 2n + 2$ **hacer**
5: Recibir de cualquier procesador $(f_{trial},\ d,\ itr)$
6: **si** $itr = k$ **entonces**
7: **si** $f_{trial} < f\left(x^{(k)}\right) - \rho\left(\Delta_k\right)$ **entonces**
8: $S_k \leftarrow true$
9: Enviar a todos mensaje SUCCESS
10: Ir a 15
11: **fin si**
12: $i \leftarrow i + 1$
13: **fin si**
14: **fin mientras**
15: **si** $S_k = true$ **entonces**
16: $x^{(k+1)} \leftarrow x^{(k)} + \Delta_k d$
17: **si** $d = d^*_{k-1}$ **entonces**
18: $\Delta_{k+1} \leftarrow 2\phi\Delta_k$
19: **sino**
20: $\Delta_{k+1} \leftarrow \phi\Delta_k$
21: **fin si**
22: $d^*_k \leftarrow d$
23: **sino**
24: $x^{(k+1)} \leftarrow x^{(k)}$
25: $\Delta_{k+1} \leftarrow \theta\Delta_k$
26: **fin si**
27: **si** $\Delta_{k+1} < \Delta_{tol}$ **entonces**
28: Enviar a todos mensaje TERMINAR
29: **terminar**
30: **sino**
31: Enviar a todos mensaje CONTINUAR
32: **fin si**
33: **fin para**

Algoritmo 18 pgssDPCycMS: Algoritmo del *Slave*

1: **En Paralelo : Para** $pr = 1, 2, \ldots, p-1$
2: **mientras** *true* **hacer**
3: Recibir de P_0 $\left(x^{(k)},\ \Delta_k,\ d^*_{k-1},\ k\right)$
4: Rotar y distribuir cíclicamente las direcciones.
5: $S_k \leftarrow false$; $i \leftarrow 0$
6: **mientras** $i < (2n+2)/(p-1)$ **hacer**
7: $f_{trial} = f\left(x^{(k)} + \Delta_k d_i\right)$
8: Enviar a P_0 $(f_{trial},\ d_i,\ k)$
9: **si** hay un mensaje SUCCESS en la cola de mensajes **entonces**
10: Ir a 14
11: **fin si**
12: $i \leftarrow i + 1$
13: **fin mientras**
14: Recibir de P_0 un mensaje CONTINUAR o TERMINAR
15: **si** el mensaje recibido es TERMINAR **entonces**
16: **terminar**
17: **fin si**
18: **fin mientras**

Tiempos de ejecución y ganancia de velocidad.

En las figuras 4.8, 4.9 y 4.10 se muestran las gráficas de las ganancias de la velocidad de los tres métodos paralelos.

En sentido general se obtienen en los tres algoritmos buenos valores de *Speed-Up*. En el caso del algoritmo **pgssSum** el *Speed-Up* que se obtiene es casi el óptimo. Se puede observar como con el algoritmo **pgssDPCyc** se obtienen ganancias de velocidades por encima del óptimo teórico, que es el número de procesadores. Esto puede ser por diferentes razones, la principal es que el algoritmo paralelo no sigue una misma estrategia de búsqueda que el algoritmo secuencial, por lo que el número de iteraciones, y más aún, el número de evaluaciones de la función objetivo es diferente en ambos algoritmos. En la mayoría de los casos, sobre todo en los de tamaño pequeño, el número de iteraciones del algoritmo paralelo fue menor que el secuencial. Los elevados *Speed-Up* obtenidos indican que también el número de evaluaciones de la función es menor en el algoritmo paralelo que en el secuencial.

Con el algoritmo asíncrono se alcanzaron buenas ganancias de velocidad y en sentido general los valores obtenidos están dentro de los parámetros normales. Sin embargo, de los tres métodos paralelos fue el menos rápido como se puede apreciar en las tablas 4.3, 4.4 y 4.5. El hecho de que use un

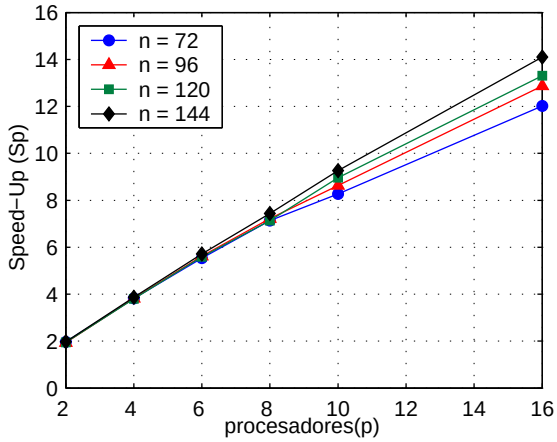


Figura 4.8: Gráfica Speed-Up vs Procesadores en el método **pgssSum**

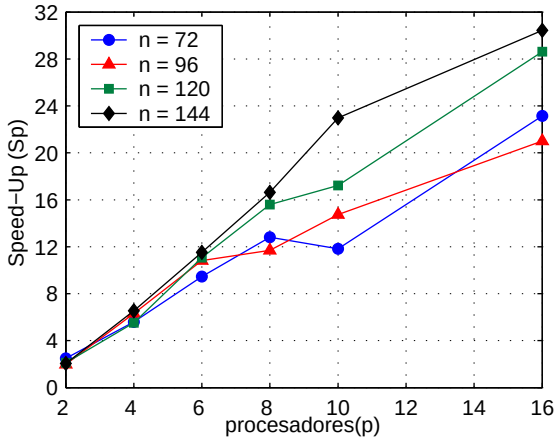


Figura 4.9: Gráfica Speed-Up vs Procesadores en el método **pgssDPCyc**

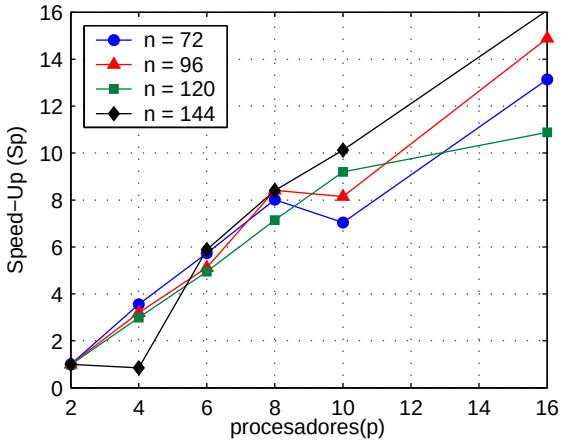


Figura 4.10: Gráfica Speed-Up vs Procesadores en el método **pgssDPCycMS**

procesador menos para realizar evaluaciones de la función objetivo no parecer ser la principal razón, pues en los casos en que $p = 10$ y $p = 16$ la influencia de un procesador menos para el cálculo no debe ser muy importante.

Comparando el rendimiento de los algoritmos **pgssDPCyc** y **pgssSum**, vemos que el rendimiento del algoritmo **pgssDPCyc** se deteriora considerablemente a medida que el tamaño del problema aumenta, a diferencia de los casos de tamaños pequeños (ver Tabla 4.3). Con un solo procesador, el algoritmo **pgssSum** es dos veces más rápido para problemas grandes.

tamaño \ procesadores	1	2	4	6	8	10	16
72	763,17	307,35	136,63	80,61	59,55	64,59	32,97
96	3929,22	1982,79	621,80	362,94	336,29	266,70	187,07
120	10430,27	4961,54	1891,92	943,07	669,25	605,65	364,46
144	36755,57	17935,18	5626,43	3187,75	2209,05	1599,85	1207,25

Tabla 4.3: Tiempos de Ejecución (segundos) pgssDPCyc

tamaño \ procesadores	1	2	4	6	8	10	16
72	278,018	141,40	72,86	50,27	38,98	33,61	23,14
96	2710,31	1400,35	710,02	482,63	375,27	313,90	210,61
120	5020,20	2581,44	1322,34	897,56	703,73	560,50	377,32
144	14497,76	7349,41	3749,35	2537,32	1948,71	1565,46	1028,03

Tabla 4.4: Tiempos de Ejecución (segundos) pgssSum

Sin embargo, cuando el número de procesadores aumenta, el algoritmo **pgssSum** obtiene buenos

tamaño \ procesadores	1	2	4	6	8	10	16
72	763,17	762,68	214,66	133,10	95,26	108,45	58,11
96	3929,22	3940,86	1228,57	767,24	467,13	482,17	263,99
120	10430,27	10614,80	3494,83	2108,65	1459,55	1134,09	958,64
144	36755,57	36786,43	12258,02	6244,33	4375,89	3630,06	2286,03

Tabla 4.5: Tiempos de Ejecución (segundos) pgssDPCycMS

valores de *Speed-Up*, pero el **pgssDPCyc** obtiene valores de Speed-Up aún mayores. De modo que cuando el número de procesadores es alto, los tiempos de ejecución de **pgssDPCyc** y **pgssSum** son similares.

En sentido general, con el algoritmo **pgssSum** se obtuvieron los mejores resultados.

Escalabilidad.

En los experimentos para la escalabilidad se deben aumentar el tamaño del problema, que se denota por ω , y la cantidad de procesadores en la misma proporción. Se debe fijar un p inicial, que en nuestro caso siempre será 2, y un tamaño del problema inicial. Los experimentos se hicieron doblando el número de procesadores y a la vez doblando el tamaño del problema. Luego, del tamaño del problema se calcula el valor de la n correspondiente.

En la Tabla 4.6 se muestran los valores de ω, n y p correspondientes a los experimentos con este problema.

p	n	$\omega = 2n^4$
2	75	63281250
4	90	126562500
8	108	253125000
16	128	506250000

Tabla 4.6: Valores de p y n para los experimentos de escalabilidad en el IASVP

En la figura 4.11 se muestra el *Speed-Up* escalado de los 3 métodos paralelos. Se puede observar que todos poseen una escalabilidad razonablemente buena.

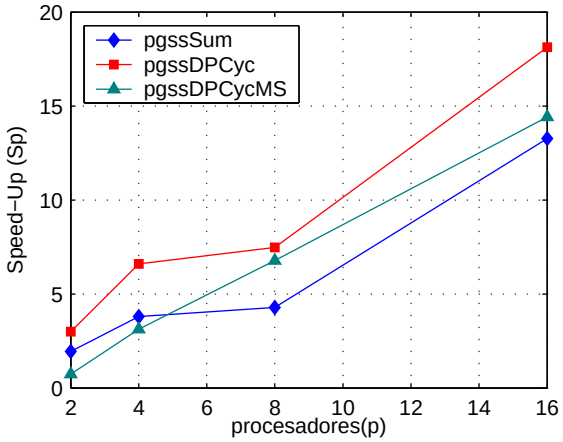


Figura 4.11: Gráfica Speed-Up escalado de los métodos **pgssSum**, **pgssDPCyc** y **pgssDPCycMS**

4.4 Aplicación de la búsqueda directa en la decodificación de señales de sistemas MIMO

4.4.1 Introducción

Recordemos que los métodos de búsqueda directa pueden tener su aplicación en aquellos problemas de optimización en los cuales no exista o sea muy costoso obtener información de las derivadas de la función objetivo. Tal es el caso de los problemas de optimización en espacios discretos, los cuales no son de naturaleza numérica y por tanto sólo se podría comparar los valores de la función objetivo para determinar qué valor es mejor o peor que el valor que se tiene hasta ese momento.

En esta sección se describe la aplicación de los métodos de búsqueda directa diseñados y descritos en la sección 4.2 al problema de encontrar el punto más cercano en una retícula, el cual está formulado y descrito en el capítulo 3.

4.4.2 Configuración de los parámetros de la búsqueda directa

Para poder aplicar al problema cada uno de los métodos de búsqueda directa descritos es necesario configurar cada uno de sus parámetros:

- **Función objetivo:** Será la función $f : \mathcal{A}^m \rightarrow \mathbb{R}^n$ tal que $f(s) = \|x - Hs\|_2, s \in \mathcal{A}^m$, donde $\mathcal{A}$

es la constelación de símbolos.

- **Punto inicial $s^{(0)}$:** Será generado aleatoriamente, aunque se puede tomar como punto inicial la solución obtenida mediante algún método heurístico.

- **Longitud inicial del paso Δ_0:** Para garantizar que sea suficiente para explorar toda la retícula, Δ_0 debe ser inicializado con el valor de $L \cdot l$, donde L es la cantidad de símbolos de la constelación y l es la distancia que hay entre cualquier par de símbolos adyacentes de $\mathcal{A}$

- **Tolerancia para la longitud del paso Δ_{tol}:** Cualquier número real menor que l, pues el valor mínimo de la longitud del paso con el cual los métodos de búsqueda directa trabajarían sería precisamente l

- **Expansión del paso ϕ:** Se puede mantener el mismo usado en la solución del IASVP, es decir, $\phi = 1{,}0$

- **Contracción del paso θ:** Se puede mantener el mismo usado en la solución del IASVP, es decir, $\theta = 0{,}5$

- **Función para el decrecimiento suficiente ρ:** Como para este problema se intenta encontrar la solución ML (*maximum-likelihood*) no es conveniente que la función ρ alcance valores distintos de cero, pues de esa manera se podrían rechazar posibles soluciones. Por tanto $\rho \equiv 0$.

- **Conjunto de direcciones de búsqueda $\mathcal{D}$:** Se puede mantener el mismo conjunto de direcciones de coordenadas. Dado que la longitud del paso siempre será un múltiplo de l, se garantizará que el método buscará sólo entre puntos pertenecientes a la retícula.

4.4.3 Resultados experimentales

Para aplicarlos al problema de decodificación de señales en sistemas MIMO se escogieron los métodos de búsqueda directa gssSum, gssDPCyc y gssEM. Para los experimentos se escogieron constelaciones L-PAM tomando para L los valores 8, 16, 32 y 64.

Se escogieron problemas donde $n = m = 4$, es decir, problemas que se derivan de sistemas MIMO con 2 antenas transmisoras y dos antenas receptoras. Las matrices con las que se trabajó fueron obtenidas generando primeramente matrices complejas 2×2 y luego transformándolas al modelo real.

A toda señal transmitida se le añadió un ruido gaussiano blanco con media cero y varianza determinada por:

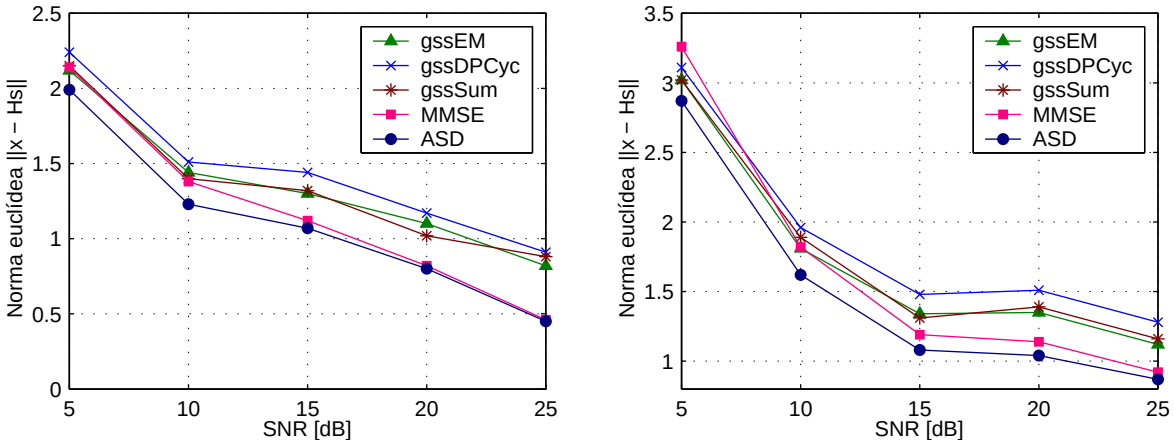
$$\sigma^2 = \frac{m\left(L^2 - 1\right)}{12\rho} \tag{4.16}$$

donde ρ es la relación señal a ruido (SNR) y para la cual se consideraron los valores de 5, 10, 15, 20 y 25. Para cada caso (n, m, L, ρ) se generaron 10 matrices, se decodificaron 10 señales por cada matriz, y finalmente se calculó el promedio de los valores mínimos alcanzados durante la búsqueda, así como el tiempo promedio de ejecución.

Los métodos de búsqueda directa fueron comparados con un método ML (*maximum-likelihood*) y con un método heurístico. El método ML escogido fue el método *Automatic Sphere-Decoding* (ASD) publicado en [112] y el método heurístico escogido fue el método MMSE publicado en [61]. En la figura 4.12 se muestran las gráficas con los valores mínimos obtenidos, es decir, con los valores de la distancia euclidiana $\|x - H\hat{s}\|$, donde $\hat{s}$ es el vector de símbolos decodificado.

Se puede observar que a medida que crece el tamaño de la constelación (y consecuentemente el problema se hace más complejo), la solución del método MMSE se aleja cada vez más de la solución ML (la cual es el óptimo global), mientras que en todos los casos las soluciones obtenidas por los métodos de búsqueda directa se mantienen a una misma distancia bastante cercana de la solución ML. De modo que ya en los casos en que $L = 32$ y $L = 64$ las soluciones de la búsqueda directa es mucho mejor que la del método heurístico.

Por otro lado, en la tabla 4.7 se muestra el promedio de puntos de la retícula explorados por cada método para encontrar el punto más cercano. Esta comparación es importante realizarla, porque ofrece una medida de cuántas veces el método realiza la operación $\|x - H\hat{s}\|$. Mientras menos se realiza dicha operación, menos tiempo de ejecución se emplea. Se comparan en este caso los métodos de búsqueda directa con el método ASD. Como es lógico, a medida que el problema se hace más complejo (a mayores valores de L y menores valores de ρ) todos los métodos exploran más puntos de la retícula. En el caso



(a) 8-PAM (b) 16-PAM

(c) 32-PAM (d) 64-PAM

Figura 4.12: Mínimos obtenidos en problemas de dimensión 4×4. Punto inicial para la búsqueda directa: aleatorio

método	$\rho = 5$	$\rho = 10$	$\rho = 15$	$\rho = 20$	$\rho = 25$	L
gssEM	74, 03	74, 68	79, 01	75, 35	79, 41	8
gssDPCyc	46, 45	46, 91	48, 25	47, 66	49, 13	
gssSum	60, 32	59, 01	61, 63	61, 47	63, 03	
ASD	93, 72	51, 72	48, 52	38, 12	35, 16	
gssEM	113, 33	113, 29	117, 56	114, 54	113, 73	16
gssDPCyc	71, 22	70, 59	70, 30	69, 68	71, 55	
gssSum	94, 13	89, 03	90, 48	91, 66	90, 70	
ASD	698, 28	173, 00	93, 16	95, 88	87, 88	
gssEM	155, 42	159, 83	163, 94	161, 76	163, 55	32
gssDPCyc	93, 55	93, 49	99, 42	95, 94	99, 50	
gssSum	123, 69	122, 52	122, 01	121, 96	124, 17	
ASD	8406, 12	1859, 24	238, 44	235, 24	188, 20	
gssEM	208, 86	190, 40	200, 26	205, 39	202, 34	64
gssDPCyc	126, 54	124, 51	123, 39	124, 93	122, 02	
gssSum	155, 31	161, 63	158, 40	162, 57	159, 95	
ASD	13908, 2	8967, 4	2121, 32	502, 66	374, 76	

Tabla 4.7: Promedio de puntos explorados en problemas de dimensión 4×4. Punto inicial para la búsqueda directa: aleatorio

de los métodos de búsqueda directa, influye poco el valor de la relación señal-a-ruido porque en esta primera prueba se ha tomado como punto inicial un punto aleatorio. Se puede observar que a medida que los problemas se hacen más complejos, los métodos de búsqueda directa exploran mucho menos puntos que el método ASD. Para $L = 32, 64$ han existido casos en que el ASD ha explorado hasta 100000 y 500000 puntos.

Comparando los tres métodos de búsqueda directa diseñados, en todos los casos el método gssDP-Cyc exploró menos nodos que los métodos gssSum y gssEM, sin embargo casi siempre estos métodos ofrecieron mayor precisión en la solución, sobre todo el método gssEM.

Si se comparan los tiempos de ejecución, el método heurístico MMSE supera ampliamente al resto de los métodos, como se puede ver en la figura 4.13. No obstante, los tiempos de ejecución de los métodos de búsqueda directa son bastante bajos, pues generalmente estuvieron por debajo de los 0,001 segundos.

En estos primeros experimentos siempre se tomó como punto inicial para la búsqueda directa un punto generado de manera aleatoria. De modo que la búsqueda directa, si se compara con el método MMSE, tiene la desventaja de trabajar sin conocimiento de la varianza con la que fue generado el ruido que se le añade a la señal emitida. Es por ello que también se hicieron pruebas tomando como

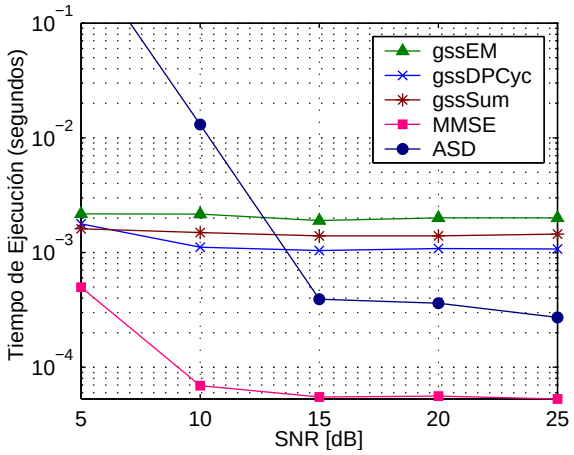
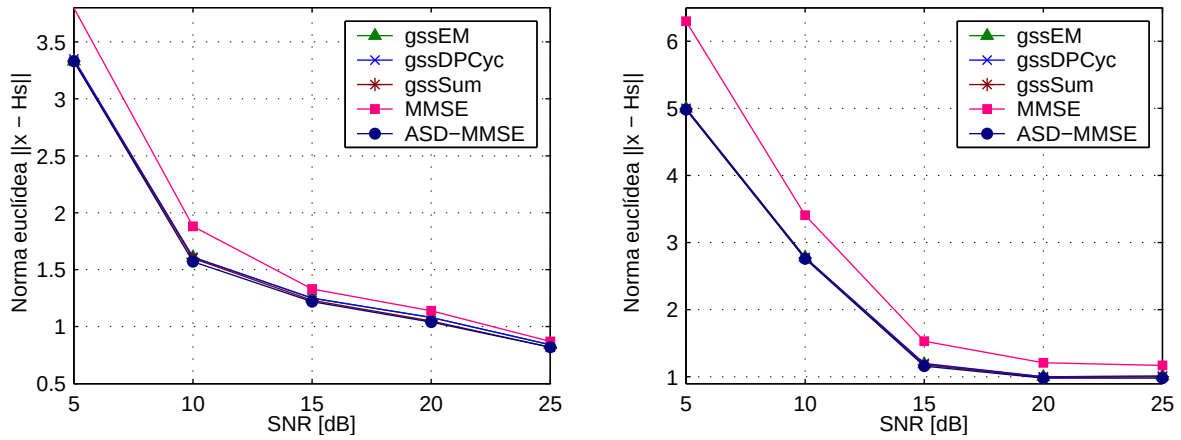


Figura 4.13: Gráfica SNR vs Tiempos de Ejecución para $L = 32$. Punto inicial para la búsqueda directa: aleatorio

punto de partida la solución obtenida por otro método heurístico, el *Zero-Forcing* (ZF). A su vez, al método ASD se le incorporó un radio de búsqueda para disminuir su promedio de nodos explorados. Se tomó como radio de búsqueda el valor $\|x - H\hat{s}_{MMSE}\|$ donde $\hat{s}_{MMSE}$ es la solución ofrecida por el método MMSE. En la figura 4.14 y en la tabla 4.8 se muestran los resultados de estos experimentos, esta vez realizados con $L = 16, 32, 64, 128$.

método	$\rho = 5$	$\rho = 10$	$\rho = 15$	$\rho = 20$	$\rho = 25$	L
gssEM	51, 72	44, 58	43, 32	42, 77	39, 02	16
gssDPCyc	38, 48	36, 62	36, 34	36, 13	35, 15	
gssSum	43, 54	40, 69	40, 03	39, 76	37, 45	
ASD-MMSE	53, 22	12, 56	15, 83	10, 11	8, 24	
gssEM	67, 81	61, 16	53, 63	50, 54	50, 65	32
gssDPCyc	51, 38	49, 33	46, 84	45, 53	45, 27	
gssSum	57, 22	54, 41	50, 35	48, 50	48, 02	
ASD-MMSE	195, 96	57, 33	17, 81	28, 62	10, 06	
gssEM	83, 82	72, 96	64, 98	65, 24	63, 18	64
gssDPCyc	63, 44	59, 91	57, 24	57, 48	57, 40	
gssSum	70, 28	64, 90	61, 78	61, 44	60, 88	
ASD-MMSE	1798, 84	560, 48	44, 03	17, 28	13, 55	
gssEM	102, 23	91, 97	83, 41	75, 19	74, 35	128
gssDPCyc	75, 35	73, 02	70, 28	68, 18	67, 89	
gssSum	86, 45	78, 48	75, 82	71, 25	71, 45	
ASD-MMSE	19917, 48	4009, 26	374, 31	28, 62	21, 10	

Tabla 4.8: Promedio de puntos explorados en problemas de dimensión 4×4. Punto inicial para la búsqueda directa: solución del método ZF



(a) 16-PAM

(b) 32-PAM

(c) 64-PAM

(d) 128-PAM

Figura 4.14: Mínimos obtenidos en problemas de dimensión 4×4. Punto inicial para la búsqueda directa: solución del método ZF

Como se esperaba, las soluciones de los métodos de búsqueda directa se acercaron mucho más, pues esta vez comienzan la búsqueda partiendo de un punto que es solución de un método heurístico. En la gran mayoría de los casos alcanzaron la solución ML. La tabla 4.8 refleja además que la búsqueda directa exploró menos puntos en la retícula. El método ASD también redujo considerablemente el campo de búsqueda al incorporar el radio de búsqueda tomado de la solución MMSE. Sin embargo, se mantiene la tendencia de que los métodos de búsqueda directa son mejores para problemas donde hay mucho ruido y el alfabeto de símbolos es mayor.

Teniendo en cuenta que ya en esta situación la solución obtenida por la búsqueda directa es casi similar a la obtenida por el método ML, cuyo tiempo de ejecución se deteriora notablemente cuando la relación señal-a-ruido aumenta, y que la búsqueda directa encuentra la solución en un tiempo considerablemente pequeño, se puede concluir afirmando que los métodos de búsqueda directa son métodos efectivos y eficientes para la decodificación de señales en los sistemas MIMO. Particularmente se debe destacar el método gssDPCyc por ser el de menor promedio de puntos explorados en todos los casos.

5
Métodos Maximum-Likelihood

En este capítulo se describe el trabajo realizado con el objetivo de obtener mejoras computacionales en los métodos ML basados en búsqueda en árbol, específicamente en los métodos *Sphere-Decoding* y *Automatic Sphere-Decoding* vistos en el capítulo 3. Se aborda el uso de la descomposición de valores singulares en la selección de radios de búsqueda en el SD, así como en el preprocesado y ordenación previa.

5.1 Uso de la descomposición de valores singulares en el Sphere-Decoding

La descomposición de valores singulares (SVD - *Singular Value Decomposition*) es una técnica poderosa en el análisis y la computación matricial. Su origen se encuentra en el intento de los geómetras del siglo XIX por conseguir la reducción de una forma cuadrática a una forma diagonal mediante cambios de bases ortogonales.

La utilización de la SVD en el cálculo matricial reduce los errores numéricos que puedan surgir en determinadas operaciones matriciales. Adicionalmente, la SVD provee información acerca de la estructura geométrica de una matriz. Para hacer una transformación de un espacio vectorial en otro, generalmente se usa una matriz. La SVD de dicha matriz refleja los cambios geométricos que sucedieron en dicha transformación. Es por ello que la SVD tiene un amplio uso en varias aplicaciones, desde

los problema de mínimos cuadrados hasta los sistemas de ecuaciones lineales, muchas de las cuales explotan algunas de las propiedades de la SVD, como es la relación que tiene con el rango de la matriz y la posibilidad de lograr aproximaciones de matrices de un determinado rango. Una completa cobertura sobre la SVD se puede ver en [54], mientras que en [35] y [129] se puede consultar muchas de las aplicaciones de la SVD en el procesamiento de señales.

La descomposición de valores singulares de una matriz $A \in \mathbb{R}^{n \times m}$ es cualquier factorización de la forma:

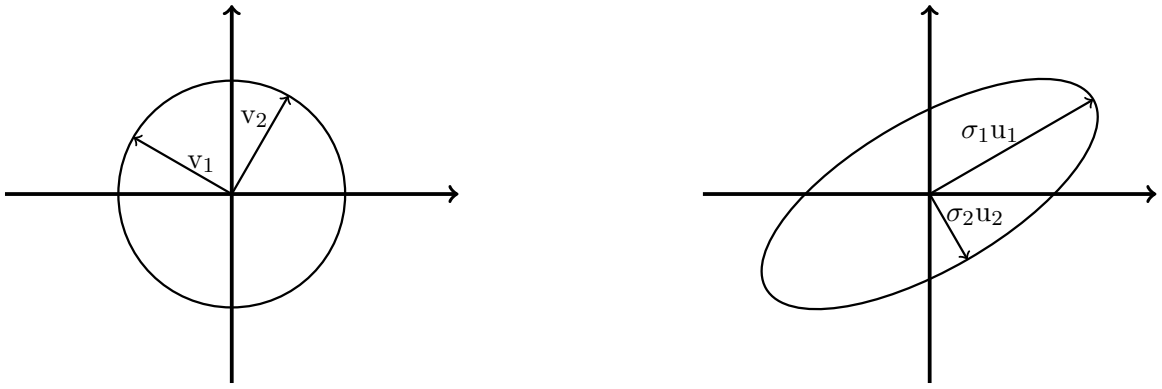
$$A = U \Sigma V^T \tag{5.1}$$

donde $U \in \mathbb{R}^{n \times n}$ y $V \in \mathbb{R}^{m \times m}$ son dos matrices ortogonales, y $\Sigma = diag\left(\sigma_1, \ldots, \sigma_p\right),\ p = \text{mín}\left(n, m\right)$, es una matriz diagonal, donde $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$ [54]. Los valores $\sigma_1, \ldots, \sigma_p$ son los valores singulares de H, mientras que los vectores de U y V se denominan vectores singulares por la izquierda y por la derecha respectivamente.

Como se mencionó anteriormente, la SVD ofrece información sobre los cambios que ocurren cuando se realiza una multiplicación de una matriz con un vector. Si se tiene un conjunto de vectores x de n componentes, todos de longitud unitaria ($\|x\|_2 = 1$), estos forman una hiper-esfera unitaria en el espacio n-dimensional. Si dichos vectores se multiplican por una matriz $A_{m \times n}$, entonces se obtiene un conjunto de vectores m-dimensionales de longitudes variadas. Geométricamente este conjunto de vectores forman un hiper-elipsoide k-dimensional embebido en un espacio m-dimensional, donde k es la cantidad de valores singulares de A distintos de cero.

La figura 5.1 representa esta situación para un espacio 2D. Las longitudes de los radios (o semiejes) del hiper-elipsoide son los valores singulares de la matriz A. Por su parte, los vectores singulares $\mathrm{u}_1, \mathrm{u}_2, \ldots, \mathrm{u}_n$ (las columnas de U), determinan la dirección de los semiejes del hiper-elipsoide, y los vectores singulares $\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_m$ son sus preimágenes.

A continuación, en esta sección, se exponen algunas propuestas de mejoras en el funcionamiento del SD en las cuales la SVD tiene un papel protagónico.



(a) $\{x\}$ (b) $\{Ax\}$

Figura 5.1: Transformación de una circunferencia mediante una matriz A

5.1.1 Selección del radio inicial para el método Sphere-Decoding basada en la SVD

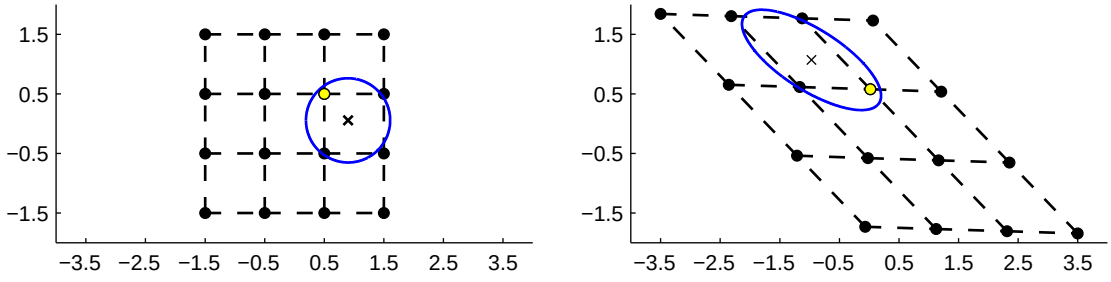
Volvamos al problema de la decodificación de señales en sistemas MIMO, que en efecto es una instancia particular del problema general CVP definido en la sección 3.1. En la subsección 3.4 se explicó la idea del método *Sphere-Decoding*, que consiste en acotar el espacio de búsqueda de la solución del problema CVP a una hiper-esfera con centro en el vector recibido por el sistema. En la propia subsección 3.4 se presentó un estado del arte sobre el tema de la selección de radios iniciales en el SD, cuestión que influye significativamente en la complejidad de la búsqueda del SD.

En este trabajo se propone una sucesión de radios de búsqueda para el SD basada en la descomposición de valores singulares. La idea que se propone parte del análisis de la interpretación geométrica de la multiplicación de una matriz H por un vector w que se comentó anteriormente. Haciendo una generalización, si se tiene un conjunto de vectores $w \in \mathbb{R}^m$ que pertenezcan a cierta hiper-esfera de radio r y centro en un punto c, entonces los vectores Hw formarán un hiper-elipsoide con centro en Hc y semi-ejes con longitudes $r\sigma_1, r\sigma_2, \ldots, r\sigma_p$, donde $\sigma_1, \sigma_2, \ldots, \sigma_p$ son los valores singulares de H.

Por otro lado, supongamos que se tiene un punto $w \in \mathbb{R}^m$ y se quiere saber cuál de los puntos de la retícula $\mathcal{L}(I) = \{Is : s \in \mathbb{Z}^m\}$, donde I es la matriz identidad, es el más cercano a w. No es difícil demostrar que el punto de $\mathcal{L}(I)$ más cercano a w se encuentra dentro de una hiper-esfera con centro en w y radio $\sqrt{m}/2$ (Ver Figura 5.2(a)). Sea w' el punto más cercano a w en la retícula cuadrada

$\mathcal{L}(I)$.

Teniendo en cuenta la interpretación geométrica de la multiplicación de una matriz por una hiper-esfera, se puede afirmar que el hiper-elipsoide con centro en Hw y semi-ejes con longitudes $\frac{\sqrt{m}}{2}\sigma_1, \frac{\sqrt{m}}{2}\sigma_2, \ldots, \frac{\sqrt{m}}{2}\sigma_p$ y con direcciones determinadas por los vectores singulares $\mathrm{u}_1, \mathrm{u}_2, \ldots, \mathrm{u}_n$, tendrá en su interior al menos un punto de la retícula $\mathcal{L}(H) = \{Hs : s \in \mathbb{Z}^m\}$. Entre los puntos de la retícula $\mathcal{L}(H)$ que estarán dentro del hiper-elipsoide, estará obviamente el punto Hw'. (Ver Figura 5.2(b))



(a) Retícula $\mathcal{L}(I)$ (b) Retícula $\mathcal{L}(H)$

Figura 5.2: En (a) el punto w se denota por el símbolo $\times$ y el punto w' se muestra en amarillo. En (b) el punto Hw se denota por el símbolo $\times$ y el punto Hw' se muestra en amarillo

Retornando al problema

$$\min_{s \in \mathcal{A}_L{}^m} \|x - Hs\|^2$$

lo anteriormente afirmado significa que el punto más cercano a x en la retícula $\mathcal{L}(H)$ se encuentra dentro de una de las hiper-esferas que tienen centro en x y radios iguales a

$$\frac{\sqrt{m}}{2}\sigma_1, \frac{\sqrt{m}}{2}\sigma_2, \ldots, \frac{\sqrt{m}}{2}\sigma_p \tag{5.2}$$

donde $\sigma_1, \sigma_2, \ldots, \sigma_p$ son los valores singulares de H.

Luego, el método que se propone para seleccionar los radios de búsqueda en el método *sphere-decoding* es que estos radios sean los valores obtenidos según (5.2). Se seleccionarían los radios en

orden creciente, teniendo en cuenta que:

$$\frac{\sqrt{m}}{2}\sigma_1 \geq \frac{\sqrt{m}}{2}\sigma_2 \geq \ldots \geq \frac{\sqrt{m}}{2}\sigma_p \tag{5.3}$$

por lo que se realizaría una primera búsqueda dentro de la hiper-esfera con centro en x y radio $r = \frac{\sqrt{m}}{2}\sigma_p$. Si no se encuentra ninguna solución, entonces se incrementa el radio haciendo $r = \frac{\sqrt{m}}{2}\sigma_{p-1}$, y así sucesivamente hasta llegar a $r = \frac{\sqrt{m}}{2}\sigma_1$ de ser necesario.

En caso que la constelación $\mathcal{A}_L$ esté compuesta por símbolos $\alpha_1, \alpha_2, \ldots, \alpha_L$ tales que $\alpha_{i+1} - \alpha_i = l$ para todo i, $1 \leq i \leq L-1$, la sucesión de radios sería

$$\frac{\sqrt{m}}{2}\sigma_1 l, \frac{\sqrt{m}}{2}\sigma_2 l, \ldots, \frac{\sqrt{m}}{2}\sigma_p l \tag{5.4}$$

Un inconveniente que presenta esta selección de radios se puede manifestar cuando el vector s pertenece a un subconjunto finito de una retícula, como es habitual en los problemas (3.8), pues en estos casos puede pasar que el vector x esté muy fuera de la retícula $\mathcal{L}(H)$ de modo que el hiper-elipsoide con centro en x esté también totalmente fuera de la retícula y no contenga ningún punto. Esto puede suceder sobre todo si el conjunto discreto $\mathcal{A}$ es muy pequeño y existe muy poca separación entre sus elementos. En estos casos entonces se aplicaría el procedimiento general de incrementar el radio de búsqueda hasta que la hiper-esfera contenga al menos un punto de la retícula. Claramente, en retículas infinitas este inconveniente no sucede.

Esta sucesión de radios de búsqueda garantiza que se encuentre una solución al problema (3.8) pues, como se vio anteriormente, al menos en la hiper-esfera con radio $\frac{\sqrt{m}}{2}\sigma_1 l$ se encontrará la solución.

Se estima que esta sucesión de radios basada en los valores singulares de H disminuya el número de nodos del árbol de búsqueda en el *sphere-decoding*, pues en muchos casos los radios iniciales $\frac{\sqrt{m}}{2}\sigma_p l, \frac{\sqrt{m}}{2}\sigma_{p-1} l, \ldots$ son menores que el antes citado $r^2 = \alpha n \sigma^2$ o que la distancia de x a algún punto solución de un método heurístico, sobre todo si los elementos de la matriz H siguen una distribución normal con media 0 y varianza 1, en los cuales los valores singulares son bastante pequeños.

Resultados experimentales

Para comprobar la efectividad de la propuesta desarrollada, se realizó una comparación con otras técnicas de selección de radios para el SD mencionadas en la sub-sección 3.4.4. Por tanto se implementó el método *Sphere-Decoding* con diferentes opciones de selección de radio inicial.

El primer método selecciona el radio según la varianza del ruido como en [62], y fue nombrado SD-alpha. Las variantes de los métodos *Sphere-Decoding* en las cuales los radios se seleccionan según el resultado arrojado por los métodos heurísticos ZF, ZF-SIC y MMSE-OSIC, se identifican con los nombres SD-ZF, SD-ZFSIC y SD-MMSE. Estos métodos heurísticos fueron implementados siguiendo los algoritmos descritos en la sección 3.3. Finalmente la variante que usa la técnica de selección de radios mediante la SVD fue nombrado SD-SVD.

Los experimentos se realizaron con constelaciones L-PAM (ver sub-sección 3.1.2), tomando para L los valores de 2, 4, 8 y 16.

Los métodos *Sphere-Decoding* se probaron en problemas donde $n = m = 4$ y $n = m = 8$, en los cuales las matrices reales H se generaron de manera aleatoria siguiendo una distribución Gaussiana con media cero y varianza 1. Las matrices con las que se trabajó fueron obtenidas generando primeramente matrices complejas 2×2 y 4×4 respectivamente, y luego transformándolas al modelo real. Es por ello que las matrices H tienen la estructura

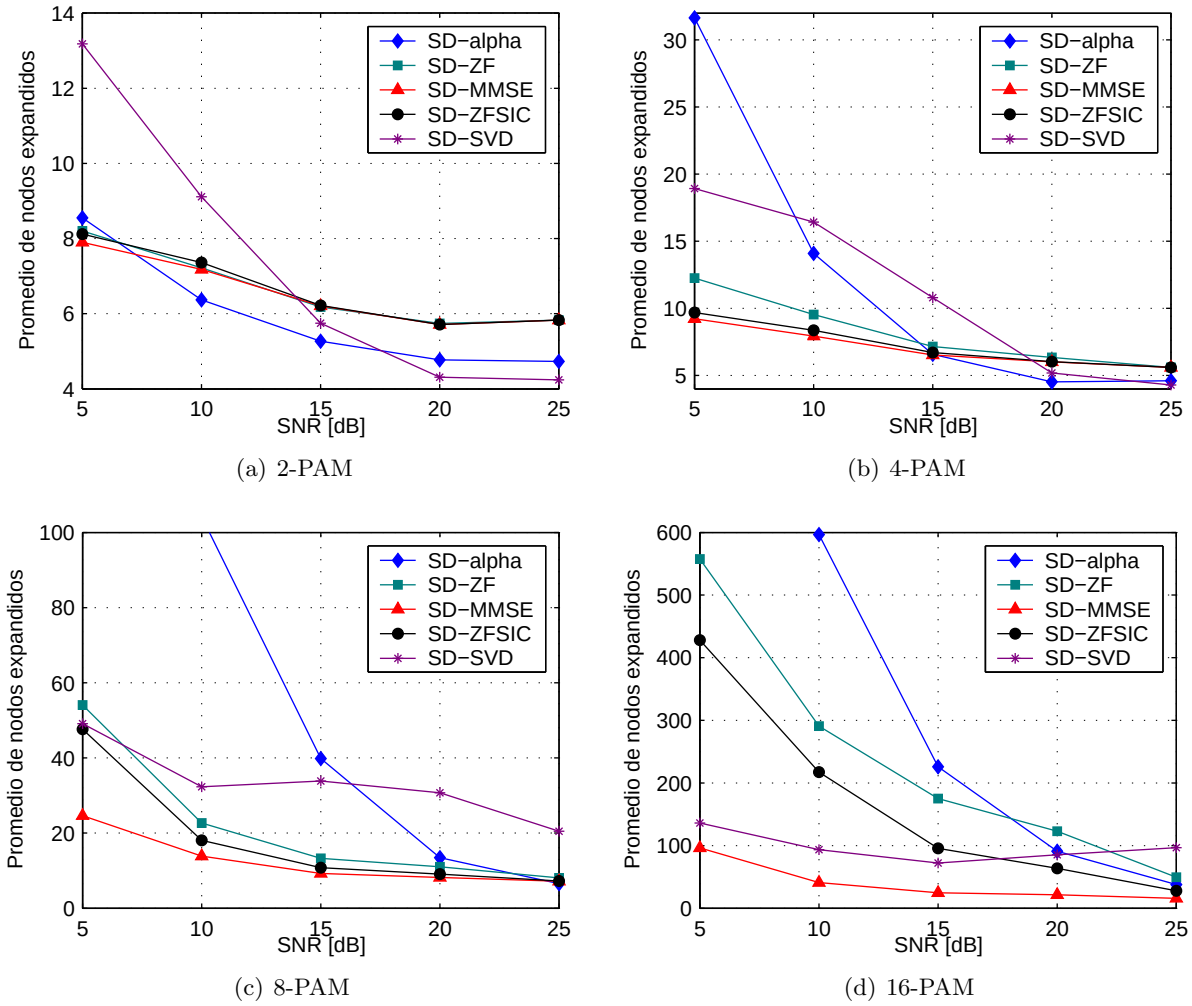
$$H = \begin{bmatrix} \mathrm{Re}\,(\mathbf{H}) & \mathrm{Im}\,(\mathbf{H}) \\ -\mathrm{Im}\,(\mathbf{H}) & \mathrm{Re}\,(\mathbf{H}) \end{bmatrix} \tag{5.5}$$

A toda señal transmitida se le añadió un ruido gaussiano blanco con media cero y varianza determinada por:

$$\sigma^2 = \frac{m\left(L^2 - 1\right)}{12\rho} \tag{5.6}$$

donde ρ es la relación señal a ruido (SNR) y para la cual se consideraron los valores de 5, 10, 15, 20, 25 y 30. Para cada caso (n, m, L, ρ) se generaron 10 matrices, se decodificaron 20 señales por cada matriz y se calculó el promedio de nodos visitados durante la búsqueda, así como el tiempo promedio de ejecución.

En la figura 5.3 se muestra el promedio de nodos examinados (o expandidos) por cada uno de los métodos, en problemas donde $n = m = 4$ y las constelaciones son 2-PAM (Fig. 5.3(a)), 4-PAM (Fig. 5.3(b)), 8-PAM (Fig. 5.3(c)) y 16-PAM (Fig. 5.3(d)).



(a) 2-PAM

(b) 4-PAM

(c) 8-PAM

(d) 16-PAM

Figura 5.3: Comparación entre los 5 métodos según el promedio de nodos visitados en problemas 4×4

Se puede notar que para valores pequeños de L y valores pequeños de SNR (el vector ruido v se genera por tanto con una varianza mayor), el número de nodos examinados por el método SD-SVD es considerable comparado con el resto de los métodos. Esto es debido a la desventaja mencionada anteriormente, pues en estos casos la probabilidad de que el hiper-elipsoide con centro en x y semi-ejes como en (5.4) esté completamente fuera de la retícula $\mathcal{L}(H)$ es mucho mayor.

También se puede ver que cuando el tamaño de la constelación aumenta, el rendimiento del método SD-SVD se acerca al del método SD-MMSE, que fue el mejor porque en la mayoría de los casos examinó el menor número de nodos. Cuando el tamaño de la constelación aumenta, los métodos SD-alpha, SD-ZF y SD-ZFSIC visitan muchos más nodos comparados con SD-SVD y SD-MMSE.

Pasando a comparar los métodos SD-MMSE y SD-SVD, en la figura 5.4 se muestra el promedio de nodos visitados por los dos métodos, en problemas donde $n = m = 4$ y las constelaciones son 32-PAM, 64-PAM, 128-PAM y 256-PAM. Se puede observar, ahora en estos problemas un poco más complejos, que a medida que crece el número de símbolos de la constelación el rendimiento del SD-MMSE encuentra la solución visitando un número mucho mayor que el SD-SVD, de modo que el rendimiento del SD-SVD a partir de $L = 64$ es significativamente mejor.

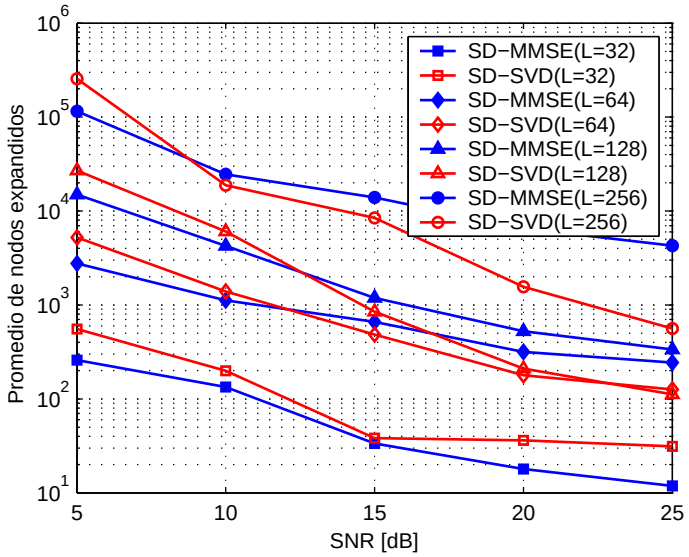
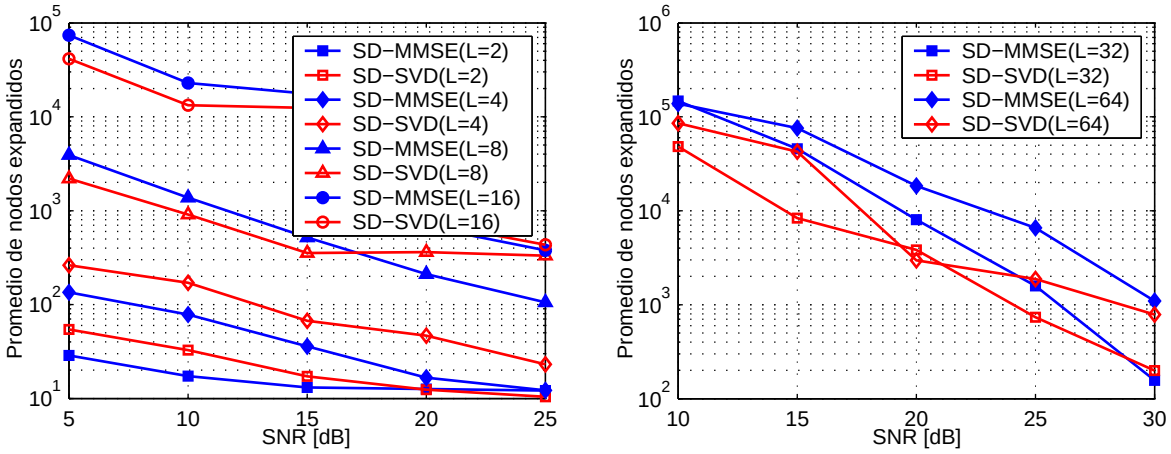


Figura 5.4: Comparación entre SD-MMSE y SD-SVD según el promedio de nodos visitados en problemas $4 \times 4, L = 32, 64, 12, 256$

También se puede ver en la figura 5.5 el promedio de nodos visitados por los métodos SD-SVD y SD-MMSE, en problemas donde $n = m = 8$ y las constelaciones son 2-PAM, 4-PAM, 8-PAM y 16-PAM (Fig. 5.5(a)), además de 32-PAM y 64-PAM (Fig. 5.5(b)). De este modo se puede hacer una comparación entre ambos métodos en problemas que son computacionalmente un poco más complejos. Vemos que para estos problemas el método SD-SVD se comporta claramente mejor a partir de $L = 16$. Nótese que las gráficas están en escala logarítmica, por lo que las diferencias que reflejan las gráficas

realmente son mucho mayores.



(a) $L = 2, 4, 8, 16$ (b) $L = 32, 64$

Figura 5.5: Comparación entre SD-MMSE y SD-SVD según el promedio de nodos visitados en problemas 8×8

La variante propuesta en este trabajo podría suponer un costo computacional adicional al método *Sphere-Decoding* pues es necesario determinar los valores singulares de la matriz generadora. En la siguiente figura, se muestra una comparación de los tiempos de ejecución del SD-MMSE y SD-SVD para los problemas con $n = m = 4$ y $L = 32, 64, 128, 256$. En ambos casos se ha incluido en el tiempo de ejecución el tiempo empleado por el método MMSE y el tiempo empleado para calcular la SVD respectivamente.

Las figuras 5.4 y 5.6 muestran una clara correspondencia entre los tiempos de ejecución y los nodos visitados en cada caso. Esto muestra que el peso computacional de los métodos *Sphere-Decoding* es la búsqueda de la solución en el árbol que genera, más allá del preprocesado que se pueda realizar. Por lo que el cálculo de la SVD de la matriz no supone una complejidad temporal mayor al método, además de que en la práctica para matrices 4×4 y 8×8 apenas se nota la demora el cálculo de los valores singulares.

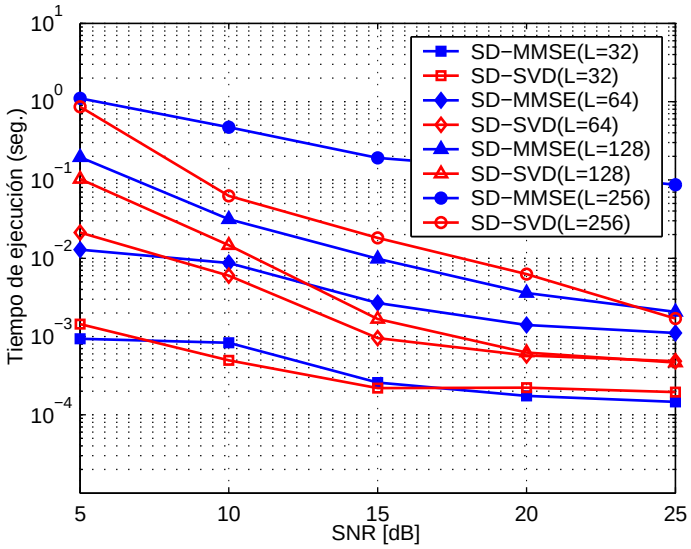


Figura 5.6: Comparación entre SD-MMSE y SD-SVD según el tiempo de ejecución en problemas $4 \times 4, L = 32, 64, 128, 256$

5.2 Sphere-Decoding Automático con radio inicial

Si bien es cierto que el método ASD propuesto en [112] encuentra la solución del problema visitando el mínimo de nodos posibles, es cierto también que en cada ramificación se generan tantos nodos como cantidad de símbolos tiene la constelación $\mathcal{A}_L$, es decir, el máximo de nodos posibles. Esto puede provocar que el costo de almacenamiento que requiere el algoritmo aumente exponencialmente a medida que aumente también las dimensiones del problema y el número de símbolos del alfabeto. También aumentaría el tiempo de ejecución del algoritmo, pues a diferencia de la pila, donde las operaciones de inserción y extracción tienen complejidad $O(1)$, en un *heap* la complejidad de las operaciones de inserción y extracción tienen complejidad $O(\log n)$ donde n es la cantidad de elementos de la estructura.

Por tanto, como parte de esta tesis, se propone considerar en el método ASD la información de un radio inicial, que permita hacer podas en la ramificación de los nodos, como mismo lo hace el SD. Ello disminuiría considerablemente el número de nodos generados, y con ello el costo de almacenamiento y el tiempo de ejecución del algoritmo. Claramente el radio inicial debe garantizar la existencia de al menos un punto de la retícula dentro de la hiper-esfera, pues de no ser así, debería aumentarse el radio

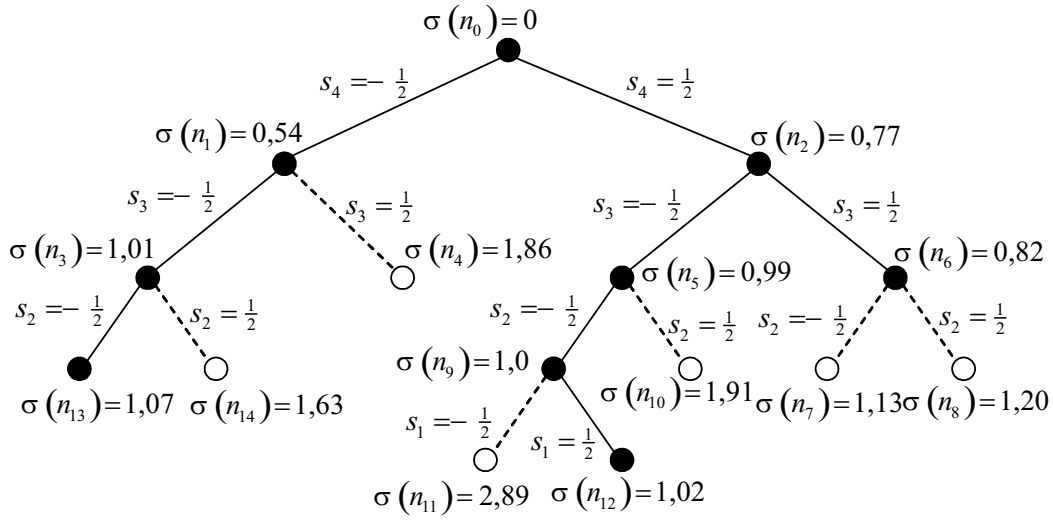


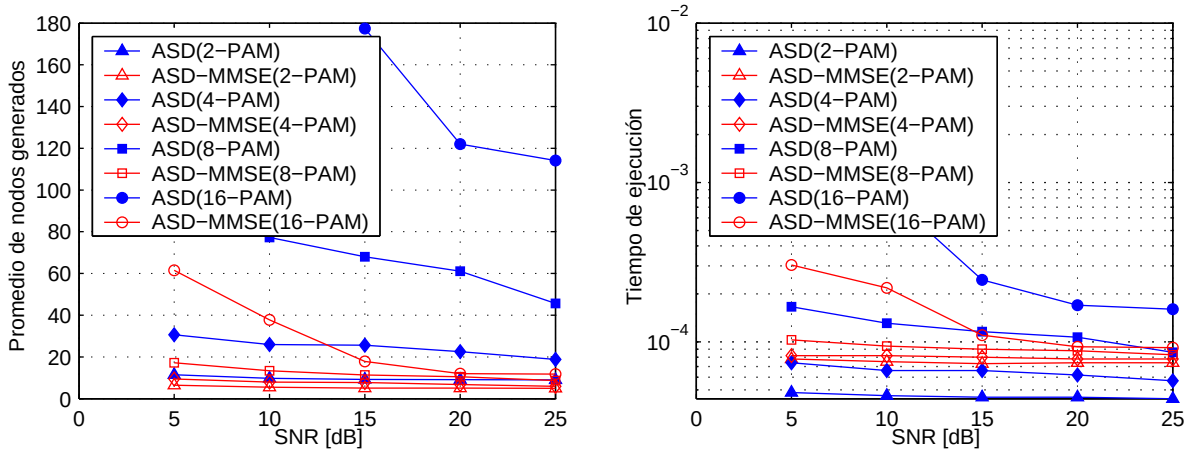
Figura 5.7: Árbol construido por el ASD. Los nodos en blanco y las ramas discontinuas no se generarían en caso de usarse un radio inicial para la búsqueda

y repetir el proceso de búsqueda, y consecuentemente ya no se lograría una búsqueda con el mínimo número de nodos visitados. La inclusión de un radio inicial que garantice la existencia de un punto de la retícula no altera el número de nodos visitados por el algoritmo ASD pues, como se explicó en la sección anterior, lo que realmente garantiza que el ASD alcance la solución luego de expandir el número mínimo de nodos es el tipo de recorrido del árbol, determinado por la estructura *heap*.

En la siguiente figura se muestra el árbol de búsqueda que construiría el método ASD en la solución de un problema donde la matriz de canal es 4×4, la constelación es L-PAM con $L = 2$. En blanco se señalan los nodos y en líneas discontinuas las ramas que no se generarían si se usara en el método ASD un radio inicial. Para este ejemplo se tomó como radio la distancia entre el punto x y el punto que arrojó como solución el método heurístico MMSE.

5.2.1 Resultados experimentales

Se implementó el método ASD descrito en la sub-sección 3.4.5 y se implementó una variante del ASD donde se incluye un radio de búsqueda tal y como se describió anteriormente. El radio de búsqueda escogido fue la distancia entre el punto x y el punto que arroja como solución el método heurístico MMSE. Por tal motivo a esta nueva versión le hemos llamado ASD-MMSE. La versión



(a) Según promedio de nodos generados

(b) Según tiempo de ejecución

Figura 5.8: Comparación entre los métodos ASD y ASD-MMSE

MMSE implementada es la variante OSIC publicada en [61].

Para los experimentos, se escogieron constelaciones L-PAM tomando para L los valores 2, 4, 8 y 16. Los métodos se probaron en problemas donde $n = m = 4$. Las matrices con las que se trabajó fueron obtenidas generando primeramente matrices complejas y luego transformándolas al modelo real, tal y como se describió en la sub-sección 3.1.2.

A toda señal transmitida se le añadió un ruido gaussiano blanco con media cero y varianza determinada por:

$$\sigma^2 = \frac{m\left(L^2 - 1\right)}{12\rho} \tag{5.7}$$

donde ρ es la relación señal a ruido (SNR) y para la cual se consideraron los valores de 5, 10, 15, 20 y 25. Para cada caso (n, m, L, ρ) se generaron 10 matrices, se decodificaron 20 señales por cada matriz y se calculó el promedio de nodos generados durante la búsqueda (que equivale a la cantidad total de nodos en el árbol), así como el tiempo promedio de ejecución.

En la figura 5.8(a) se muestran los promedios de nodos generados por ambos métodos en problemas donde $n = m = 4$ y las constelaciones son 2-PAM, 4-PAM, 8-PAM y 16-PAM, y en la figura 5.8(b) se muestran los tiempos de ejecución de ambos métodos en tales problemas.

Como era de esperarse, en todos los casos el método ASD-MMSE es mejor que el ASD en cuanto a

promedio de nodos generados, es decir, en todos los casos el ASD-MMSE genera mucho menos nodos que el ASD. Comparando ambos métodos en cuanto tiempo de ejecución, se puede apreciar que en problemas donde el tamaño de la constelación es pequeño el método ASD se comporta mejor que el ASD-MMSE, como es en los casos donde $L = 2$ y $L = 4$. La razón por la cual el ASD-MMSE tiene tiempos superiores a pesar de generar menos nodos se debe a los cálculos adicionales que debe realizar al ejecutar el método MMSE, y a que la diferencia de nodos generados entre ambos métodos no es significativa. Sin embargo, en los casos donde $L = 8$ y $L = 16$ el método ASD-MMSE obtiene mucho mejores tiempos de ejecución, esto es debido a que la diferencia de nodos generados comparado con el ASD es bien notable.

6

Paralelización de Métodos Maximum-Likelihood

En este capítulo se describen las versiones paralelas diseñadas e implementadas de los métodos *Sphere-Decoding* y *Automatic Sphere-Decoding*, para distintos entornos paralelos.

6.1 Paralelización del Sphere-Decoding

En esta sección se expondrán algunas variantes de paralelización del método *Sphere-Decoding* para distintos entornos paralelos: con memoria distribuida, con memoria compartida, y en entornos que se componen de varios multiprocesadores comunicados mediante una red de interconexión. Un estudio sobre la paralelización de métodos *Branch-and-Bound* se puede ver en [55], donde se enuncian los principios básicos que se deben considerar para distribuir la carga de trabajo en los procesadores. En esta sección se expondrán cómo se han adaptado estos esquemas generales a la paralelización del método *Sphere-Decoding*, aprovechando información específica del problema.

6.1.1 Paralelización del Sphere-Decoding en un entorno de memoria distribuida

De cara a la paralelización del *Sphere-Decoding* en un entorno de paso de mensajes, el problema más complicado radicaría en encontrar una distribución de los nodos de la estructura, de modo tal que el desequilibrio de la carga de trabajo de los procesadores sea el menor posible.

En el algoritmo paralelo inicialmente el procesador *root* crea un nodo inicial y a partir de este nodo comienza una ejecución secuencial del *Sphere-Decoding* hasta que en la estructura hayan suficientes nodos para ser distribuidos al resto de los procesadores (al menos uno por procesador). Luego, los nodos de la estructura formada por el procesador *root* son distribuidos de manera cíclica al resto de los procesadores, por lo que cada procesador tendría su propia estructura. Se propone una distribución cíclica para evitar que a un procesador le lleguen sólo nodos de los últimos niveles (y probablemente tendría poco trabajo) y a otro procesador le lleguen sólo nodos de los primeros niveles (y probablemente tendría mucho trabajo). Ver Figura 6.1.

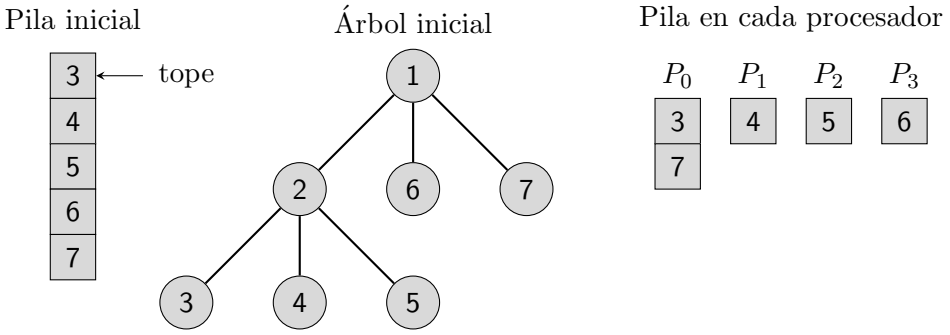


Figura 6.1: Expansión inicial y distribución de la estructura por los procesadores

Luego, en cada iteración, cada procesador expande un número fijo de nodos de su estructura. Después que cada procesador expande el número prefijado de nodos, habrá un punto de sincronización para que todos sepan el estado de la pila de cada uno (si está vacía o no). Es entonces cuando los procesadores cuyas estructuras quedaron vacías escogen aleatoriamente un procesador para enviarle un mensaje de solicitud de nodos (mensaje *request*). Si recibe una respuesta negativa (mensaje *reject*) entonces escogería otro procesador, así hasta que reciba un mensaje con respuesta positiva (mensaje *accept*) o haya recibido respuesta negativa de todos. Siempre luego de la expansión del número prefijado de nodos, los procesadores deben chequear si en su cola de mensajes tienen algún mensaje *request*. En ese caso responden *reject* si su estructura está vacía o *accept* si tienen nodos para enviar. Siempre que un procesador va a enviar nodos a otro procesador cuya estructura está vacía, envía un por ciento de los nodos de su estructura, escogidos de forma intercalada simulando una distribución cíclica entre dos procesadores.

Al final de cada iteración los procesadores deben reunir las soluciones que han encontrado para determinar la mejor, y usarla en la comparación con las soluciones que encontrarían en iteraciones posteriores. También reúnen el estado de cada procesador, para determinar la terminación del algoritmo, que tendría lugar cuando la estructura de cada uno de ellos esté vacía.

Una cuestión importante es la elección para cada iteración del número de nodos a expandir por cada procesador. Para evitar desequilibrios de carga este número debe ser el mismo en cada procesador. Este número de nodos a expandir no debe ser muy pequeño comparado con el tamaño de las estructuras, porque incrementaría las comunicaciones, y tampoco puede ser muy grande porque podría provocar que algunos procesadores terminen su búsqueda y deban esperar mucho tiempo porque otros procesadores terminen.

En este trabajo se propone que la estimación del número de nodos a expandir en una iteración se realice al inicio de cada iteración: cada procesador realiza un estimado de nodos que podría expandir, reúnen estos estimados en todos los procesadores (*AllGather*) y finalmente el número de nodos a expandir en la siguiente iteración sería la mediana de las estimaciones reunidas.

Se propone además que cada procesador realice su respectiva estimación teniendo en cuenta el número de nodos de su estructura, los niveles de cada uno de los nodos, y la cantidad de símbolos del alfabeto $\mathcal{A}$. Suponiendo que la estructura tiene l_1 nodos con nivel $m-1$ y l_2 nodos con nivel menor que $m-1$, entonces el estimado de nodos que el procesador podría expandir en la siguiente iteración es

$$l_1 L + l_2 L^2 \tag{6.1}$$

En la versión paralela del *Sphere-Decoding* implementada en este trabajo, se incluyeron ambas propuestas.

En el algoritmo 19 se muestran los pasos del algoritmo paralelo para memoria distribuida. La invocación `BestBoundExpand(EGlobal, minNodes, solution, solutionValue)` ejecuta los pasos del algoritmo 7, descrito en la sección 3.4, hasta que la estructura `EGLobal` contenga al menos `minNodes` nodos. Deben pasarse por parámetros las variables `solution` y `solutionValue` pues puede suceder que sea necesario actualizarlas. La invocación `BestFixExpand(fixedNodes, ELocal, solution,`

`solucionValue)` realiza la misma ejecución, lo que en este caso expande exactamente `fixedNodes` de la estructura `ELocal`. Por otro lado, la rutina `EstimateFixedNodes` realiza la estimación de nodos a expandir en la iteración, siguiendo la propuesta descrita. La rutina `LoadBalance` se especifica a continuación en el algoritmo 20

6.1.2 Paralelización del Sphere-Decoding en un entorno de memoria compartida

Una de las desventajas de la paralelización en memoria distribuida es, como se vio anteriormente, que es necesario fijar un número de nodos a expandir en cada iteración, para luego chequear el estado de la estructura de cada procesador (si está vacía o no). Esta cantidad de nodos a expandir debe ser cuidadosamente estimada, pues si se estima una cantidad pequeña puede provocar más sincronizaciones entre los procesos, o sea, más tiempo de comunicaciones, mientras que si se estima una cantidad muy grande puede provocar mucho desequilibrio en la carga de trabajo. En un entorno de memoria compartida, este problema no existiría.

Para un entorno paralelo con memoria compartida, una primera idea de algoritmo paralelo es que compartan toda la estructura, de modo que en cada iteración cada procesador extrae un nodo de la estructura, genera sus nodos hijos y adiciona estos a la estructura. De esta forma el algoritmo terminaría cuando la estructura está vacía.

El problema de que todos los procesadores trabajen sobre una misma estructura en común es que las operaciones de extracción y adición en la estructura deben realizarse dentro de una sección crítica, es decir, en un mismo instante de tiempo sólo un procesador puede adicionar o extraer de la estructura. Ello implicaría que el algoritmo apenas ganaría en velocidad. De aquí que cada procesador debe tener su propia estructura local.

El algoritmo que se propone por tanto es similar al descrito en la sección 6.1.1, pero la diferencia radicaría en que existiría una variable global que indicaría si algún procesador tiene su estructura vacía. Esta variable sería actualizada (dentro de una sección crítica) por el primer procesador que quede sin trabajo, y luego por todos los demás cuando se haga una nueva distribución de los nodos. De esta forma, todos los procesadores realizarían su búsqueda sobre los nodos de sus estructuras hasta

Algoritmo 19 Algoritmo SD paralelo para memoria distribuida

```
1: En Paralelo : Para pr = 0, 1, ..., p − 1
2: ELocal ← CreateStruct(''STACK'') {Cada procesador crea su pila ELocal}
3: si pr = root entonces
4:   {El procesador root expandirá algunos nodos hasta que sean suficientes para distribuirlos}
5:   EGlobal ← CreateStruct(''STACK'') {Se crea la estructura EGlobal}
6:   minNodes ← p {Al menos p nodos deben haber en EGlobal}
7:   node ← CreateInitialNode() {Se crea el nodo inicial}
8:   Add(EGlobal, node) {Se adiciona a EGlobal}
9:   BestBoundExpand(EGlobal, minNodes, solution, solutionValue)
10: fin si
11: CyclicScatterStruct(EGlobal, ELocal, root) {Distribución cíclica de EGlobal}
12: Broadcast(solutionValue, root) {Distribución del valor solución hasta ahora obtenido}
13: terminate ← false
14: mientras terminate = false hacer
15:   fixedNodes ← EstimateFixedNodes(ELocal) {Estimación de los nodos a expandir}
16:   AllGather(fixedNodes, allFixedNodes)
17:   fixedNodes ← median(allFixedNodes) {La cantidad de nodos a expandir será la mediana de
      todas las estimaciones}
18:   BestFixExpand(fixedNodes, ELocal, solution, solucionValue)
19:   AllReduceMin(solutionValue, procmin) {Se actualiza solutionValue. La variable procmin al-
      macena el número del procesador que tiene la solución correspondiente}
20:   AllGather(Size(ELocal), sizes)
21:   LoadBalance(ELocal, sizes, nIDLE){Se balancea la carga de trabajo}
22:   si nIDLE = p entonces
23:     terminate ← true {Si todas las estructuras están vacías, entonces termina el algoritmo}
24:   fin si
25: fin mientras
26: si pr = procmin entonces
27:   si pr ≠ root entonces
28:     Send(solution, root) {El procesador que tiene la solución se la envía al procesador root}
29:   sino
30:     retornar solution solution
31:   fin si
32: sino
33:   si pr = root entonces
34:     Receive(procmin, solution)
35:     retornar solution {El procesador root retorna la solución}
36:   fin si
37: fin si
```

Algoritmo 20 Subrutina `LoadBalance(↓↑ELocal, ↓sizes, ↑nIDLE)`

```
1: En Paralelo : Para pr = 0, 1, ..., p − 1
2: para proc = 0, 1, ..., p − 1 hacer
3:   si sizes[proc] = 0 entonces
4:     nIDLE ← nIDLE+1;
5:     states[proc] ← IDLE
6:   sino
7:     states[proc] ← ACTIVE
8:   fin si
9: fin para
10: procs ← (0, 1, ..., p − 1)
11: Ordenar procs de mayor a menor según los valores correspondientes de sizes
12: maxToSend ← Size(E)/(nIDLE+1)
13: si state[pr] = IDLE entonces
14:   i ← 0
15:   mientras (i < p) AND (states[pr] = IDLE) hacer
16:     si states[procs[i]] = ACTIVE entonces
17:       Send(REQUEST, 'Y', procs[i])
18:       Receive(procs[i], REPLY, msg)
19:       si msg = ACCEPT entonces
20:         Receive(procs[i], ELocal); states[pr] ← ACTIVE
21:       fin si
22:     fin si
23:     i ← i+1
24:   fin mientras
25:   si state[pr] = ACTIVE entonces
26:     mientras i < p hacer
27:       si states[index[i]] = ACTIVE entonces
28:         Send(REQUEST, 'N', procs[i])
29:       fin si
30:       i ← i+1
31:     fin mientras
32:   fin si
33: sino
34:   para i ← 0, 1,..., nIDLE hacer
35:     Receive(ANY, REQUEST, msg, source)
36:     si msg = 'Y' entonces
37:       si state[pr] = IDLE entonces
38:         Send(REPLY, REJECT, source)
39:       sino
40:         Send(REPLY, ACCEPT, source)
41:         Escoger de ELocal a lo sumo maxToSend nodos y formar una estructura E
42:         Send(E, source)
43:       fin si
44:     fin si
45:   fin para
46: fin si
```

que éstos se agoten o hasta que la variable global antes mencionada indique que algún procesador terminó su trabajo. Luego todos reúnen sus respectivos nodos en una estructura global con el objetivo de distribuir nuevamente de forma cíclica el trabajo.

Esta variante, aplicable sólo en máquinas con varios procesadores que comparten una misma unidad de memoria, disminuye considerablemente los riesgos de que ocurran desequilibrios en la carga de trabajo. Sin embargo, como el algoritmo procura de mantener el trabajo lo más equitativamente posible, puede volverse lento cuando el espacio de búsqueda global sea pequeño, es decir, cuando globalmente queden pocos nodos a expandir, pues la frecuencia conque se detendría la búsqueda porque un procesador terminó su trabajo por agotamiento de nodos crecería.

En el algoritmo 21 se muestran los pasos del algoritmo paralelo para memoria distribuida. La invocación `BestInfExpand(ELocal, mySolution, solutionValue, empty)` ejecuta los pasos del algoritmo 7 hasta que la estructura `ELocal` quede vacía o hasta que la variable compartida `empty` adquiera el valor de `TRUE`. Siempre que se vacíe la estructura de un procesador, este pondrá en `TRUE` la variable `empty`, y como esta variable es compartida, entonces todos saldrán de la rutina.

6.1.3 Paralelización híbrida del Sphere-Decoding

Uno de los objetivos de este trabajo es el diseño e implementación de algoritmos paralelos del SD para entornos paralelos jerarquizados, es decir, redes de multiprocesadores donde la memoria global es distribuida y la memoria local en cada nodo es compartida por varios núcleos de procesamiento. Obviamente para estos entornos se puede usar el modelo de paso de mensajes dada la portabilidad de la librería MPI [109], sin embargo, puede que este modelo no sea el adecuado para aprovechar eficientemente el sistema de memoria compartida en cada nodo de la red.

Explicados los algoritmos paralelos del SD para memoria distribuida y para memoria compartida, se puede inducir de forma sencilla el algoritmo para el entorno paralelo híbrido. Este algoritmo tendría como esquema global el mismo esquema paralelo para memoria distribuida explicado en 6.1.1, mientras que la búsqueda que realizaría cada multiprocesador se haría mediante el algoritmo paralelo de memoria compartida explicado en 6.1.2. Hay que tener en cuenta que en cada iteración del algo-

Algoritmo 21 Algoritmo SD paralelo para memoria compartida

1: {*El hilo master crea el nodo inicial y expande varios nodos*}
2: `EGlobal` ← `CreateStruct(''STACK'')` {*Se crea la pila EGlobal*}
3: `minNodes` ← `nthreads` {*Al menos nthreads nodos deben haber en EGlobal*}
4: `node` ← `CreateInitialNode()` {*Se crea el nodo inicial*}
5: `Add(EGlobal, node)` {*Se adiciona a EGlobal*}
6: `BestBoundExpand(EGlobal, minNodes, solution, solutionValue)`
7: **En Región Paralela con** `nthreads` **hilos de ejecución**
8: **privadas:** `myThreadNum, mySolution, mySolutionValue`
9: **compartidas:** `EGlobal, solution, solutionValue, empty, terminate, redistribute`
10: `ELocal` ← `CreateStruct(''STACK'')` {*Se crean las pilas locales*}
11: {*Distribución cíclica de los elementos de EGlobal*}
12: **para** `i` ← `myThreadNum, myThreadNum+nthreads,...,Size(EGlobal)` **hacer**
13: `Add(ELocal, EGlobal[i])`
14: **fin para**
15: `terminate` ← `false` {*Indica cuando el algoritmo debe terminar porque todas las estructuras están vacías*}
16: `empty` ← `false` {*Indica si hay alguna estructura vacía*}
17: `BestInfExpand(ELocal, mySolution, solutionValue, empty)`
18: **En Región Crítica** actualizar `redistribute` y `terminate` según los tamaños de las estructuras locales
19: **si** `(redistribute = true) AND (terminate = false)` **entonces**
20: **En Región Crítica** insertar los elementos de `ELocal` en `EGlobal`
21: `BARRERA`
22: {*Distribución cíclica de los elementos de EGlobal*}
23: **para** `i` ← `myThreadNum, myThreadNum+nthreads,...,Size(EGlobal)` **hacer**
24: `Add(ELocal, EGlobal[i])`
25: **fin para**
26: **fin si**
27: **En Región Crítica** actualizar `solution` a partir de `mySolution`
28: **Fín de la Región Paralela**
29: **retornar** solution

ritmo, cada multiprocesador debe seleccionar de su pila un número fijo de nodos para expandir. De aquí que el algoritmo paralelo en memoria compartida que ejecutarían los núcleos computacionales de un multiprocesador presenta una ligera diferencia respecto al algoritmo presentado en la sección anterior, y es que además de compartir una variable que indica si algún procesador agotó sus nodos, compartirían además una variable con el valor de la cantidad de nodos a expandir en esa iteración por el procesador. Dicha variable sería decrementada dentro de una región crítica por cada procesador cada vez que expandan un nodo, y si alcanza el valor de cero todos deben detener su búsqueda para, mediante una reducción, determinar la mejor solución encontrada por el multiprocesador y reconstruir su estructura con los nodos que quedaron sin expandir.

En el algoritmo 22 se muestran los pasos del algoritmo paralelo para un esquema híbrido. En la línea 18 del algoritmo 22 se deben ejecutar los pasos del algoritmo 21, pero esta vez se comenzaría justamente a partir de la línea 7, y en este caso la estructura global para los hilos de ejecución será la estructura local de cada proceso del esquema de memoria distribuida. La variable `fixedNodes` también debe compartirse entre los hilos de ejecución, y en lugar de realizar la invocación `BestInfExpand(ELocal, mySolution, solutionValue, empty)` de la línea 17, se debe realizar la invocación `BestInfFixExpand(ELocal, mySolution, solutionValue, empty, fixedNodes)`. Este último método ejecuta los pasos del algoritmo 7 hasta que la estructura `ELocal` quede vacía o hasta que la variable compartida `empty` adquiera el valor de `TRUE`, o hasta que `fixedNodes` llegue a cero. Siempre que un hilo de ejecución expande un nodo de su estructura local, decrementará el valor de la variable `fixedNodes`, y de esta forma se expanden `fixedNodes` de forma paralela.

6.1.4 Resultados experimentales

Los algoritmos paralelos diseñados fueron implementados en C, usando la librería BLAS para las operaciones vectoriales. La versión para la arquitectura de memoria compartida fue implementada usando la librería OpenMP, mientras que en la versión híbrida se usaron ambas librerías. En el capítulo 2 se argumenta la decisión de usar estas herramientas software. Los algoritmos fueron ejecutados en el cluster Rosebud, específicamente en los nodos `rosebud05` y `rosebud06` (ver sus características también

Algoritmo 22 Algoritmo SD paralelo híbrido

1: **En Paralelo : Para** $pr = 0, 1, \ldots, p-1$
2: `EProc` ← `CreateStruct(''STACK'')` {*Cada procesador crea su pila EProc*}
3: **si** `pr = root` **entonces**
4: {*El procesador root expandirá algunos nodos hasta que sean suficientes para distribuirlos*}
5: `EGlobal` ← `CreateStruct(''STACK'')` {*Se crea la pila global EGlobal*}
6: `minNodes` ← `p*nthreads` {*Al menos p*nthreads nodos deben haber en EGlobal*}
7: `node` ← `CreateInitialNode()` {*Se crea el nodo inicial*}
8: `Add(EGlobal, node)` {*Se adiciona a EGlobal*}
9: `BestBoundExpand(EGlobal, minNodes, solution, solutionValue)`
10: **fin si**
11: `CyclicScatterStruct(EGlobal, EProc, root)` {*Distribución cíclica de EGlobal*}
12: `Broadcast(solutionValue, root)` {*Distribución del valor solución hasta ahora obtenido*}
13: `terminate` ← `false`
14: **mientras** `terminate = false` **hacer**
15: `fixedNodes` ← `EstimateFixedNodes(EProc)` {*Estimación de los nodos a expandir*}
16: `AllGather(fixedNodes, allFixedNodes)`
17: `fixedNodes` ← `median(allFixedNodes)`
18: **Algoritmo SD Paralelo para memoria compartida** acotado con el valor de `fixedNodes`. Con `EProc` como estructura global, la cual debe distribuirse por los distintos threads.
19: `AllReduceMin(solutionValue, procmin)`
20: `AllGather(Size(EProc), sizes)`
21: `LoadBalance(EProc, sizes, nIDLE)`{*Se balancea la carga de trabajo*}
22: **si** `nIDLE = p` **entonces**
23: `terminate` ← `true` {*Si todas las estructuras están vacías, entonces termina el algoritmo*}
24: **fin si**
25: **fin mientras**
26: **si** `pr = procmin` **entonces**
27: **si** `pr` $\neq$ `root` **entonces**
28: `Send(solution, root)` {*El procesador que tiene la solución se la envía al procesador root*}
29: **sino**
30: **retornar** `solution solution`
31: **fin si**
32: **sino**
33: **si** `pr = root` **entonces**
34: `Receive(procmin, solution)`
35: **retornar** `solution` {*El procesador root retorna la solución*}
36: **fin si**
37: **fin si**

en el capítulo 2).

Para los experimentos paralelos se escogieron problemas en los cuales el algoritmo SD generó un árbol de búsqueda con un número considerable de nodos, y consecuentemente el tiempo de ejecución en su versión secuencial es notable. De esta forma, se escogieron problemas donde el árbol de búsqueda generado tiene tamaños (tomando como tamaño la cantidad de nodos en total) 50000, 100000, 500000, 1000000, 4000000, 5000000, 8000000 y 10000000. Estos árboles se generaron en problemas donde $n = m = 8$, por lo que el número de niveles del árbol en todos los casos es 8. Es importante resaltar que los tiempos de ejecución del método SD depende no sólo de la cantidad de nodos del árbol generado, sino además de la cantidad de posibles soluciones (nodos del último nivel, que representarían los puntos de la retícula dentro de la hiper-esfera). Si se tienen dos árboles distintos con la misma cantidad de nodos, pero con cantidad de posibles soluciones diferentes, entonces se demoraría menos aquel que tenga más nodos del último nivel, pues haría menos inserciones en la pila, además de que serían menos los nodos a ramificar.

En la figura 6.2 se muestra el comportamiento de la ganancia de velocidad de la versión paralela MPI para los distintos problemas mencionados. Se puede apreciar que para los problemas de menor tamaño (los señalados en el gráfico con colores más claros) se deja de ganar velocidad cuando se usan más de 4 procesadores. Las curvas de speed-up para los problemas de mayor tamaño (los señalados en el gráfico con líneas continuas) sí presentan un comportamiento creciente, aunque las ganancias de velocidad está aún lejos del óptimo teórico.

En la figura 6.3 se muestra el comportamiento de la ganancia de velocidad de la versión paralela OpenMP. Esta versión se ejecutó en una de las máquinas mencionadas. Es notable en este caso que todas las curvas presentan un mismo comportamiento, y que en todos los problemas el número óptimo de procesadores (con el cual se alcanza la mayor ganancia de velocidad) es 6. Las ganancias de velocidades que se obtienen en esta versión para memoria compartida son muy cercanas a la ganancia óptima teórica, superando en la mayoría de los casos las obtenidas por la versión MPI. Este resultado es muy importante si tenemos en cuenta que en el mercado es cada vez más significativa la presencia de computadoras personales con procesadores *dual-core* o incluso *quad-core*, con las cuales se podrían obtener tiempos de ejecución hasta 4 veces menor.

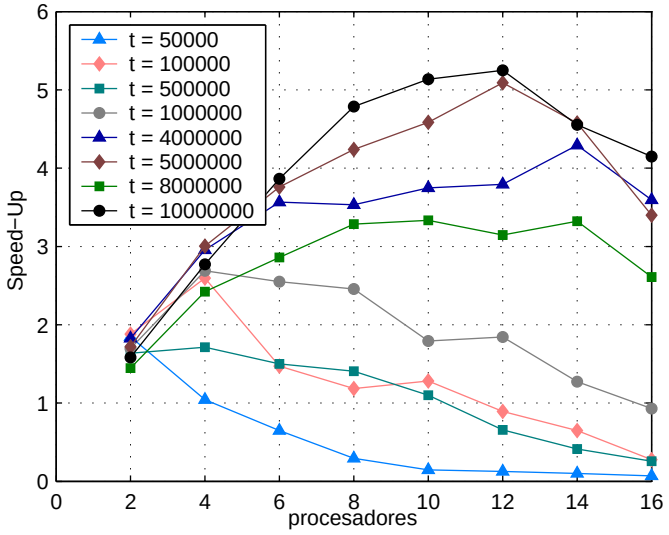


Figura 6.2: SpeedUp de la versión paralela MPI del SD

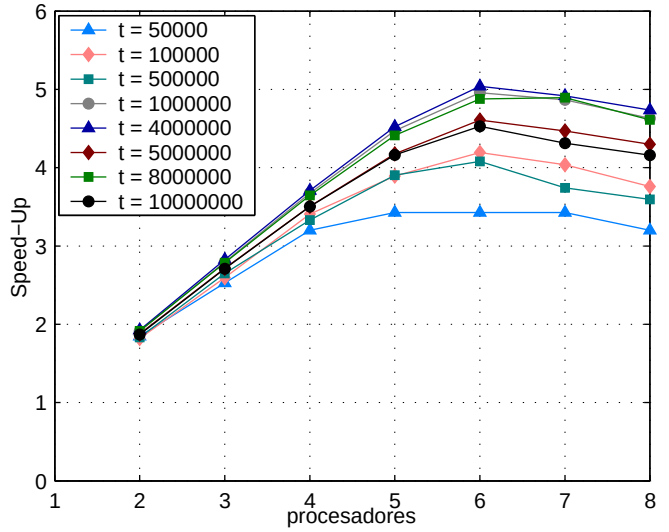


Figura 6.3: SpeedUp de la versión paralela OpenMP del SD

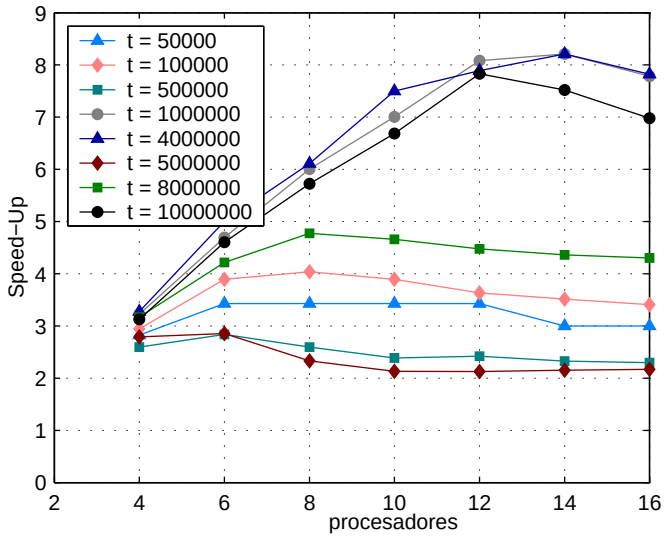


Figura 6.4: SpeedUp de la versión paralela híbrida del SD

Finalmente en la figura 6.4 se muestra el comportamiento de la ganancia de velocidad de la versión paralela híbrida. La versión paralela híbrida se implementó usando MPI para un primer nivel (nivel global) de paralelización, y OpenMP para un segundo nivel (nivel local). Con este algoritmo para algunos problemas se obtuvo una ganancia de velocidad mucho mayor que las obtenidas en las versiones usando sólo MPI u OpenMP. Los problemas en los que se obtuvo una ganancia de velocidad superior a 7, son aquellos en los que hay mayor cantidad de nodos solución en el árbol de búsqueda (más de un 70 % de nodos solución). En problemas donde el por ciento de nodos solución en el árbol es muy pequeño, el rendimiento de este método se degrada, como en el problema donde la cantidad de nodos es aproximadamente 500000, de los cuales sólo el 11 % son nodos solución.

En resumen, con la versión híbrida (MPI + OpenMP) se obtuvieron las mayores ganancias de velocidad, aunque el comportamiento de las ganancias de velocidad fue mucho más estable para todos los problemas usando el algoritmo para memoria compartida. Dicho comportamiento era esperado debido a la inexistencia de comunicaciones y a que se puede lograr un mayor equilibrio de la carga de trabajo, de ahí que el algoritmo para memoria compartida sea el de mayor eficiencia de los tres. El equilibrio de la carga de trabajo es mucho más complicado en entornos con memoria distribuida, de ahí que el rendimiento de los algoritmos MPI y MPI+OpenMP es variable en dependencia de la

estructura del árbol de búsqueda generado en el problema.

6.2 Paralelización del Automatic Sphere-Decoding

En esta sección se expondrán algunas variantes de paralelización del método ASD para distintos entornos paralelos: con memoria distribuida, con memoria compartida, y en entornos que se componen de varios multiprocesadores comunicados mediante una red de interconexión.

La paralelización de este algoritmo tiene principios básicos en común respecto a los algoritmos paralelos del SD descritos en la sección 6.1.

6.2.1 Paralelización del ASD en un entorno de memoria distribuida

En la paralelización del ASD para un entorno con memoria distribuida se puede emplear la misma distribución inicial de la estructura que se emplea en el algoritmo descrito en la sección 6.1.1. Por tanto, los pasos iniciales del algoritmo paralelo son los mismos: el procesador *root* crea un nodo inicial y a partir de este nodo comienza una ejecución secuencial del ASD hasta que en el *heap* hayan suficientes nodos para ser distribuidos al resto de los procesadores (al menos uno por procesador). Luego la distribución es igualmente cíclica.

Al igual que en el algoritmo paralelo del SD, en cada iteración, cada procesador expande un número fijo de nodos de su estructura. Después que cada procesador expande el número prefijado de nodos, habrá un punto de sincronización para que todos sepan el estado de el *heap* de cada uno (si está vacío o no).

En este punto de sincronización se determina primeramente si el algoritmo debe terminar o debe continuar. Para ello, los procesadores deben reunir (mediante un *AllGather*) un valor que indique si encontraron o no una solución (un nodo hoja del árbol), y un valor real que represente el peso de un nodo, que en el caso de que se haya encontrado una posible solución sería el peso de ese nodo hoja y en caso contrario sería el peso del nodo que ocupa la primera posición de su correspondiente estructura. Estos valores son necesarios para poder determinar si el algoritmo debe continuar o terminar, pues no basta con la condición de si se encontró un nodo de nivel m, pues puede suceder que un procesador encuentre un nodo hoja que no sea precisamente la solución, es decir, la que ofrece el punto de la

retícula más cercano a x. Para poder determinar si un nodo hoja encontrado por un procesador es la solución del problema, su costo debe ser menor que los costos de todos los nodos que están en la cabeza de cada *heap* local en cada procesador. Si sucede que al menos el primer elemento del *heap* de un procesador tiene un peso menor que el nodo hoja encontrado, entonces no se puede asegurar que dicho nodo hoja es solución, pues no se puede descartar que a partir de dicho primer elemento se pueda llegar a un nodo hoja con un peso menor.

En caso que efectivamente el nodo hoja encontrado sea la solución del problema entonces se retorna, en otro caso se almacena como solución temporal. Luego, cada procesador donde el primer elemento de su correspondiente *heap* tiene un peso mayor que el de la solución temporal debe proceder a vaciar la pila, pues ninguno de los nodos que tiene almacenados podrían derivar en un nodo con menor peso que la solución temporal. Por la misma razón, el procesador que encontró la solución temporal, debe también eliminar todos los nodos almacenados en su *heap*.

Luego se procede a balancear nuevamente la carga de trabajo, que se haría de la misma forma que en el algoritmo paralelo para memoria distribuida del SD.

El tema de la estimación de la cantidad de nodos a expandir en la próxima iteración cobra más importancia en este algoritmo, pues ahora los procesadores pueden detener su trabajo bien porque se le agotaron los nodos en el *heap* o porque encontró un nodo hoja. Por lo que la posibilidad de que ocurran desequilibrios es mayor. En este caso se precisa que la cantidad de nodos a expandir sea incluso menor que la que se puede estimar en el algoritmo paralelo del SD. Es por ello que se propone en este trabajo, que cada procesador recorra su *heap* sumando por cada elemento la cantidad de nodos que este pueda expandir, hasta que encuentre uno cuyo nivel sea $m-1$. La suma total sería su estimación de nodos a expandir en la próxima iteración. En este algoritmo, se escoge, mediante una operación de reducción, la menor de todas las estimaciones hechas por los procesadores, y sería esa cantidad de nodos la que se visitaría en la próxima iteración.

En el algoritmo 23 se muestran los pasos del algoritmo paralelo para memoria distribuida. La invocación `FirstBoundExpand(EGlobal, minNodes, solution, solutionValue, success)` ejecuta los pasos del algoritmo 6, descrito en la sección 3.4, hasta que la estructura `EGLobal` contenga al menos `minNodes` nodos. Deben pasarse por parámetros las variables `solution` y `solutionValue`

pues puede suceder que sea necesario actualizarlas, y el parámetro de salida `success` indicaría si efectivamente se encontró una solución en la búsqueda. La invocación `FirstFixExpand(fixedNodes, ELocal, solution, solucionValue, success)` realiza la misma ejecución, lo que en este caso expande exactamente `fixedNodes` de la estructura `ELocal`. Por otro lado, en este algoritmo la rutina `EstimateFixedNodes` realiza la estimación de nodos a expandir en la iteración, siguiendo la propuesta descrita en el párrafo anterior.

6.2.2 Paralelización del ASD en un entorno de memoria compartida

Como se explicó anteriormente, la probabilidad de que ocurran desequilibrios en el algoritmo paralelo del ASD para memoria distribuida es mayor, dado que la continuación del trabajo en cada procesador depende de dos condiciones: que su *heap* tenga nodos que expandir y que aún no haya encontrado algún nodo del último nivel. La paralelización del método ASD en un entorno con memoria compartida es bastante similar al algoritmo paralelo descrito en la sección 6.1.2. La diferencia estaría en que esta vez es necesaria una variable adicional, compartida por todos los procesadores, que indique si un procesador ha encontrado un nodo hoja durante su búsqueda. Esta variable será actualizada dentro de una sección crítica por el primer procesador que haya encontrado un nodo hoja. Cuando se haya detenido la búsqueda porque algún procesador agotó todos sus nodos o porque encontró un nodo hoja, se deben tener las mismas consideraciones que se tienen en cuenta en el algoritmo descrito en 6.2.1 para determinar si se debe proseguir la búsqueda o si se debe terminar porque se encontró la solución. En caso que el algoritmo deba continuar, y algún procesador tiene su *heap* vacío, todos reúnen sus respectivos nodos en una estructura global con el objetivo de distribuir nuevamente de forma cíclica el trabajo.

En el algoritmo 24 se muestran los pasos del algoritmo paralelo para memoria distribuida. La invocación `FirstInfExpand(ELocal, mySolution, empty, success, searchInfo)` ejecuta los pasos del algoritmo 6 hasta que la estructura `ELocal` quede vacía o hasta que una de las dos variables compartidas `empty` y `success` adquieran el valor de `TRUE`. Siempre que un hilo de ejecución encuentre una solución, este pondrá en `TRUE` la variable `success`, y como esta variable es compartida, entonces

Algoritmo 23 Algoritmo ASD paralelo para memoria distribuida

```
1: En Paralelo : Para pr = 0, 1, ..., p − 1
2: ELocal ← CreateStruct(''HEAP'') {Cada procesador crea su heap ELocal}
3: si pr = root entonces
4:   {El procesador root expandirá algunos nodos hasta que sean suficientes para distribuirlos}
5:   EGlobal ← CreateStruct(''HEAP'') {Se crea el heap EGlobal}
6:   minNodes ← p {Al menos p nodos deben haber en EGlobal}
7:   node ← CreateInitialNode() {Se crea el nodo inicial}
8:   Add(EGlobal, node) {Se adiciona a EGlobal}
9:   FirstBoundExpand(EGlobal, minNodes, solution, solutionValue, success)
10: fin si
11: FirstCyclicScatter(EGlobal, ELocal, root, success, solution) {En caso que success sea
    true, no se distribuye el heap EGlobal, sino que se envía a todos la solución y el valor de success}
12: si success = true entonces
13:   retornar solution {Si se llegó a alguna solución, esta es retornada}
14: fin si
15: terminate ← false
16: mientras terminate = false hacer
17:   fixedNodes ← EstimateFixedNodes(ELocal) {Estimación de los nodos a expandir}
18:   AllGather(fixedNodes, allFixedNodes)
19:   fixedNodes ← min(allFixedNodes) {Se toma el mínimo de las estimaciones}
20:   FirstFixExpand(fixedNodes, ELocal, solution, solucionValue, success)
21:   {Si se encontró alguna solución, hay que determinar cuál es y cuál procesador la tiene}
22:   si success = true entonces
23:      value ← ValueOf(solution)
24:   sino
25:     si IsNotEmpty(ELocal) entonces
26:        value ← ValueOf(ELocal[0]) {Se toma el valor del primer elemento}
27:     sino
28:        value ← ∞ {Un valor grande cualquiera}
29:     fin si
30:   fin si
31:   AllGather(success, allSuccess)
32:   AllReduceMin(value, procmin) {El proceso procmin tiene la variable value de menor valor}
33:   si allSuccess[procmin] = true entonces
34:     Broadcast(solution, procmin) {Distribución de la solución}
35:     retornar solution
36:   fin si
37:   AllGather(Size(ELocal), sizes)
38:   LoadBalance(ELocal, sizes, nIDLE){Se balancea la carga de trabajo}
39:   si nIDLE = p entonces
40:     terminate ← true {Si todas las estructuras están vacías, entonces termina el algoritmo sin
        haber encontrado solución}
41:   fin si
42: fin mientras
43: retornar NoSolution() {Se retorna un nodo nulo}
```

todos saldrán de la rutina.

6.2.3 Paralelización híbrida del ASD

También se trabajó en la paralelización del ASD para entornos paralelos donde varios multiprocesadores están conectados mediante una red de interconexión, como se explicó en la sección 6.1.4.

Al igual que el algoritmo paralelo híbrido del SD, al algoritmo paralelo híbrido del ASD tendría como esquema global el mismo esquema paralelo para memoria distribuida explicado en 6.2.1, mientras que la búsqueda que realizaría cada multiprocesador se haría mediante el algoritmo paralelo de memoria compartida explicado en 6.2.2. De igual manera, hay que tener en cuenta que en cada iteración del algoritmo, cada multiprocesador debe seleccionar del *heap* un número fijo de nodos para expandir. De aquí que el algoritmo paralelo en memoria compartida que ejecutarían los núcleos computacionales de un multiprocesador compartirían también una variable con el valor de la cantidad de nodos a expandir en esa iteración por el procesador.

En el algoritmo 25 se muestran los pasos del algoritmo paralelo para un esquema híbrido. En la línea 20 se deben ejecutar los pasos del algoritmo 24, pero esta vez se comenzaría justamente a partir de la línea 7, y en este caso la estructura global para los hilos de ejecución será la estructura local de cada proceso del esquema de memoria distribuida. La variable `fixedNodes` también debe compartirse entre los hilos de ejecución, y en lugar de realizar la invocación `FirstInfExpand( ELocal, mySolution, empty, success, searchInfo)` de la línea 17, se debe realizar la invocación `FirstInfFixExpand(ELocal, mySolution, empty, success, searchInfo, fixedNodes)`. Este último método ejecuta los pasos del algoritmo 6 hasta que la estructura `ELocal` quede vacía o hasta que una de las variables compartidas `empty` y `success` alcance el valor de `TRUE`, o hasta que `fixedNodes` llegue a cero.

6.2.4 Resultados experimentales

En la implementación de los algoritmos paralelos del ASD se usaron las mismas herramientas software y hardware en su implementación y puesta a punto.

Algoritmo 24 Algoritmo ASD paralelo para memoria compartida

1: {*El hilo master crea el nodo inicial y expande varios nodos*}
2: `EGlobal` ← `CreateStruct(''HEAP'')` {*Se crea el heap EGlobal*}
3: `minNodes` ← `nthreads` {*Al menos nthreads nodos deben haber en EGlobal*}
4: `node` ← `CreateInitialNode()` {*Se crea el nodo inicial*}
5: `Add(EGlobal, node)` {*Se adiciona a EGlobal*}
6: `FirstBoundExpand(EGlobal, minNodes, solution, solutionValue, success)`
7: **En Región Paralela con** `nthreads` **hilos de ejecución**
8: **privadas:** `myThreadNum, mySolution, mySolutionValue`
9: **compartidas:** `EGlobal, solution, value, empty, success terminate, redistribute, searchInfo` {*searchInfo es un arreglo de 2*nthreads elementos, donde en la posición* $2i$ *se almacena si el hilo* i *encontró una solución, y en la posición* $2i+1$ *se almacena el valor de la solución*}
10: `ELocal` ← `CreateStruct(''HEAP'')` {*Se crean los heaps locales*}
11: {*Distribución cíclica de los elementos de EGlobal*}
12: **para** `i` ← `myThreadNum, myThreadNum+nthreads,...,Size(EGlobal)` **hacer**
13: `Add(ELocal, EGlobal[i])`
14: **fin para**
15: `terminate` ← `false` {*Indica cuando el algoritmo debe terminar*}
16: `empty` ← `false` {*Indica si hay alguna estructura vacía*}
17: `FirstInfExpand(ELocal, mySolution, empty, success, searchInfo)`
18: **si** `searchInfo[2*myThreadNum] = false` **entonces**
19: {*El hilo myThreadNum no encontró solución*}
20: **si** `IsNotEmpty(ELocal)` **entonces**
21: `searchInfo[2*myThreadNum+1]` ← `ValueOf(ELocal[0])` {*Se toma el valor del primer elemento*}
22: **sino**
23: `searchInfo[2*myThreadNum+1]` $\leftarrow \infty$ {*Un valor grande cualquiera*}
24: **fin si**
25: **fin si**
26: `threadmin` ← toma el valor `i` que minimiza a `searchInfo[2*i+1]`
27: **si** `myThreadNum = threadmin` **entonces**
28: solution ← mySolution {*Se actualiza la variable compartida solution*}
29: **fin si**
30: **En Región Crítica** actualizar `redistribute` y `terminate` según el valor de success y los tamaños de las estructuras locales
31: **si** `(redistribute = true) AND (terminate = false)` **entonces**
32: **En Región Crítica** insertar los elementos de `ELocal` en `EGlobal`
33: `BARRERA`
34: {*Distribución cíclica de los elementos de EGlobal*}
35: **para** `i` ← `myThreadNum, myThreadNum+nthreads,...,Size(EGlobal)` **hacer**
36: `Add(ELocal, EGlobal[i])`
37: **fin para**
38: **fin si**
39: **Fín de la Región Paralela**
40: **retornar** solution

Algoritmo 25 Algoritmo ASD paralelo híbrido

1: **En Paralelo : Para** $pr = 0, 1, \ldots, p-1$
2: `EProc` ← `CreateStruct(''HEAP'')`
3: **si** `pr = root` **entonces**
4: {*El procesador root expandirá algunos nodos hasta que sean suficientes para distribuirlos*}
5: `EGlobal` ← `CreateStruct(''HEAP'')` {*Se crea el heap EGlobal*}
6: `minNodes` ← `p*nthreads` {*Al menos p*nthreads nodos deben haber en EGlobal*}
7: `node` ← `CreateInitialNode()` {*Se crea el nodo inicial*}
8: `Add(EGlobal, node)` {*Se adiciona a EGlobal*}
9: `FirstBoundExpand(EGlobal, minNodes, solution, solutionValue, success)`
10: **fin si**
11: `FirstCyclicScatter(EGlobal, ELocal, root, success, solution)`
12: **si** `success = true` **entonces**
13: **retornar** solution {*Si se llegó a alguna solución, esta es retornada*}
14: **fin si**
15: `terminate` ← `false`
16: **mientras** `terminate = false` **hacer**
17: `fixedNodes` ← `EstimateFixedNodes(ELocal)` {*Estimación de los nodos a expandir*}
18: `AllGather(fixedNodes, allFixedNodes)`
19: `fixedNodes` ← `min(allFixedNodes)` {*Se toma el mínimo de las estimaciones*}
20: **Algoritmo ASD Paralelo para memoria compartida** acotado con el valor de `fixedNodes`. Con `EProc` como estructura global, la cual debe distribuirse por los distintos threads.
21: {*Si se encontró alguna solución, hay que determinar cuál es y cuál procesador la tiene*}
22: **si** `success = true` **entonces**
23: `value` ← `ValueOf(solution)`
24: **sino**
25: **si** `IsNotEmpty(ELocal)` **entonces**
26: `value` ← `ValueOf(ELocal[0])` {*Se toma el valor del primer elemento*}
27: **sino**
28: `value` ← ∞ {*Un valor grande cualquiera*}
29: **fin si**
30: **fin si**
31: `AllGather(success, allSuccess)`
32: `AllReduceMin(value, procmin)` {*El proceso procmin tiene la variable value de menor valor*}
33: **si** `allSuccess[procmin] = true` **entonces**
34: `Broadcast(solution, procmin)` {*Distribución de la solución*}
35: **retornar** solution
36: **fin si**
37: `AllGather(Size(ELocal), sizes)`
38: `LoadBalance(ELocal, sizes, nIDLE)`{*Se balancea la carga de trabajo*}
39: **si** `nIDLE = p` **entonces**
40: `terminate` ← `true`
41: **fin si**
42: **fin mientras**
43: **retornar** `NoSolution()` {*Se retorna un nodo nulo*}

Para los experimentos paralelos se escogieron problemas en los cuales el algoritmo ASD generó un árbol de búsqueda con un número considerable de nodos, y consecuentemente el tiempo de ejecución en su versión secuencial es notable. De esta forma, se escogieron problemas donde el árbol de búsqueda generados tiene tamaños (tomando como tamaño la cantidad de nodos en total) $5 \cdot 10^4$, $1 \cdot 10^5$, $2 \cdot 10^5$, $4 \cdot 10^5$, $5 \cdot 10^5$, $6 \cdot 10^5$, $8 \cdot 10^5$ y $1 \cdot 10^6$. Estos árboles se generaron en problemas donde $n = m = 8$, por lo que el número de niveles del árbol en todos los casos es 8.

Es importante resaltar que los tiempos de ejecución del método ASD depende no sólo de la cantidad de nodos del árbol generado, sino específicamente de la cantidad de nodos hijos que como promedio tienen los nodos en su ramificación. Si el promedio de nodos en cada ramificación es alto, entonces el tiempo secuencial es mayor, pues se harían más operaciones de inserciones en el *heap* y recordemos que cada operación de inserción en el heap tiene complejidad $O(\log n)$. Para aquellos problemas donde cada nodo del árbol ramifica muy pocos nodos, puede ser difícil mejorarlos significativamente Por otro lado, los algoritmos paralelos pueden presentar *speed-ups* por encima del máximo teórico debido a pueden encontrar la solución examinando menos nodos que el algoritmo secuencial, además de hacerlo de forma concurrente. De modo análogo, puede suceder que el algoritmo paralelo genere y analice más nodos durante la búsqueda, sobre todo en los algoritmos diseñados para memoria distribuida. Ello puede implicar que la ganancia de velocidad no llegue al óptimo deseado. En [77] se describen las diferentes anomalías que pueden existir en la paralelización de algoritmos *branch-and-bound*.

En la figura 6.5 se muestra el speed-up de la versión paralela para memoria compartida (MPI) en los casos de prueba considerados. En los problemas de menor tamaño se alcanza muy poca ganancia de velocidad, alcanzando su valor máximo cuando la cantidad de procesadores es 6. En los problemas de mayor tamaño se alcanzan mejores ganancias de velocidad, incluso superando el máximo teórico notablemente en algunos casos.

En la figura 6.6 se muestra el speed-up de la versión paralela para memoria distribuida (OpenMP) en los casos de prueba considerados. Se puede observar que las ganancias de velocidad son algo mejores que las alcanzadas por la versión MPI tomando la misma cantidad de procesadores. Esto se puede deber a que en la versión paralela para memoria distribuida se puede evitar mejor el desbalance en la carga de trabajo, y puede lograrse con mayor probabilidad que la versión paralela no genere y

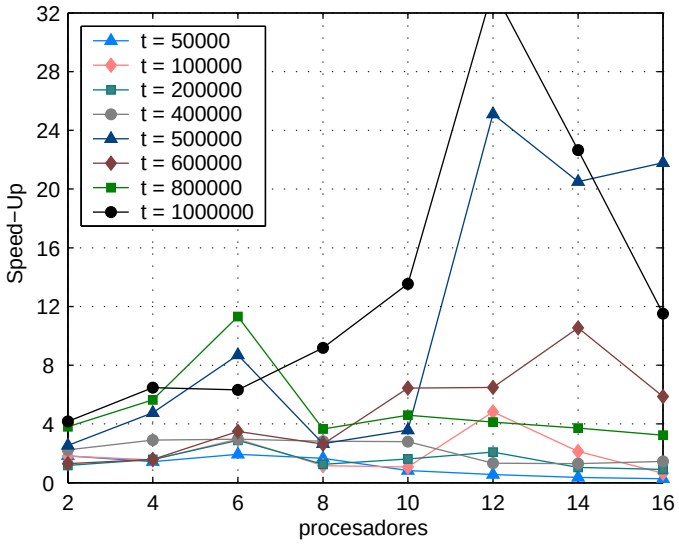


Figura 6.5: SpeedUp de la versión paralela MPI del ASD

ramifique mucho más nodos que la versión secuencial.

Finalmente en la figura 6.7 se muestra el speed-up de la versión paralela híbrida. Ya en esta versión en la mayoría de los casos se alcanzan speed-ups mayores que 4 incluso en los problemas pequeños. En sentido general, la versión híbrida tuvo mejor rendimiento que las dos versiones anteriores.

En resumen, la versión híbrida obtuvo las mejores ganancias de velocidad, y estas fueron más estables comparados con los obtenidos por las versiones en MPI y OpenMP.

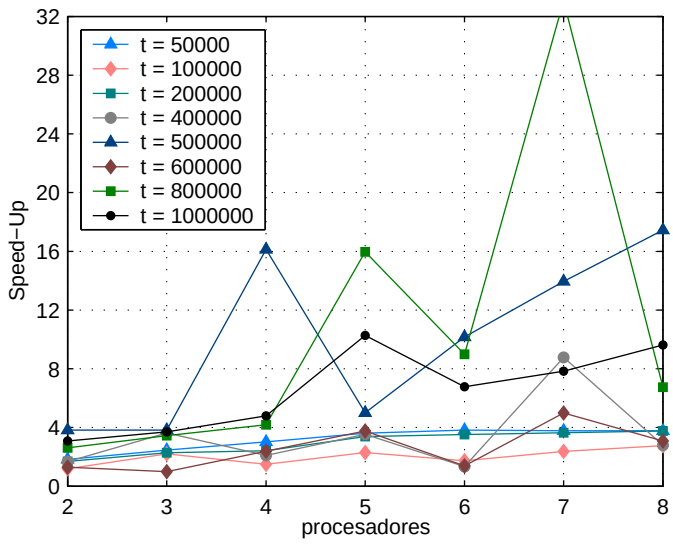


Figura 6.6: SpeedUp de la versión paralela OpenMP del ASD

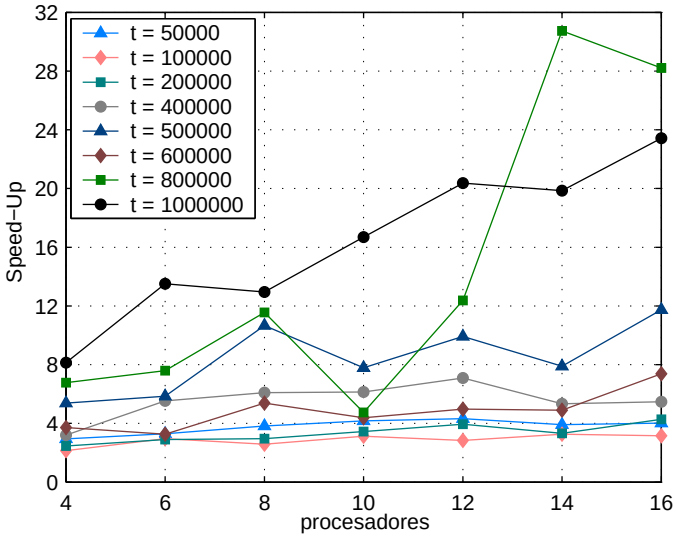


Figura 6.7: SpeedUp de la versión paralela híbrida del ASD

7

Conclusiones y trabajo futuro

7.1 Conclusiones

En la presente tesis se han abordado soluciones paralelas eficientes al Problema del Punto más Cercano (CVP) en retículas aplicado a la decodificación de señales en los sistemas de comunicaciones inalámbricas con arquitecturas MIMO. Las paralelizaciones fueron realizadas tanto a métodos de optimización por búsqueda directa como a métodos basados en búsqueda por árbol, específicamente los métodos conocidos por el nombre *Sphere-Decoding* que surgen para resolver de manera particular el problema CVP.

Se hizo un estudio profundo y detallado del problema CVP, y de igual manera se realizó un Estado del Arte de los principales métodos de solución del problema CVP encontrados en la literatura, como es la reducción de bases generadoras, los métodos llamados heurísticos o sub-óptimos, y los métodos conocidos como exactos u óptimos, entre los cuales destaca el *Sphere-Decoding* y sus diferentes variantes.

De modo análogo se realizó un estudio del Estado del Arte de la optimización mediante métodos de Búsqueda Directa. Se presentaron e implementaron varios algoritmos secuenciales de búsqueda directa pertenecientes a la clase GSS (*Generating Set Search*). Los algoritmos fueron aplicados primeramente a un problema altamente costoso como es el Problema Inverso de Valores Singulares, y permitió corrob-

orar que efectivamente estos métodos son asintóticamente lentos, pero a la vez son robustos y capaces de acercarse bastante a la solución en problemas lo suficientemente complejos y partiendo de puntos lo suficientemente lejanos como para que algunos métodos basados en derivadas en esas condiciones no logren converger.

Se realizaron implementaciones paralelas de los métodos de búsqueda directa de mejor rendimiento. Se desarrollaron dos métodos paralelos síncronos, uno por cada algoritmo secuencial seleccionado, y un método paralelo asíncrono correspondiente al segundo método secuencial. En sentido general los métodos paralelos lograron buenas ganancias de velocidad en el problema IAVSP, además de que resultaron poseer una escalabilidad relativamente buena.

Debido a que la clase de métodos GSS ofrece varios parámetros que puedan ser adaptados al tipo de problema que se quiera resolver, se pudo realizar una configuración de estos métodos para que puedan resolver el problema de decodificación de señales en sistemas MIMO. Los resultados obtenidos en esta aplicación resultan en gran medida muy interesantes, pues se pudo comprobar que pueden alcanzar la solución ML o casi ML empleando, por lo general, tiempos menores que los métodos SD o ASD. A nuestro juicio este es uno de los resultados de la tesis que merece ser destacado, pues los métodos de búsqueda directa son mucho más sencillos y más "paralelizables" que los métodos SD y se podría pensar en lograr una solución paralela al problema CVP más rápida (sobre todo con estabilidad en los tiempos de ejecución), y a la vez bastante precisa.

En la investigación realizada sobre los métodos *Sphere-Decoding* se alcanzaron también resultados alentadores. En primer lugar se realizó una propuesta de aplicación de la descomposición de valores singulares en la selección de radios iniciales para la búsqueda, y se pudo comprobar que esta estrategia es efectiva en casos donde la decodificación se realiza en retículas relativamente grandes (que implica una mayor cantidad de símbolo en el alfabeto o constelación, o una mayor dimensión del problema) y en condiciones donde el ruido no tenga bajos niveles. En esos casos la esquema propuesto alcanzó mejores resultados que propuestas de radios basados en métodos heurísticos.

De igual manera se pudo realizar una mejora, hasta cierto punto elemental, del método ASD incorporándole un radio de búsqueda con el cual puede construir árboles de búsqueda con menos ramificaciones. En las experimentaciones realizadas se pude comprobar al ahorro de tiempo que ello

implica.

Se realizaron implementaciones paralelas de los métodos SD y ASD, para diferentes arquitecturas paralelas: memoria distribuida, memoria compartida y esquemas híbridos. Con los esquemas híbridos en ambos casos se alcanzaron las mayores ganancias de velocidad. Pero hay que resaltar que el rendimiento de los algoritmos paralelos dependen en gran medida de la estructura del árbol de búsqueda, que a su vez es único para cada instancia del problema a resolver. Esto se nota sobre todo en los métodos ASD, donde no se recorre de manera total el árbol. En sentido general, el balanceo de la carga entre los procesadores en la paralelización de métodos tipo *Branch-and-Bound* es una tarea complicada de resolver sin emplear más comunicaciones. En estos casos siempre sería necesario experimentar cuál sería el justo equilibrio entre balance de carga y tiempo de comunicaciones, que influye de manera directa en el tiempo paralelo de los algoritmos.

Tanto los algoritmos secuenciales como los paralelos se han implementado utilizando librerías estándar en aras de obtener tres características deseables en toda implementación: portabilidad, robustez y eficiencia. También se realizaron con un nivel de abstracción y encapsulamiento de modo que puedan ser usadas no sólo para solucionar el problema de la decodificación en sistemas MIMO sino que permiten abordar cualquier problema de optimización numérica con estos métodos.

El trabajo realizado en la presente tesis doctoral ha dado lugar a las publicaciones [125], [126] y [127].

7.2 Propuestas para el trabajo futuro

En el tintero quedaron abiertas varias líneas que merecen su atención e investigación en un futuro y que servirían de complemento y perfección del trabajo realizado. Estas son las que se proponen:

- Estudiar otras estrategias de acelerar la convergencia de los métodos de búsqueda directa. Se debe hacer énfasis en la selección y actualización de las direcciones de búsqueda.
- Investigar otras variantes de paralelismo asíncrono y aplicarlos en multicomputadores heterogéneos.
- Implementación de los algoritmos paralelos en otras arquitecturas como implementaciones VLSI (FPGA) y multiprocesadores DSP.

- Investigar la posible hibridación de la búsqueda directa con métodos de decodificación por esfera, de modo que la búsqueda directa pueda ser acotada mediante hiper-esferas.

- Investigar otras variantes de búsquedas de la solución del problema CVP donde se use la descomposición de valores singulares, de modo que la búsqueda pueda ser dirigida por los valores y vectores singulares de la matriz de canales.

- Aplicar los métodos implementados en problemas CVP que puedan surgir en otros campos como la criptografía, sistemas de audio, etc.

Índice de figuras

Índice de algoritmos

Bibliografía

[1] *Optimization ToolBox User's Guide*, The MathWorks, Inc, 3 Apple Hill Drive Natick, MA 01760-2098, 2001.

[2] L. Afflerbach and H. Grothe, *Calculation of Minkowski-reduced lattice bases*, Computing, 35 (1985), pp. 269–276.

[3] E. Agrell, T. Eriksson, A. Vardy, and K. Zeger, *Closest Point Search in Lattices*, IEEE Transactions on Information Theory, 48 (2002), pp. 2201–2214.

[4] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *Data Structures and Algorithms*, Addison Wesley, January 1983.

[5] M. Ajtai, *Generating hard instances of lattice problems*, in STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, New York, NY, USA, 1996, ACM, pp. 99–108.

[6] ——, *The shortest vector problem in L2 is NP-hard for randomized reductions*, in STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing, New York, NY, USA, 1998, ACM, pp. 10–19.

[7] M. Ajtai, R. Kumar, and D. Sivakumar, *A sieve algorithm for the shortest lattice vector problem*, in STOC '01: Proceedings of the thirty-third annual ACM symposium on Theory of computing, New York, NY, USA, 2001, ACM, pp. 601–610.

[8] S. M. Alamouti, *A simple transmit diversity technique for wireless communications*, IEEE Journal Selected Areas in Communications, 16 (1998), pp. 1451–1458.

[9] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. Mackenney, S. Ostrouchv, and D. Sorensen, *LAPACK User Guide*, second ed., 1995.

[10] S. Arora, L. Babai, J. Stern, and Z. Sweedyk, *The Hardness of Approximate Optima in Lattices, Codes, and Systems of Linear Equations*, in IEEE Symposium on Foundations of Computer Science, 1993, pp. 724–733.

[11] L. Babai, *On Lovász lattice reduction and the nearest lattice point problem*, Combinatorica, 6 (1986), pp. 1–13.

[12] L. G. Barbero and J. S. Thompson, *A Fixed-Complexity MIMO Detector Based on the Complex Sphere Decoder*, IEEE 7th Workshop on Signal Processing Advances in Wireless Communications. SPAWC '06., (2006), pp. 1–5.

[13] M. Bayat and V. T. Vakily, *Lattice Decoding Using Accelerated Sphere Decoder*, in The 9th International Conference on Advanced Communication Technology, July 2007, pp. 1062–1065.

[14] R. E. Bellman and S. E. Dreyfus, *Applied Dynamic Programming*, Princeton University Press, Princeton, NJ, USA, 1962.

[15] I. Berenguer and X. Wang, *Space-time coding and signal processing for MIMO communications*, Journal of Computer Science and Technology, 6 (2003), pp. 689–702.

[16] G. Berman, *Lattice Approximations to the Minima of Functions of Several Variables*, J. ACM, 16 (1969), pp. 286–294.

[17] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *ADIFOR: Generating derivative codes from Fortran programs*, Scientific Programming, 1 (1992), pp. 11–29.

[18] C. Bischof, A. Carle, P. Hovland, P. Khademi, and A. Mauer, *ADIFOR 2.0 User's Guide (Revision D)*, Tech. Report CRPC-95516-S, Center for Research on Parallel Computation, Rice University, Houston, TX, June 1998.

[19] C. Bischof, A. Carle, P. Khademi, and A. Mauer, *The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs*, in Rice University, 1994, pp. 18–32.

[20] C. Bischof, L. Roh, and A. Mauer, *ADIC: An extensible automatic differentiation tool for ANSI-C*, Tech. Report ANL/CS-P626-1196, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1996.

[21] A. Brooke, D. Kendrick, A. Meeraus, and R. Raman, *GAMS: A User's Guide*, December 1998.

[22] L. Brunel and J. J. Boutros, *Euclidean space lattice decoding for joint detection in CDMA systems*, Proceedings of the 1999 IEEE Information Theory and Communications Workshop., (1999), p. 129.

[23] J. Céa, *Optimisation: Théorie et algorithmes*, (1971).

[24] R. Calkin, R. Hempel, H. C. Hoppe, and P. Wypior, *Portable programming with the PARMACS message-passing library*, Parallel Computing, 20 (1994), pp. 615–632.

[25] J. W. S. Cassels, *An Introduction to the Geometry of Numbers*, Springer, 1 ed., 1959.

[26] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[27] M. T. Chu, *Numerical Methods for Inverse Singular Value Problems*, SIAM Journal Numerical Analysis, 29 (1991), pp. 885–903.

[28] H. Cohen, *A Course in Computacional Algebraic Number Theory*, 1993.

[29] A. R. Conn, N. I. M. Gould, and P. L. Toint, *Trust-Region Methods*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[30] L. Dagum and R. Menon, *OpenMP: An Industry Standard API for Shared-Memory Programming*, IEEE Computational Science and Engineering, 5 (1998), pp. 46–55.

[31] M. O. Damen, H. E. Gamal, and G. Caire, *On Maximum-Likelihood Detection and the Search for the Closest Lattice Point*, IEEE Transactions on Information Theory, 49 (2003), pp. 2389–2402.

[32] S. Dash, R.Ñeelamani, and G. B. Sorkin, *Minkowski and KZ reduction of Nearly Orthogonal Lattice Bases*, Tech. Report W0811-137, IBM Research Division, November 2008.

[33] W. C. Davidon, *Variable metric method for minimization.*, SIAM Journal Optimization, 1 (1991), pp. 1–17.

[34] J. E. Dennis, Jr. and V. Torczon, *Direct search methods on parallel machines.*, SIAM Journal Optimization, 1 (1991), pp. 448–474.

[35] E. F. Deprettere, ed., *SVD and Signal Processing: Algorithms, Applications and Architectures*, North-Holland Publishing Co., Amsterdam, The Netherlands, 1988.

[36] I. Dinur, G. Kindler, R. Raz, and S. Safra, *An Improved Lower Bound for Approximating-CVP*, Combinatorica, 23 (2003), pp. 205–243.

[37] J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, *An extended set of fortran basic linear algebra subroutines*, ACM Trans. Math. Soft., 14 (1988), pp. 1–17.

[38] J. J. Dongarra, R. Hempel, A. J. Hey, and D. W. Walker, *A Proposal for a User-Level, Message-Passing Interface in a Distributed Memory Environment*, Tech. Report UT-CS-93-186, University of Tennessee, Knoxville, TN, USA, 1993.

[39] P. Erdos, J. Hammer, and P. M. Gruber, *Lattice Points*, John Wiley & Sons Inc, hardcover ed., 1989.

[40] A. Ferrari, A. Filipi-Martin, and S. Viswanathan, *The NAS Parallel Benchmark Kernels in MPL*, Tech. Report CS-95-39, Department of Computer Science, University of Virginia, Charlottesville, Virginia 22903, September 1995.

[41] U. Fincke and M. Pohst, *Improved methods for calculating vectors of short length in a lattice, including a complexity analysis.*, Mathematics of Computation, 44(170) (1985), pp. 463–471.

[42] R. Fischlin and J.-P. Seifert, *Tensor-Based Trapdoors for CVP and Their Application to Public Key Cryptography*, in Proceedings of the 7th IMA International Conference on Cryptography and Coding, London, UK, 1999, Springer-Verlag, pp. 244–257.

[43] G. Flores, V. M. García, and A. M. Vidal, *Numerical Experiments on the solution of the Inverse Additive Singular Value Problem*, in Computational Science - ICCS 2005, S. B. . Heidelberg, ed., vol. 3514/2005 of Lecture Notes in Computer Science, 2005, pp. 17–24.

[44] G. J. Foschini, *Layered space-time architecture for wireless communication in a fading environment when using multi-element antennas*, Bell Labs Technical Journal, 1 (1996), pp. 41–59.

[45] G. J. Foschini and M. J. Gans, *On limits of wireless communications in a fading environment when using multiple antennas*, Wireless Personal Communications, 6 (1998), pp. 311–335.

[46] G. J. Foschini, G. D. Golden, F. Reinaldo A. Valenzuela, and P. W. Wolniansky, *Simplified Processing for High Spectral Efficiency Wireless Communication Employing Multi-Element Arrays*, IEEE Journal On Selected Areas in Communications, 17 (1999), pp. 1841–1852.

[47] R. Fourer, D. M. Gay, and B. W. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*, Duxbury Press, November 2002.

[48] U. M. García-Palomares and J. F. Rodríguez, *New sequential and parallel derivative-free algorithms for unconstrained minimization.*, SIAM Journal on Optimization, 13 (2002), pp. 79–96.

[49] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, New York, NY, USA, 1979.

[50] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine: a users' guide and tutorial for networked parallel computing*, MIT Press, Cambridge, MA, USA, 1994.

[51] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, *A User's Guide to PICL: A Portable Instrumented Communication Library*, Tech. Report ORNL/TM-11616, Oak Ridge National Laboratory, September 1990.

[52] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, Academic Press, 1981.

[53] O. Goldreich, S. Goldwasser, and S. Halevi, *Public-Key Cryptosystems from Lattice Reduction Problems*, in CRYPTO '97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology, London, UK, 1997, Springer-Verlag, pp. 112–131.

[54] G. H. Golub and C. F. V. Loan, *Matrix computations*, Johns Hopkins University Press, 3rd ed., 1996.

[55] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*, Addison Wesley, second ed., 2003.

[56] A. Y. Grama and V. Kumar, *Parallel Search Algorithms for Discrete Optimization Problems*, ORSA Journal on Computing, 7 (1995), pp. 365–385.

[57] A. Griewank, *On Automatic Differentiation*, in Mathematical Programming: Recent Developments and Applications, Kluwer Academic Publishers, 1989, pp. 83–108.

[58] M. Grotschel, L. Lovasz, and A. Schrijver, *Geometric Algorithms and Combinatorial Optimization*, Springer Verlag, 2nd ed., September 1993.

[59] P. Gruber and C. Lekkerkerker, *Geometry of Numbers*, North Holland, 2 ed., May 1987.

[60] A. Hassibi and S. Boyd, *Integer parameter estimation in linear models with applications to GPS*, IEEE Transactions on Signal Processing, 46 (1998), pp. 2938–2952.

[61] B. Hassibi, *An efficient square-root algorithm for BLAST*, ICASSP '00: Proceedings of the Acoustics, Speech, and Signal Processing, 2000. on IEEE International Conference, 2 (2000), pp. II737–II740.

[62] B. Hassibi and H. Vikalo, *On sphere decoding algorithm. I. Expected Complexity.*, IEEE Transactions on Signal Processing, 53 (2005), pp. 2806–2818.

[63] B. Helfrich, *Algorithms to construct Minkowski reduced and Hermite reduced lattice bases*, Theor. Comput. Sci., 41 (1985), pp. 125–139.

[64] R. Hooke and T. A. Jeeves, *Direct Search solution of numerical and statistical problems.*, Journal of the Association for Computing Machinery, (1961), pp. 212–229.

[65] P. D. Hough, T. G. Kolda, and V. J. Torczon, *Asyncronous parallel pattern for nonlinear optimization*, SIAM J. Sci. Comput., 23 (2001), pp. 134–156.

[66] J. Jalden, L. G. Barbero, B. Ottersten, and J. S. Thompson, *Full Diversity Detection in MIMO Systems with a Fixed-Complexity Sphere Decoder*, IEEE International Conference on Acoustics, Speech and Signal Processing. ICASSP 2007., 3 (2007), pp. III–49–III–52.

[67] D. S. Johnson, *A Catalog of Complexity Classes*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. A, Elsevier, Amsterdam, 1990, pp. 67–161.

[68] T. Kailath, H. Vikalo, and B. Hassibi, *Space-Time Wireless Systems: From Array Processing to MIMO Communications*, Cambridge University Press, 2006, ch. MIMO receive algorithms, pp. 302–322.

[69] L. Kanal and V. Kumar, *Search in Artificial Intelligence*, Springer-Verlag, London, UK, 1988.

[70] R. Kannan, *Improved algorithms for integer programming and related lattice problems*, ACM Symp. Theory of Computing, (1983).

[71] ——, *Minkowski's convex body theorem and integer programming*, Math. Oper. Res., 12 (1987), pp. 415–440.

[72] C. T. Kelley, *Iterative Methods for Optimization.*, SIAM, 1999.

[73] T. G. Kolda, R. M. Lewis, and V. Torczon, *Optimization by Direct Search: New perspective on Some Clasical and Modern Methods.*, SIAM Review, 3 (2003), pp. 385–442.

[74] T. G. Kolda and V. Torczon, *On the Convergence of Asyncronous Parallel Pattern Search.*, tech. report, Sandia National Laboratories, Livermore, CA, 2002.

[75] A. Korkine and G. Zolotareff, *Sur les formes quadratics*, Math. Annalen, 6 (1873), pp. 366–389.

[76] J. C. Lagarias, J. A. Reeds, M. H. Wright, and P. E. Wright, *Convergence properties of the Nelder-Mead simplex method in low dimensions.*, SIAM Journal on Optimization, 9 (1998), pp. 112–147.

[77] T.-H. Lai and S. Sahni, *Anomalies in Parallel Branch-and-Bound Algorithms*, Communications of the ACM, 27 (1984), pp. 594–602.

[78] A. R. Larrabee, *The P4 Parallel Programming System, the Linda Environment, and Some Experiences with Parallel Computation*, Scientific Programming, 2 (1993), pp. 23–35.

[79] E. L. Lawler and D. E. Wood, *Branch-And-Bound Methods: A Survey*, Operations Research, 14 (1966), pp. 699–719.

[80] A. K. Lenstra, J. Hendrik W. Lenstra, and L. Lovasz, *Factoring polynomials with rational coefficients*, Math. Annalen, 261 (1982), pp. 515–534.

[81] B. Lewis and D. J. Berg, *Multithreaded Programming With PThreads*, Prentice Hall PTR, December 1997.

[82] R. M. Lewis and V. Torczon, *Rank Ordering and Positive Bases in Pattern Search Algorithms.*, tech. report, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA, 1996.

[83] R. M. Lewis, V. Torczon, and M. W. Trosset, *Direct search methods: Then and now.*, Journal Comput. Appl. Math., 124 (2000), pp. 191–207.

[84] Q. Li and Z. Wang, *Improved K-Best Sphere Decoding algorithms for MIMO systems*, in IEEE International Symposium on Circuits and Systems, ISCAS 2006, 2006.

[85] ——, *Reduced Complexity K-Best Sphere Decoder Design for MIMO Systems*, Circuits, Systems and Signal Processing, 27 (2008), pp. 491–505.

[86] Q. Liu and L. Yang, *A Novel Method for Initial Radius Selection of Sphere Decoding*, IEEE Vehicular Technology Conference, 2 (2005), pp. 1280–1283.

[87] S. Lucidi and M. Sciandrone, *On the global convergence of derivative-free methods for unconstrained optimization.*, SIAM Journal on Optimization, 13 (2002), pp. 97–116.

[88] K. I. M. McKinnon, *Convergence of the Nelder-Mead simplex method to a nonstationary point.*, SIAM Journal on Optimization, 9 (1998), pp. 148–158.

[89] D. Micciancio, *The Hardness of the Closest Vector Problem with Preprocessing*, IEEE Transactions on Information Theory, 47 (2001).

[90] D. Micciancio and S. Goldwasser, *Complexity of Lattice Problems*, Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[91] H. Minkowski, *Diskontinuitatsbereich für arithmetische Äquivalenz*, J. Reine und Angewandte Math., 129 (1905), pp. 220–274.

[92] S. G. Nash and A. Sofer, *Linear and Nonlinear Programming*, McGraw-Hill, New York, January 1996.

[93] R.Ñeelamani, S. Dash, and R. G. Baraniuk, *On Nearly Orthogonal Lattice Bases and Random Lattices*, SIAM Journal on Discrete Mathematics, 21 (2007), pp. 199–219.

[94] J. A. Nelder and R. Mead, *A simplex method for function minimization.*, The Computer Journal, 7 (1965), pp. 308–313.

[95] J. Nocedal, *Theory of algorithms for unconstrained optimization*, Acta Numérica, 1 (1992), pp. 199–242.

[96] J. Nocedal and S. J. Wright, *Numerical Optimization*, Springer Series in Operations Research, Springer, New York, 1999.

[97] P. Pacheco, *Parallel Programming With MPI*, Morgan Kaufmann, 1st ed., October 1996.

[98] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[99] P. Pierce, *The NX message passing interface*, Parallel Computing, 20 (1994), pp. 463–480.

[100] M. Pohst, *On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications*, ACM SIGSAM Bull., 15 (1981), pp. 37–44.

[101] E. Polak, *Computational Methods in Optimization: A Unified Approach.*, Academic Press, New York, 1971.

[102] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C. The Art of Scientific Computing.*, Cambridge University Press, second ed., 1992.

[103] M. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw-Hill Science/Engineering/Math, 1 ed., June 2003.

[104] C.-P. Schnorr, *A hierarchy of polynomial time lattice basis reduction algorithms*, Theor. Comput. Sci., 53 (1987), pp. 201–224.

[105] ——, *A more efficient algorithm for lattice basis reduction*, J. Algorithms, 9 (1988), pp. 47–62.

[106] ——, *Factoring Integers and Computing Discrete Logarithms via Diophantine Approximations*, in EUROCRYPT, 1991, pp. 281–293.

[107] C.-P. Schnorr and M. Euchner, *Lattice basis reduction: Improved practical algorithms and solving subset sum problems*, Math. Programming, 66 (1994), pp. 181–191.

[108] A. Schonhage, *Factorization of univariate integer polynomials by diophantine approximation and an improved basis reduction algorithm*, Colloq. Automata, Languages and Programming, (1984), pp. 436–447.

[109] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*, MIT Press, 1996.

[110] W. Spendley, G. Hext, and F. Himsworth, *Sequencial application of simplex designs in optimization and evolutionary operation.*, Technometrics, 4 (1962), pp. 441–461.

[111] A. Stefanov and T. Duman, *Turbo-coded modulation for systems with transmit and receive antenna diversity over block fading channels: system model, decoding approaches, and practical considerations*, IEEE Journal on Selected Areas in Communications, 19 (2001), pp. 958–968.

[112] K. Su, *Efficient Maximum-Likelihood detection for communication over Multiple Input Multiple Output channels*, tech. report, Department of Engineering, University of Cambridge, 2005.

[113] K. Su and I. J. Wassell, *A New Ordering for Efficient Sphere Decoding*, in International Conference on Communications, May 2005.

[114] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek, *The PVM concurrent computing system: Evolution, experiences, and trends*, Parallel Computing, 20 (1994), pp. 531–546.

[115] W. Swann, *Numerical Methods for Unconstrained Optimization*, Academic Press, London and New York, 1972, ch. Direct search methods, pp. 13–28.

[116] V. Tarokh, H. Jafarkhami, and A. R. Calderbank, *Space-time block codes from orthogonal designs*, IEEE Transactions on Information Theory, 45 (1999), pp. 1456–1467.

[117] ——, *Space-time block coding for wireless communications: Performance results.*, IEEE Journal on Selected Areas in Communications, 17 (1999), pp. 451 – 460.

[118] V. Tarokh, N. Seshadri, and A. R. Calderbank, *Space-time codes for high data rate wireless communication: Performance criterion and code construction*, IEEE Transactions on Information Theory, 44 (1998), pp. 744 – 765.

[119] I. E. Telater, *Capacity of multi-antenna gaussian channels*, Europ. Trans. Telecommun., (1999), pp. 585–595.

[120] V. Torczon, *Multi-Directional Search: A Direct Search Algorithm for Parallel Machines.*, tech. report, Department of Computational and Applied Mathematics, Rice University, Houston, TX, 1990.

[121] ——, *On the convergence of the multidirectional search algorithm.*, SIAM Journal on Optimization, 1 (1991), pp. 123–145.

[122] ——, *On the convergence of pattern search algorithms.*, SIAM Journal on Optimization, 7 (1997), pp. 1–25.

[123] M. W. Trosset, *I know it when I see it: Toward a definition of direct search methods.*, SIAG/OPT Views-and-News: A Forum for the SIAM Activity Group on Optimization, (1997).

[124] ——, *On the Use of Direct Search Methods for Stochastic Optimization.*, tech. report, Department of Computational and Applied Mathematics, Rice University, Houston, TX, 2000.

[125] R. A. Trujillo, A. M. Vidal, and V. M. García, *Parallel Optimization Methods Based on Direct Search*, in Computational Science - ICCS 2006, vol. 3991/2006 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 324–331.

[126] R. A. Trujillo, A. M. Vidal, and V. M. García, *Métodos Secuenciales y Paralelos de Optimización basados en Búsqueda Directa*, Revista Cubana de Ciencias Informáticas, 3 (2007), pp. 26–37.

[127] R. A. Trujillo, A. M. Vidal, V. M. García, and A. González, *Parallelization of Sphere-Decoding Methods*, in High Performance Computing for Computational Science - VECPAR 2008, vol. 5336/2008 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2008, pp. 2–12.

[128] L. W. Tucker and A. Mainwaring, *CMMD: Active Messages on the CM-5*, December 1993.

[129] R. J. Vaccaro, *SVD and Signal Processing II: Algorithms, Analysis and Applications*, Elsevier Science Inc., New York, NY, USA, 1991.

[130] P. van Emde Boas, *Another NP-complete problem and the complexity of computing short vectors in a lattice*, tech. report, Mathematische Instiut, University of Amsterdam, Amsterdam, The Netherland, 1981.

[131] H. Vikalo and B. Hassibi, *On the sphere-decoding algorithm II. Generalizations, second-order statistics, and applications to communications*, IEEE Transactions on Signal Processing, 53 (2005), pp. 2819 – 2834.

[132] H. Vikalo, B. Hassibi, and T. Kailath, *Iterative decoding for MIMO channels via modified sphere decoding*, IEEE Transactions on Wireless Communications, 3 (2004), pp. 2299– 2311.

[133] B. W. Wah, G.-J. Li, and C.-F. Yu, *Parallel algorithms for machine intelligence and vision*, Springer-Verlag New York, Inc., New York, NY, USA, 1990, ch. Multiprocessing of combinatorial search problems, pp. 102–145.

[134] F. H. Walters, L. R. Parker, Jr., S. L. Morgan, and S.Ñ. Deming, *Sequential Simplex Optimization*, CRC Press, Inc., Boca Raton, Florida, 1991.

[135] P. W. Wolniansky, G. J. Roschini, G. D. Golden, and R. A. Valenzuela, *V-BLAST: an architecture for realizing very high data rates over the rich-scattering wireless channel.*, in 1998 Int. Symp.Sig. Sys. Elect. (ISSSE'98), 1998.

[136] M. Wright, *Direct Search Methods: One Scorned, Now Respectable.*, tech. report, Computing Science Research Center, AT&T Bell Laboratories, Murray Hill, NJ, 1996.

[137] D. Wübben, R. Böhnke, V. Kühn, and K.-D. Kammeyer, *MMSE Extension of V-BLAST based on Sorted QR Decomposition*, in IEEE Semiannual Vehicular Technology Conference (VTC2003-Fall), vol. 1, Orlando, Florida, USA, October 2003, pp. 508–512.

[138] W. Zhao and G. B. Giannakis, *Sphere Decoding Algorithms With Improved Radius Search*, IEEE Transactions on Communications, 53 (2005), pp. 1104–1109.